PROCEEDINGS

C++ Conference

Denver, CO

October 17-21, 1988

For additional copies of these proceedings write

USENIX Association
P.O. Box 2299
Berkeley, CA 94710 USA

Price: $30.00

Outside the U.S.A. and Canada please add
$20 per copy for air printed matter postage.

Second Printing (December 1988)

Program and Table of Contents

C++ Conference

October 17-21, 1988

Monday and Tuesday, October 17 & 18

Tutorials 9:00 - 5:00

Wednesday, October 19 — Salons E & F

Opening Session 9:00 - 10:30

Keynote Speech
W. N. Joy, Sun Microsystems

Break 10:30 - 11:00

Technique 11:00 - 12:30

Lunch 12:30 - 2:00

Databases and File Systems 2:00 - 3:30

Break 3:30 - 4:00

Thursday, October 20 — Salons E & F

Friday, October 21

Parameterized Types for C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Type parameterization is the ability to define a type in terms of another, unspecified, type. Versions of the parameterized type may then be created for several particular parameter types. A language supporting type parameterization allows specification of general container types such as *list*, *vector*, and *associative array* where the specific type of the elements is left as a parameter. Thus, a parameterized class specifies an unbounded set of related types; for example: list of int, list of name, list of shape, etc. Type parameterization is one way of making a language more extensible.

In the context of C++ , the problem are

[1] Can type parameterization be easy to use?

[2] Can objects of a parameterized type be used as efficiently as objects of a "hand-coded" type?

[3] Can a general form of parameterized types be integrated into C++ ?

[4] Can parameterized types be implemented so that the compilation and linking speed is similar to that achieved by a compilation system that does not support type parameterization?

[5] Can such a compilation system be simple and portable?

A design is presented for which the answer to all of these questions is *yes*. The implementation of this scheme is a fairly simple extension of current C++ implementations.

WARNING: The scheme for providing parameterized types described here is not implemented. It is not part of the C++ language, nor is there any guarantee that it ever will be.

1 Introduction

For many people, the largest single problem using C++ is the lack of an extensive standard library. A major problem in producing such a library is that C++ does not provide a sufficiently general facility for defining "container classes" such as lists, vectors, and associative arrays. There are two approaches for providing such classes/types:

[1] The Smalltalk approach: rely on dynamic typing and inheritance.

[2] The Clu approach: rely on static typing and a facility for arguments of type *type*.

The former is very flexible, but carries a high run-time cost, and more importantly defies attempts to use static type checking to catch interface errors. The latter approach has traditionally given rise to fairly complicated language facilities and also to slow and elaborate compile/link time environments. This approach also suffered from inflexibility because languages where it was used, notably Ada, had no inheritance mechanism.

Ideally we would like a mechanism for C++ that is as structured as the Clu approach with ideal run-time and space requirements, and with low compile-time overheads. It also ought to be as flexible as Smalltalk's mechanisms. The former is possible; the latter can be approximated for many important cases.

Note that C++ appears to have sufficient expressive power to cope with the demands of library writing provided there is a single basic kind of object, such as a character (for string manipulation, pattern matching, character I/O, etc.), a double precision floating point number (for engineering libraries), or a bitmap (for graphics libraries). The "container class problem" is particularly serious, though, since container classes are needed to specify all but the simplest interfaces; they are the "glue" for larger systems.

2 Class Templates

A C++ parameterized type will be referred to as a class template. A class template specifies how individual classes can be constructed much like the way a class specifies how individual objects can be constructed. A vector class template might be declared like this:

```
template<class T> class vector {
        T* v;
        int sz;
public:
        vector(int);
        T& operator[](int);
        T& elem(int i) { return v[i]; }
        // ...
};
```

The `template <class T>` prefix specifies that a template is being declared and that an argument `T` of type *type* will be used in the declaration. After its introduction, `T` is used exactly like other type names within the scope of the template declaration. Vectors can then be used like this:

```
vector<int> v1(20);
vector<complex> v2(30);

typedef vector<complex> cvec;    // make cvec a synonym for vector<complex>
cvec v3(40);                     // v2 and v3 are of the same type

v1[3] = 7;
v2[3] = v3.elem(4) = complex(7,8);
```

Clearly class templates are no harder to use than classes. The complete names of instances of a class template, such as `vector<int>` and `vector<complex>`, are quite readable. They might even be considered more readable than the notation for the built-in array type: `int[]` and `complex[]`. When the full name is considered too long, abbreviations can be introduced using `typedef`.

It is only trivially more complicated to declare a class template than it is to declare a class. The keyword `class` is used to indicate arguments of type *type* partly because it appears to be an appropriate word, partly because it saves introducing a new keyword. In this context, `class` means "any type" and not just "some user-defined type."

The `<...>` brackets are used in preference to the parentheses `(...)` partly to emphasize the different nature of template arguments (they will be evaluated at compile time) and partly because parentheses are already hopelessly overused in C++.

The keyword `template` is introduced to make template declarations easy to find, for humans and for tools, and to provide a common syntax for class templates and function templates.

3 Member Function Templates

The operations on a class template must also be defined. This implies that in addition to class templates, we need function templates. For example:

```
template<class T> T& vector<T>::operator[](int i)
{
        if (i<0 || sz<=i) error("vector: range error");
        return v[i];
}
```

A function template is a specification of a family of functions; `template<class T>` specifies that `T` is a template argument (of type *type*) that must somehow be supplied to specify a particular function.

Note that you don't usually have to specify the template arguments to use a function template. For example, the template argument for `vector<T>::operator[]` will be determined by the vector to which the subscripting operation is applied:

```
vector<int> v1(20);
vector<complex> v2(30);

v1[3] = 7;              // vector<int>::operator[]()
v2[3] = complex(7,8);   // vector<complex>::operator[]()
```

Member functions of a class template are themselves function templates with the template arguments specified in the class templates. Function templates and member function templates will be discussed in greater detail in §9 and §12.

4 Outline of an Implementation

The basic idea for an implementation that incurs no additional costs in run-time or space compared with "hand coding" is to "macro-expand" a template for each different set of template arguments with which it is used. Naturally, template expansion is not really/just macro expansion; it obeys proper scope and syntax rules. Names such as `vector<int>` can be encoded into composite class names such as `__PTvector_int`.

The example above expands into:

```
class __PTvector_int {
        int* v;
        int sz;
public:
        __PTvector_int(int);
        int& operator[](int);
        int& elem(int i) { return v[i]; }
        // ...
};

class __PTvector_complex {
        complex* v;
        int sz;
public:
        __PTvector_complex(int);
        complex& operator[](int);
        complex& elem(int i) { return v[i]; }
        // ...
};

__PTvector_int v1(20);
__PTvector_complex v2(30);
__PTvector_complex v3(40);

v1[3] = 7;
v2[3] = v3.elem(4) = complex(7,8);
```

A compiler need not have a separate template expansion pass. Since the information to do such an expansion exists in the compiler's tables, the appropriate actions can simply be taken at the

proper places in the analysis and code generation process.

In addition to this expansion mechanism, a facility is needed for detecting which member functions have been used for which instances of a parameterized type. The example above used:

```
__PTvector_int::__PTvector_int();                    // constructor
__PTvector_complex::__PTvector_complex();            // constructor
__PTvector_int::operator[]();                        // subscripting
__PTvector_complex::operator[]();                    // subscripting
__PTvector_complex::elem();
```

Note that the full list of such functions for a program can be known only after examining every source file. The linker provides a form of this list as part of its list of undefined objects and functions.

The definition of an operation on a class template might look like this:

```
template<class T> T& vector<T>::operator[](int i)
{
        if (i<0 || sz<=i) error("vector: range error");
        return v[i];
}
```

From this, the following two function definitions will have to be generated:

```
int& __PTvector_int::operator[](int i)
{
        if (i<0 || sz<=i) error("vector: range error");
        return v[i];
}

complex& __PTvector_complex::operator[](int i)
{
        if (i<0 || sz<=i) error("vector: range error");
        return v[i];
}
```

This approach ensures that no run-time efficiency is lost compared to "hand-coding". Code space might wasted by creating separate copies of functions that could have shared implementation. For example, `vector<int>` and `vector<unsigned>` need not have separate subscripting operations. Such waste can, if necessary, be reduced through suitable coding practices (see § 11) and/or through a clever compile time environment.

A programmer can provide a definition for a particular version of an operation on a class by specifying the template argument(s) in a function definition:

```
int& vector<int>::operator[](int i) { return v[i]; }
```

The general version of such a function as defined by its template will be used to create a function for a particular argument type only when no user-provided version is specified for that type.

Replacing the default implementation of a function as defined by a template is useful where implementations with greater precision, higher efficiency, etc. can be provided given some understanding of a particular type. It may also be useful for debugging and for supplying different versions of a function to different parts of a program (using `static` functions).

5 Some Design Considerations

Let us consider a few choices that were made to write the example above:

[1] Should all template arguments be of type *type*?

[2] Should a user be required to specify the set of operations that may be used for a template argument of type *type*?

[3] Should a user be required to explicitly declare what versions of a template can be used in a program?

[4] Should it be possible for a user to declare variables of type *type*?

The answer to all (in the context of C++) is *no*. Let us examine them in turn.

Template Arguments

"Should all template arguments be of type *type*?" No, there appear to be useful examples of type parameters of "normal" types. For example, a vector template might be parameterized with an error handling function:

```
typedef void (*PF)(char*);

template<class T, PF error> class vector {
        T* v;
        int sz;
public:
        T& operator[](int i) {
                if (i<= || sz<=i) error("vector: range error");
                return v[i];
        }
};

void my_error_fct() { ... }
vector<complex,&my_error_fct> v(10);
```

This example implies that default arguments might be useful:

```
template <class T, PF error=&standard_error_fct> class vector { ... }
```

Another example is a buffer type with a size argument:

```
template<class T, int sz=128> class buffer {
        T v[sz];
        // ...
};

void f()
{
        buffer<char> buf1;
        buffer<complex,20> buf2;
        // ...
}

buffer<char*,1000> glob;
```

Making `sz` an argument of the template `buffer` itself rather than of its objects implies that the size of a `buffer` is known at compile time so that a `buffer` can be allocated without use of free store. It appears that default arguments will be at least as useful for template arguments as they are for ordinary arguments. Default arguments of type *type* might even be useful:

```
template<class T, class TEMP = double> class store {
        // ...
        T sum() { TEMP sum = 0; ... return sum; }
};

store<int,long> istore;
store<float> fstore;
```

These examples demonstrate that the range of templates with which a type can be parameterized should be restricted only if there are compelling arguments that the restriction will significantly ease the implementation of templates. I see no such argument.

Type Argument Attributes

"Should a user be required to specify the set of operations that may be used for a template argument of type *type*?" For example:

```
        // The operations =, ==, <, and <=
        // must be defined for an argument type T

template <
                class T {
                        T& operator=(const T&);
                        int operator==(const T&, const T&);
                        int operator<=(const T&, const T&);
                        int operator<(const T&, const T&);
                };
        >
        class vector {
        // ...
        };
```

No. Requiring the user to provide such information decreases the flexibility of the parameterization facility without easing the implementation or increasing the safety of the facility.

Consider `vector<T>`. To provide a sort operation one must require that type `T` has some order relation. This is not the case for all types. If the set of operations on `T` must be specified in the declaration of `vector` one would have to have two vector types: one for objects of types with an ordering relation and another for types without one. If the set of operations on `T` need not be specified in the declaration of `vector` one can have a single vector type. Naturally, one still cannot sort a vector of objects of a type `glob` that does not have an order relation. If that is tried, the generated sort function `vector<glob>::sort()` would be rejected by the compiler.

It has been argued that it is easier to read and understand parameterized types when the full set of operations on a type parameter is specified. I see two problems with this: such lists list would often be long enough to be de facto unreadable and a higher number of templates would needed for many applications.

Should experience show a need for specifying the operations on a parameterized type then such a facility can be easily and compatibly added later.

Source Code

There might be a more fundamental reason for requiring that the operations performed on a template argument of type *type* should be listed in the template declaration. The implementation technique outlined here achieves near optimal run-time characteristics by requiring the complete source code of a template to be available to the compiler when processing a use of the template. In some contexts, this is considered a deficiency and an implementation of templates that requires only the object code for functions implementing the function templates would be preferable.

At first glance it would appear that requiring the full set of operations on a template argument to be specified would make it significantly easier to produce such an implementation. In this case, a function template would be implemented by code using calls through vectors of function pointers to perform operations on template arguments of type *type*. The specification of the set of operations for a *type* argument would provide the definition for such vectors. Such an implementation would trade run-time for compile and link time, but would be semantically equivalent to the implementation scheme presented here.

Could an implementation along these lines be provided without requiring the user to list the set of operations needed for each function template argument of type *type*? I think so. Given a function template, the compiler can create a vector layout for the required set of operations without the help of a user. Given the full set of function definitions for the members of a class, the compiler can again create a vector layout for the required set of operations without the help of a user. If the compile and link environment cannot provide such a list a less optimized scheme where each member function has its own vector of operations can be used.

It thus appears that both implementation styles can be used even in the absence of template argument attribute lists so that we need not require them to preserve the implementers' freedom of action. It might be noticed that a virtual function table is in many ways similar to a vector of operations for a template so that the benefits of the vector of operations approach can often be

achieved by a coding style relying on virtual functions rather than the expansion of function templates. Class `pvector` presented in §11 is an example of this.

Type Instantiation

"Should a user be required to explicitly declare what versions of a template can be used in a program?" For example, should one require the use of an operation like Ada's `new`? No. Such a requirement would increase the size of the program text and decrease the flexibility of the template facility without yielding any benefits to the programmer or the implementer.

Type Variables

"Should it be possible for a user to declare variables of type *type*?" For example:

```
type t = int;

void f(type t)
{
        switch (t) {
        case int:
                ...
        case char*:
                ...
        case complex:
                ...
        default:
                ...
        }
}
```

Such a facility would be useful in many contexts, but does not appear suitable for C++. In particular, it is not possible to assign integer values to represent constants of type *type* such as `int`, `line_module*`, `double(*)(complex*,int)`, and `vector<complex>` while maintaining the current style of separate compilation. Since the C++ type system is open such assignment of values in general requires an unbounded number of bits to represent a type. In practice, even simple cases require lots of bits (the current cfront scheme for encoding function types in character strings regularly uses dozens of characters) or some system of hashing involving a database of types. Furthermore, the introduction of such variables would require an order of magnitude greater changes to the C++ language and its implementations than the scheme (without type variables) described here.

6 Type Inquiries

It would be possible to enable a programmer to inquire about properties of a template argument of type *type*. This would allow the programmer to write code that depends on specific properties of the actual types used.

An Inquiry Operator

Consider providing a print function for a vector type that sorts the elements before printing if and only if sorting is possible. A facility for inquiring if a certain operation, such as <, can be performed on objects of a given type can be provided. For example:

```
template<class T> void vector<T>::print()
{
        if (?T::operator<) sort();      // if (T has a <) sort_this_vector
        for (int i=0; i<sz; i++) { ... }
}
```

Because the < operation is defined for *int*s, printing of a `vector<int>` gives rise to an expansion:

```
void __PTvector_int::print()
{
        sort();         // that is, this->sort()
        for (int i=0; i<sz; i++) { ... }
}
```

On the other hand, printing a `vector<glob>` where the < operation is not defined for `glob`s gives rise to an expansion:

```
void __PTvector_glob::print()
{
        for (int i=0; i<sz; i++) { ... }
}
```

Tests on expressions of the form *?typ::oper* ("does type *typ* have an operation *oper*?") must be evaluated at compile time and can be part of constant expressions.

It would probably be wise *not* to include such a type inquiry feature in the initial experimental implementation but to wait and see what properties (if any) programmers would find useful. Potentially every aspect of a type known to the compiler can be made available to the programmer; `sizeof` is a most rudimentary version of this kind of facility.

The absence of a type inquiry facility will be compensated for by the ability to define a function for a particular set of template arguments, thus overriding the generation of the "standard" version from the template. Furthermore, it can sometimes be preferable to define separate templates to represent the different concepts. For example, one might have both a `vector<T>` class and a `sorted_vector<T>` class derived from it.

The `typeof` Operator

Writing code where the control flow depends of the properties of a type parameter doesn't appear to be necessary, but defining variables of types dependent on type parameters does. Given a template argument of type *type*, `T`, one can express a variety of derived types using the declarator syntax; for example, `T*`, `T&`, `T[10]`, `T(*)(T,T)`. One can also express types obtained by template expansion such as `vector<T>`. However, this does not conveniently express all types one might like. In particular, the ways of expressing types that depends on two or more template arguments are weak. To compensate, one might introduce a `typeof` operator that yields the type of its argument. For example:

```
template<class X, class Y> void f(X x, Y y)
{
        typeof(x*y) temp = x*y;
        // ...
}
```

It would probably be wise *not* to introduce a `typeof` operator before gaining more experience. The uses of `typeof` appears to be quite limited and the scope for misuses large. In particular, `typeof` appears more suited for the writing of macros (which templates are designed to replace in many contexts) than for templates and heavy use of typeof will reduce the compilers ability to pinpoint type errors.

7 More about Implementation

So how can we generate the proper code for definitions of operations on a template for a given set of arguments? Assume that we know that versions of `vector`'s subscripting operation

```
template<class T> vector<T>::operator[](int) { ... }
```

are needed for `T==int` and `T==complex`. How can we create the proper expansions (as presented above)?

We might have a compiler option, `-X`, for creating such expansions. Assuming that the definitions for `vector`'s member functions resides in a file called `vector.c`, one might call the

compiler like this:

```
CC -X "vector<int>" vector.c
CC -X "vector<complex>" vector.c
```

and have the appropriate `.o` files created. This would create not only the required subscript operator functions but also functions for any other vector operation that has its definition stored in `vector.h`. The strategy for splitting a program into separately compiled parts is in the hands of the programmer. Where a finer granularity is required of `.o` files for a library, the programmer can handle it using standard C library techniques.

Note that an expansion using the template expansion option, `-X`, may give rise to a program that uses an instance of a template that has not already been used in the program. This implies that another stage of "missing template implementation detection" is required after each expansion. Expansion is really a recursive activity. The depth of this recursion will typically be 1, though. It will be necessary to have a mechanism protecting against recursive expansion. For example:

```
template<class T> void f(T a) { T* p; ... f(p); }
```

Naturally, one would try to ensure that `CC -X` is used to generate `.o` files only for definitions of templates when

[1] a new template was used, or

[2] a new set of template arguments was used, or

[3] the declaration of a template was changed.

I imagine that after a short startup period all the necessary `.o` files for templates for a program/project will reside in a library and not interfere with the compilation process. When a program/project reaches this state the compilation overhead incurred by using templates becomes negligible.

Tools for Ensuring Consistent Linking

Consider having the tools described above:

[1] a C++ compiler handling the expansion of class templates into class declarations, and

[2] a `-X` option on this compiler to handle the expansion of function templates into function definitions.

One could then compile a C++ program using templates. A little manual intervention would be needed to get a complete program to link and load.

What additional tools would be needed to

[1] guarantee consistent and complete expansion and linking?

[2] make programming reasonably convenient?

I conjecture that [1] is perfectly feasible, but non-trivial, where portability across operating systems, compile and link time efficiency, and flexibility are all required. I also conjecture that very little is needed to achieve [2]. Experience using such a system is clearly needed, but it might well be sufficient to modify a tool with access to the complete compiled program, such as `munch` or the linker itself, to produce

[1] a list of function definitions required, or

[2] a list of files for which `CC -X` needs to be run (assuming some correspondence between type names and file names), or

[3] a make script for running `CC -X` for an appropriate set of files.

It would also be important to ensure that `CC` produces readable error messages when an operation is applied to a particular template argument of type *type* for which it is not defined. For example:

```
"foo.c", line 144: error: operator<= applied to glob in vector<glob>::sort()
```

This discussion of how one might provide a minimal and portable mechanism supporting templates in C++ should not be taken as an indication that such a mechanism provides the ideal programming environment. On the contrary, it is exactly a *minimal* facility. Much better facilities can be built (think of a smart `make`, an incremental compiler, a Smalltalk-like browser, etc.),

However, a minimal facility *must* exist to ensure portability of C++ programs between all implementations since there is no hope that a single maximal programming environment will ever be agreed on and implemented on every system supporting C++ .

8 Function Templates

In addition to providing class templates, it is necessary to provide function templates. Consider providing a general `sort` function:

```
template<class T> void sort(vector<T>);
```

Given a vector `v`, one might call such a function like this:

```
sort(v);
```

The compiler can deduce the type of the sort function from the type of the vector. For example, had `v` been declared

```
vector<int> v(10);
```

the sort function `sort<int>` would have been required. On the other hand had the declaration of `v` been

```
vector<double> v(2000);
```

the sort function `sort<double>` would have been used.

Overloading

Declaring a function template is simply a way of declaring a whole bundle of overloaded functions at one time. This implies that we can use functions with arguments that can be distinguished by the overloaded function resolution mechanism only. The following function cannot be used because it takes no argument:

```
template<class T> T* create() { return (T*) malloc(sizeof(T)); }
```

The C++ syntax could be extended to cope with this by allowing the full generality of the *name<type>* notation so that template arguments could be supplied explicitly in a call:

```
int* pi = create<int>();        // create_int()
char* pc = create<char>();      // create_char()
```

Unless programmers define templates sensibly this form of resolution can become quite cryptic:

```
template<class X, class Y> f(Y,X);  // template argument order differs
                                    // from function argument order
 ...
f<char*,int>(1,"asdf");
```

I think it would be wise not to include any explicit resolution method in an initial implementation. I suspect that the implicit resolution provided by the overloaded function resolution rules are sufficient – and more elegant – in almost all cases and it is not obvious that a mechanism for explicit overloading is worth the added complexity.

Allowing explicit resolution would imply that a C++ compiler should treat function template names differently from other names and similarly to the way keywords and class names are treated. For example, without special rules for template names the last expression above would be parsed as two comparisons and a parenthesized comma expression:

```
(g<123)>(vv,10);
```

A Problem

Consider writing a function `apply()` that applies another function to all the elements of a vector. A traditional first cut would look something like this:

```
template<class T> void apply(vector<T>& v, T& (*g)(T&))
{
        for (int i = 0; i<v.size(); i++) v[i] = (*g)(v[i]);
}
```

This follows the C style of using a pointer to function. Potential problems with this are

[1] efficiency, because there can be no inline expansion of the applied function, and

[2] generality, because standard operations of built-in types such as - and ~ for `int`s cannot be applied.

Naturally, these are not problems to all people. However, an ideal template mechanism will provide solutions.

A Solution

One might consider the function to be applied by `apply()` a template argument rather than a function argument:

```
template<class T, T& (*g)(T&)> void apply(vector<T>& v)
{
        for (int i = 0; i<v.size(); i++) v[i] = (*g)(v[i]);
}
```

To call `apply()` one must specify the function to be applied. Since this version of `apply()` takes only a single `vector` argument the syntax for disambiguating an overloaded function call using <...> must be used:

```
class X { ... };

vector<X> v2(200);

inline void hh(X&) { ... };
void gg(X&);     // not inline

apply<X,hh>(v2);
apply<X,gg>(v2);
```

Clearly, the `X` is redundant and not elegant. Since in principle each such call of `apply()` results in writing a new function `apply()` inlining can be applied where sufficient information is available. Consequently, one would expect a C++ compiler to inline `hh()` in the first call in the example above and generate a standard function call of `gg()`. The fact that function pointers and not functions are passed in C++ is at most a minor annoyance for the compiler writer.

Operators for built-in types can be considered inline functions in this context:

```
vector<int> v(100);
apply< int, &int::operator-- >(v);
```

However, as for the explicit resolution scheme itself, it remains to be seen if this degree of sophistication and complexity is actually needed.

9 Syntax Issues

Consider the declarations:

```
template<class T> class vector { ... };
template<class T> T* index<class T>(vector<T>,int);
```

[1] Why use the `template` keyword?

[2] Why use <...> brackets and not parentheses?

[3] Why use the `class` keyword?
[4] What is the scope of a template argument?

The `template` keyword

If a keyword is to be used `template` seems to be a reasonable choice, but it is actually not necessary to introduce a new keyword at all. For class templates, the alternative syntax seems more elegant at first glance:

```
class vector<class T> {          // possible alternative class syntax
        ...
}
```

Here the template arguments are placed after the template name in exactly the way they are in the use of a class template:

```
vector<int> vi(200);
vector<char*> vpc(400);
```

The function syntax at first glance also looks nicer without the extra keyword:

```
T& index<class T>(vector<T> v, int i) { ... }
```

There is typically no parallel in the usage, though, since function template arguments are not usually specified explicitly:

```
int i = index(vi,10);
char* p = index(vpc,29);
```

However, there appears to be nagging problems with this "simpler" syntax. It is too clever. It is relatively hard to spot a template declaration in a program because the template arguments are deeply embedded in the syntax of functions and classes and the parsing of some function templates is a minor nightmare. It is possible to write a C++ parser that handles function template declarations where a template argument is used before it is defined, as in `index()` above. I know, because I wrote one, but it is not easy nor does the problem appear amenable to traditional parsing techniques. In retrospect, I think that not using a keyword and not requiring a template argument to be declared before it is used would result in a set of problems similar to those arising from the clever and convoluted C and C++ declarator syntax.

<...> vs (...)

But why use brackets instead of parentheses? As mentioned before, parentheses already have many uses in C++. A syntactic clue (the <...> brackets) can be useful for reminding the user about the different nature of the type parameters (they are evaluated at compile time). Furthermore, the use of parentheses could lead to pretty obscure code:

```
template(int sz = 20) class buffer {
        buffer(sz)(int i = 10);
        // ...
};

buffer b1(100)(200);
buffer b2(100);          // b2(100)(10) or b2(20)(100)?
buffer b3;               // legal?
```

These problems would become a serious practical concern if the notation for explicit disambiguation of overloaded function calls were adopted. The chosen alternative seems much cleaner:

```
template<int sz = 20> class buffer {
        buffer(sz)(int i = 10);
        // ...
};

buffer b1<100>(200);
buffer b2<100>;          // b2<100>(10)
buffer b3;               // b3<20>(10)
buffer b4(100);          // b4<20>(100)
```

The `class` keyword

Unfortunately, the ideal word for introducing the name of a parameter of type *type*, that is, `type` cannot be used; `type` appears as an identifier in too many programs. Why use the `class` keyword then? Why not? Classes are already types to the extent that the built-in types can be considered second class citizens in some contexts (you cannot derive a class from a built in type, you cannot take the address of an operation on a built-in type, etc.). What is done here is simply to use `class` in a slightly more general form than is done elsewhere.

Scope of Template Argument Names

The scope of a template argument name is the template declaration and the template name obeys the usual scope rules:

```
const int T;

template<class T>        // hides the const int T
class vector {
        int sz;
        T* v;
public:
        // ...
};

int T2 = T;              // here const int T is visible again
```

Template declarations may not be declaration lists:

```
template<class T> f(T*), g(T);  // error: two declarations
```

This restriction is made to avoid users making unwarranted assumptions about relations between the template arguments in the different templates.

10 Templates and Typedef

The template concept is easily extended to cover all types. For example:

```
template<class T, int i> T array[i];
 ...
array<int,10> v;         // array of 10 ints
```

This allows great freedom in defining type names. In particular, a `template` without arguments is equivalent to a `typedef`. For example:

```
template<> int I1;
typedef int I2;

I1<> x;          // ``x'' is an int
I2 y;            // ``y'' is an int
int z;           // ``z'' is an int
```

For example, it follows that `x` and `z` in the example above are both of the same type (`int`) . `I1<>` is simply a rather unusual way of writing `int`.

11 Type Equivalence

Consider:

```
template<class T, int i> class X {
        T vec[i];
        // ...
};

array<int,10> x;
array<int,10> y;
array<int,11> z;
```

Here, x and y is of the same type, but z is of the different type. Since the template arguments used in the declarations of x and y are identical they refer to the same class. Naturally, only a single class declaration is generated by a C generating C++ compiler. On the other hand, the template arguments used in the declaration of z differs and gives rise to a different class.

Different template arguments give rise to different classes even if the argument is used in a way that does not affect the type of the generated class:

```
template<class T, int i> class Y {
public:
        foo() { int buf[i]; ... }
};

Y<int,10> xx;
Y<int,10> yy;
Y<int,11> zz;
```

Template arguments must be types, constants, or integer expression that can be evaluated at compile time.

12 Derivation and Templates

Among other things, derivation (inheritance) ensures code sharing among different types (the code for a non-virtual base class function is shared among its derived classes). Different instances of a template do not share code unless some clever compilation strategy has been employed. I see no hope for having such cleverness available soon. So, can derivation be used to reduce the problem† of code replicated because templates are used? This would involve deriving a template from an ordinary class. For example:

```
template<class T> class vector {                    // general vector type
        T* v;
        int sz;
public:
        vector(int);
        T& elem(int i) { return v[i]; }
        T& operator[](int i);
        // ...
};
```

† If that really is a problem: memory is cheap, etc. I think it is a problem and will remain so for the foreseeable future. People's expectations of computers have consistently outstripped even the astounding growth in hardware performance.

```
template<class T>
class pvector : vector<void*> {          // build all vector of pointers
                                         // based on vector<void*>
public:
        pvector(int i) : (i) {}
        T*& elem(int i) { return (T*&) vector<void*>::elem(i); }
        T*& operator[](int i) { return (T*&) vector<void*>::operator[](i); }
        // ...
};

pvector<int*> pivec(100);
pvector<complex*> icmpvec(200);
pvector<char*> pcvec(300);
```

The implementations of the three vector of pointer classes will be completely shared. They are all implemented exclusively through derivation and inline expansion relying on the implementation of `vector<void*>`. The `vector<void*>` implementation is a good candidate for a standard library.

I conjecture that many class templates will in fact be derived from another template. For example:

```
template<class T> class D : B<T> {
        ...
};
```

This also ensures a degree of code sharing.

13 Members and Friends

Here are some more details:

Member Functions

A member function of a class template is implicitly a template with the template arguments of its class. Consider:

```
template<class T> class C {
        T p;
        T m1() { T a = p; p++; return a; }
};

C<int> c1;
C<char*> c2;

int i = c1.m1();     // int C<int>::m1() { int a = p; p++; return a; }

char* s = c2.m1();  // char* C<char*>::m1() { char* a = p; p++; return a; }
```

These two calls of `m1()` gives rise to two expansions of the definition of `m1()`.

Naturally a member template may also be declared:

```
template<class T> class C {
        template<class TT> void m(TT*,T*);
};
```

This case will be discussed below. However, explicit use of class template arguments in member function names is unnecessary and illegal:

```
template<class T> class C {
        T m<T>();                    // error
};

template<class T> C<T>::m<T>() { ... }       // error

template<class T> C<T>::m() { ... }          // correct
```

This also applies to constructors:

```
template<class T> class C {
        C();                         // correct, a constructor
        C<T>(int);                   // error constructor
};

template<class T> C<T>::C() { ... }      // correct
```

To avoid confusion it is not legal to define a template as a member with the same template argument name as was used for the class template:

```
template<class T> class C {
        template<class T> T m(T*);                  // error
};
```

Friend Functions

A friend function differs from other functions only in its access to class members. In particular, a friend of a class template is not implicitly a template. Consider:

```
template<class T> class C {
        friend f1(T a);
        template<class TT> friend f2(TT a);
};
```

The definitions of `f1()` and `f2()` are legal, but clearly not equivalent.

The `friend` declaration of `f1(T)` specifies that for all types `T`, `f1<T>` is a friend of `C<T>`. For example, `f1<int>` is a friend of `C<int>`. However, `f1<int>` is not a friend of `C<double>`. The definition of `f1()` would probably look something like this:

```
template<class TT> f1(TT a) { ... };
```

The friend `f1()` need not be a template, but if it isn't the programmer might have a tedious time constructing the necessary set of overloaded functions "by hand."

The declaration of `f2()` specifies that for all types `T` and `TT`, `f2<TT>` is a friend of `C<T>`. For example `f2<int>` is a friend of `C<double>`.

Note that a friend function of a parameterized class need not itself be parameterized:

```
template<class T> class C {
        static int i;
        friend f() { i++; }
};
```

Static Members

Each version of a class template has its own copy of the static members of the class:

```
template<class T> class C { static T a; static int b; ... };

C<int> xx;
C<double> yy;
```

This implies allocation of the static variables:

```
static int C<int>::a;
static int C<int>::b;

static double C<double>::a;
static int C<double>::b;
```

Similarly, each version of a parameterized function has its own copy of static local variables:

```
template<class T> f() { static T a; static int b; ... };
```

Friend Classes

Friend classes can (as usual) be declared as a shorthand for declaring all functions friends:

```
template<class T> class C {
        friend template<class TT> class X;      // all X<TT>s
        friend class Y<T>;                      // only Y<T>
        friend class Z<int>;                    // only Z<int>
};
```

14 Examples of Templates

Here are some more examples of potentially useful templates. Versions of many of the templates used as examples in this paper have been created using macros and actually used in real programs. "Faking" templates using macros have been a major design technique for the template facilities. In this way the language facilities could be designed in parallel with the key examples and techniques they were to support.

An associative array:

```
template<class E, class I> class Map {
        // arrays of Es indexed by Is
        // ...
        E& operator[](I);
};
```

A "record" stream; the usual stream of characters is a special case:

```
template<class R> class ostream {
        // ...
        ostream<R>& operator<<(R&);     // output an R
};
```

An array for mapping information from files into primary memory:

```
template<class T, int bsz> class huge {
        T in_core_buf[bsz];
        // ...
        T& operator[](int i);
        seek(long);
        // ...
```

A linked list class:

```
template<class T> class List { ... };
```

A queue tail template for sending messages of various types:

```
template<class T> class mtail : public qtail {
        // ...
        void send(T arg)
        {
                // bundle ''arg'' into a new message buffer
                // and put than on the queue
        }
};
```

A counted pointer template (for user-defined automatic memory management):

```
template<class T> class CP {
        // ...
public:
        CP();
        CP(T);
        CP(CP<T>&);
        // ...
};
```

15 Conclusions

A general form of parameterized types can be cleanly integrated into C++. It will be easy to use and easy to document. The implementation can be efficient in both run-time and space. It can be implemented portably and efficiently (in terms of compiler and link time) provided some responsibility for generating the complete set of definitions of function templates is placed on the programmer. This implementation can be refined, but probably not without loss of either portability or efficiency. The required compiler modifications are manageable. In particular, cfront can be modified to cope with templates. Compatibility with C is maintained.

16 Caveat

The key thing to get right for a C++ template facility is assuring that basic parameterized classes are implemented in an easy to use and efficient way for the relatively simple key examples. The compilation system *must* be efficient and portable at least for these examples. The most reasonable approach to building a template system for C++ would be to achieve this first, make the inevitable changes in concepts based on that experience, and proceed with more advanced features *only* as far as they makes sense *then*.

17 Acknowledgements

Andy Koenig, Jon Shopiro, and Alex Stepanov wrote many template-style macros to help determine what language features was needed to support this style of programming. Jim Coplien, Margaret Ellis, Brian Kernighan, and Doug McIlroy supplied many valuable suggestions and questions.

Building Well-Behaved Type Relationships in C++

R. B. Murray

AT&T Bell Laboratories
Warren, New Jersey 07060

1. Introduction

The C++ programming language[1] allows the designer of a new user-defined type to define the conversions between that type and another type. When the arguments to a function call, overloaded operator, or initialization don't match a declaration exactly, the compiler can use these conversions to coerce arguments to make them match. If exactly one declaration can be matched using conversions, the compiler supplies the conversions automatically; otherwise it is a compile time error.

These *implicit type conversions* can make it easier to write more concise code; however, they can also create problems. The builder of a type structure is walking a thin line between supplying enough conversions to avoid frequent explicit casts, and supplying so many conversions that casts have to be added to resolve ambiguities. In addition, the type conversion rules of C++ make it possible for the addition of other declarations at a later time to break existing code. As the use of C++ libraries grows, these interactions between different packages are likely to become more common.

This paper will begin by reviewing the existing behavior of implicit type conversions in C++. We will then will suggest "rules of thumb" for avoiding unwanted interactions, both for the type designer, and for the type user.

2. Review: type conversions in C++

In both C and C++, the compiler understands how to make certain type conversions, and will quietly insert these conversions into the generated code when appropriate. For example, C and C++ compilers know how to convert an `int` into a `double`; so if a compiler is presented with the code fragment

```
double d;
d = 2;
```

it will quietly convert the `int` 2 into a `double` before doing the assignment to `d`.

In C++, the designer of a user-defined type can specify conversions between that type and any other type. This can either be a specification of how to the convert this type into some other type, or how to convert some other type into this type. Either kind of conversion tells the compiler how to make one type into another; the difference is in which of the two types involved knows how to do the conversion.

2.1 Constructors

Conversion *from* another type `From_type` *to* a new type `To_type` is specified by supplying a *constructor* that takes exactly one argument, either of type `From_type` or `From_type&`:

```
class From_type {
// ...
};

class To_type {
// ...
public:
  To_type();
  To_type(From_type&);
};


To_type::To_type(From_type& o)
{
  //Do what it takes to make a From_type into this To_type
}

main(){
  From_type other_thing;
  To_type this_thing;
  this_thing = other_thing; //To_type(From_type&) called
}
```

2.2 Conversion operators

Conversion *to* another type `To_type` *from* a type `From_type` is specified by supplying a member function (called a *conversion operator*) of the form `operator To_type`:

```
class To_type {
// ...
};

class From_type {
// ...
public:
  operator To_type();
};

From_type::operator To_type()
{
  //Do what it takes to make this From_type into a To_type
  //This function returns the new To_type
}

main(){
  From_type other_thing;
  To_type this_thing;
  this_thing = other_thing; //From_type::operator To_type() called
};
```

2.3 Function matching

The C++ compiler will attempt to use implicit type conversions when the arguments supplied to a function, overloaded operator, or initialization do not match any existing declaration exactly. (For the remainder of this paper, the term "function" will be used to include overloaded operators and initializations). The compiler may supply implicit type conversions in order to coerce one or more of the arguments to the types expected by the function.

For instance, if the function `sqrt` expects an argument of type `double`, but the call passes an `int`, the generated code will include a conversion of the `int` into a `double` and pass the result to `sqrt`. This is also true in Draft Proposed ANSI C[2]; however, C++ extends this behavior to include user defined types.

The C++ compiler will only call implicit type conversions if:

- no declaration for the function matches the argument list exactly, *and*
- there is exactly one such declaration for the function such that each argument either:
  - — matches the corresponding argument in the function declaration exactly; *or*
  - — has exactly one direct conversion that will change the argument into the type specified by the function declaration.

If there are no function declarations that can be made to match by adding conversions, or there are two or more, it is a compile time error.

For example:

```
class Orange {
// ...
};

class Apple {
// ...
public:
  Apple();
  Apple(Orange&); //convert Orange to Apple
};

overload cross;
void cross(Orange,Apple);
void cross(Apple,Apple);

main(){
  Orange navel;
  Apple mcintosh;
  cross(navel,mcintosh);     //calls cross(Orange,Apple);
  cross(mcintosh,mcintosh); //calls cross(Apple,Apple);
  cross(mcintosh,navel);     //converts navel and calls cross(Apple, Apple);
  cross(navel,navel);        //error: two possible conversions
};
```

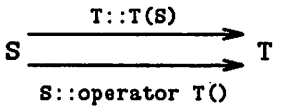
The first two calls to `cross` match a declaration exactly, so no type conversions are called. Since the third call does not match any declaration exactly, and there is no way to make an `Apple` into an `Orange`, the third call can only be resolved by converting `navel` to an `Apple` and calling `cross(Apple,Apple)`. The fourth call also does not match any declaration exactly. The compiler could convert only the second argument (which would match `cross(Orange,Apple)`, or it could convert both arguments (which would match `cross(Apple,Apple)`. Since there is no exact match, and there is not exactly one declaration that can be made to match by supplying conversions, it is a compile time error.

3. Type conversion pitfalls

Implicit conversions can be convenient. However, if the class designer doesn't put some thought into their structure, they can cause troubles (in the form of compile-time errors) later. This section looks at some of the most common problems and describes ways to avoid them.

3.1 Multiple owner problem

This problem occurs when a conversion from S to T is necessary, and both `T::T(S)` (or `T::T(S&)`) and `S::operator T` exist. We call this the *multiple owner* problem because there are two "owners" for the conversion S → T:



(In this picture, each arrow from type S to type T indicates that an implicit type conversion from S to T is declared.) It is never correct for both of these routines to exist.

This problem is minor because it cannot be introduced after the fact, since it requires that each class involved knows about the other. Normally this is only true when the same set of people is maintaining both classes, and it is therefore easily fixed by removing one of the conversions.

3.2 Ambiguous Type Structure

This error occurs when there is more than one possible set of conversions that will match the function being called:

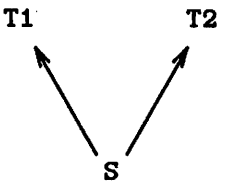
```
class T1;
class T2;

overload func;
void func(T1);
void func(T2);

class S {
// ...
public:
  operator T1();
  operator T2();
};

main(){
  S s;
  func(s);  //Error: two possible conversions
            // (S->T1 or S->T2)
};
```

In this case, there is no `func(S)`, and the type structure of the application allows either of two conversions to resolve the function call:



Avoiding the structural problem is harder, because it can be introduced after the fact. The later addition of a new type can cause existing code that depends on an implicit type conversion to no longer compile:

```
class Apple {
// ...
public:
  Apple(int);
};

void peel(Apple);

main(){
  peel(2);  //calls peel(Apple(2));
};
```

The above code works, but if we later add another class that also defines `peel` we get a problem:

```
class Orange {
// ...
public:
  Orange(int);
};

void peel(Orange);
```

When the new declarations (which are probably buried in a new header file) are added, all the calls to `peel` that depend on the implicit conversion `int` → `Apple` will no longer compile.

This problem will become more common in the future. As software development moves toward more aggressive reuse of code (as economics dictates that it must)[3], the percentage of code in an application that consists of libraries written by someone other than the application developer will grow. This both increases the chances of an accidental name collision and reduces the power of the victim to do anything about it (particularly if the victim does not have access to the library source). A lot of trouble can be avoided if some thought about type structures goes into the library design.

4. Rules for designing type structures

The structure problem arises when there are two or more conversions *from* the same type. If we define the number of types that a type T can be implicitly converted *to* as the *fanout* of T, the number of opportunities for collision from structural problems is $O(fanout(T)^2)$. This is because a function name collision between any two types in the fanout is a possible structure error. So our first rule of thumb is:

Minimize the fanout in the type structure.

By avoiding multiple implicit conversions from a given type, the chances that ambiguities will be created are minimized. This does not mean that it should be impossible for users to convert a given type to more than one other type; the point is that no more than one of these conversions should be *implicit*. Other conversions should be normal member functions. For example, this type structure has high fanout:

```
class Thing {
// ...
public:
  operator Another_thing();
  operator Still_another_thing();
  operator Yet_another_thing();
};
```

Rather than have implicit conversions to three other types, at most one of the types should be chosen for implicit conversions. In this case, suppose `Another_thing` was the most common of the types involved; we should only supply a conversion operator for `Another_thing`:

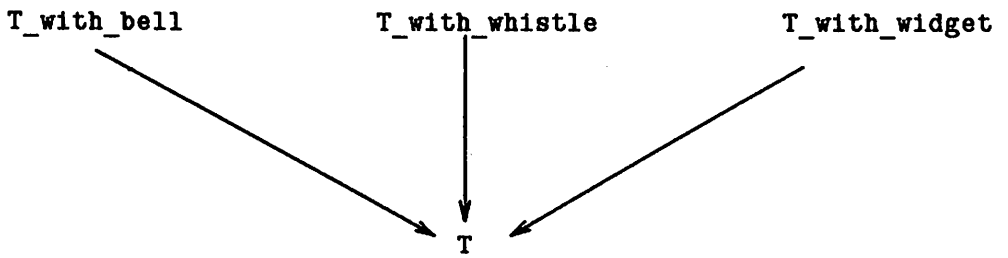
```
class Thing {
// ...
public:
  operator Another_thing();
  Still_another_thing  cvt_Still_another_thing();
  Yet_another_thing  cvt_Yet_another_thing();
};
```

Conversions to `Still_another_thing` and `Yet_another_thing` will now require an explicit call to the appropriate member function.

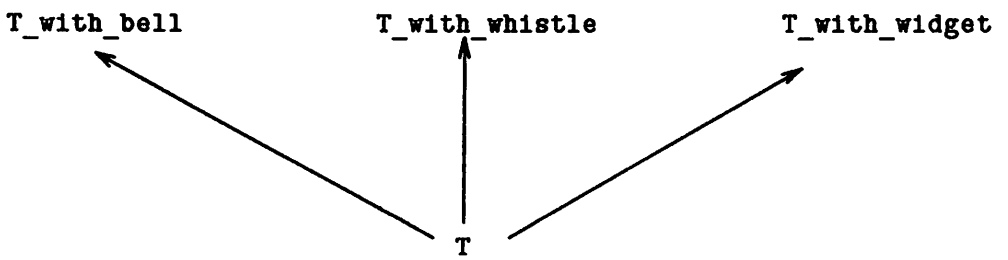
4.1 Simplifying conversions

Implicit type conversions are especially useful when one of the classes is an extension of the other. This may be an extension of the domain (e.g. `Complex` is an extension of `double`), or of the concepts (`String` is an extension of `char*`). In these cases, the implicit conversion between the two classes should be from the extension to the simpler class; we call this a *no-value-added* conversion.

Why are no-value-added conversions better? In general, there will be more than one extension for a given simpler class. Implicit conversions from the extensions to the simpler class will fan *in* to the simpler class, which doesn't cause ambiguities:



On the other hand, implicit conversions from the simpler class to the extensions (*value-added*) will fan *out*; there can be an ambiguity as to which extension the simpler type should be converted to:



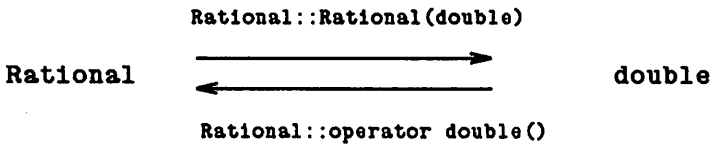
Since the simpler type should not know about the extension, no-value-added conversions will be conversion operators (as opposed to constructors). So the rule of thumb is:

Conversion operators should provide unique conversions to simpler types (no "value added").

4.2 Mutual conversions are OK

It may not always be obvious when one type is an extension of another. For example, consider the relationship between a type `Rational`, which implements rational numbers as the quotient of a pair of arbitrary precision integers, and the type `double`. Conceptually, the set of real numbers represented by `double` includes all rationals. However, since the implementation of most `doubles` is a fixed sized mantissa and a fixed size exponent, the set of data representable by `Rational` may in fact include all the data representable by `double`!

For situations like this, it is often simplest to use mutual conversions; each type can be implicitly converted to the other:



Surprisingly enough, this structure is not necessarily bad. If a function `f` accepts either a `Rational` or a `double`, any use of `Rational` or `double` will match exactly; if the function only accepts one of these types, the conversion will be called for the other type.

4.3 If in doubt, leave it out

Every automatic type conversion opens up a new opportunity for error, either from multiple conversions requiring an explicit cast to be added, or (more sinister) from an unintended conversion causing the wrong function to be called when the right function declaration was missing. The most important rule of thumb is therefore:

If an implicit conversion is not obviously necessary, leave it out.

5. Implicit conversions in user code

The user of a library needs to be aware that constructors often break the rule that implicit conversions should be unique, no-value-added conversions. Often, at least one form of the constructor for an extension will take, as its sole argument, the simpler type:

```
class T {
// ...
};

class T_with_bell {
// ...
public:
  T_with_bell(T);
};
```

The constructor `T_with_bell::T_with_bell(T)` defines an implicit value-added conversion. The designers of `T_with_bell` may not have intended their users to depend on this implicit conversion; it may simply exist because there is no way to specify a constructor of this form without also declaring an implicit conversion.

This is particularly common with constructors that take built in types. For example, a class that provides a buffer pool might have an integer parameter to the constructor that specifies an initial size of the pool:

```
class Buffer_pool {
// ...
public:
  Buffer_pool(int);
};
```

The fact that this defines an implicit conversion from `int` to `Buffer_pool` is just an accident; users' code should not depend on it. For example, if there is a function

```
void flush(Buffer_pool);
```

which fills a `Buffer_pool` with available things, users should not call

```
flush(5);
```

with the intention of throwing away the next five things. This code depends on the conversion `int` → `Buffer_pool` to construct a temporary `Buffer_pool` of size 5, pass it to `flush`, and destroy it after the call returns. A compilation error can be introduced by the later addition of a constructor that takes `int` as its only argument if there is also a name collision on `flush`:

```
class Toilet {
// ...
public:
  Toilet(int);
};

void flush(Toilet);
```

Now, `flush`ing an `int` no longer works. The second example in section 3.2, where code stopped working because an implicit conversion from `int` to `Apple` was broken by the subsequent addition of a conversion from `int` to `Orange`, is another example of this. So the rule of thumb here is:

Avoid the use of value-added conversions, even if they happen to be available. This is especially important for conversions from built in types.

5.1 Repairing a broken type structure

If the user of a type gets bitten by a problem in the type structure, there are two ways to repair the problem in the users' code.

5.1.1 Provide explicit conversions An ambiguity can be resolved by providing an explicit conversion in each function call:

```
class Orange {
// ...
public:
  Orange(int);
};

class Apple {
// ...
public:
  Apple(int);
};

overload peel;
void peel(Orange);
void peel(Apple);

main(){
  peel(Apple(2));
};
```

This has the advantage of making explicit an operation that may not have been obvious beforehand; but it also may clutter up the code, and can be a big effort. If there are few lot of calls involved this is probably the simplest and clearest solution.

5.1.2 Provide disambiguators An alternative is to provide a *disambiguator* for the function involved:

```
inline void
peel(int i)
{
  peel (Apple(i));
}
```

Since the disambiguator is `inline`, there is no additional run time cost. This allows a fix to be made in one place, as opposed to being scattered throughout the code. However, we have added yet another inline function declarator to our headers. If there are many calls involved this is probably the easiest solution.

6. An Example

As an example, we'll consider a type structure for various kinds of numbers. The types involved will be:

— `int`;

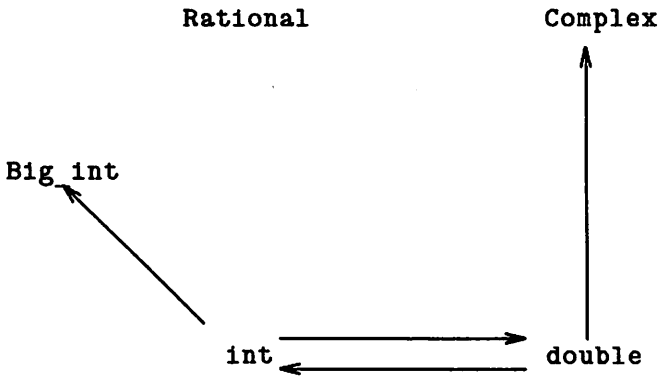
— `Big_int`, supporting integers of arbitrary length;

— `Rational`, rational numbers (implemented as a pair of `Big_ints`);

— `double`;

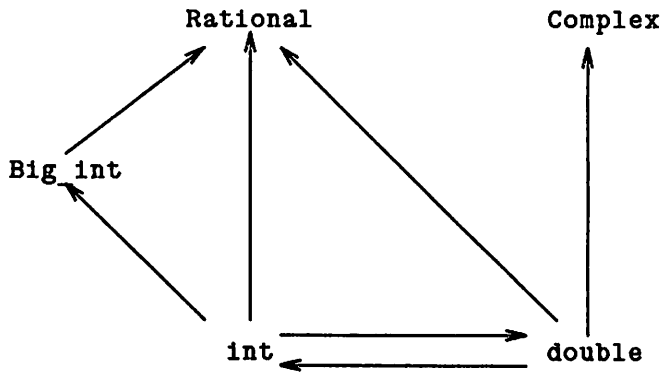
— `Complex`, a complex number implemented as a pair of `doubles`.

We'll build this type structure in two steps. The first step is to figure out what conversions must exist, either because they are implied by constructors, or because they already exist:

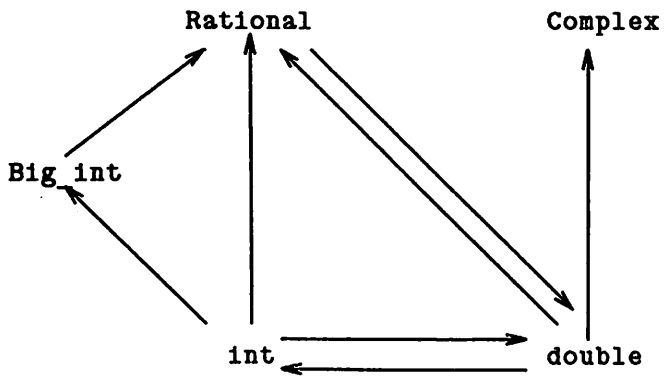


`int` and `double` are mutually convertible by the rules of C. We clearly need to initialize `Big_int` from an int, and it makes sense to initialize `Complex` from a `double`.

How do we initialize `Rational`? This is a little harder. The fact that `Rational` is implemented using `Big_int` is really an implementation detail; you should not require the user of `Rational` to know about `Big_int` in order to use the package. However, we can imagine that a `Rational` could be initialized by `int`s, `double`s, or `Big_int`s, and we supply constructors for this:



Having understood the conversions that exist because of constructors, the second step is to decide what simplifying (no-value-added) implicit conversions (using conversion operators) we should supply. We avoid implicit conversions when there are points in the domain that do not obviously map to the range; e.g. it's not obvious how to convert a `Complex` to a `double` when there is a nonzero imaginary part. However, there is a clear way to convert a `Rational` to a `double` (although we should be aware of possible loss of precision or range errors). So we add `Rational::operator double` to our type structure:



We don't add any other downward conversions, since that would increase the fanout from `Rational`. The high fanout from `int` and `double` is unavoidable (and is characteristic of built in types).

Suppose we want to convert a `Rational` to an `int` or a `Big_int`? Adding implicit conversions from `Rational` would increase the fanout; but we can imagine times when these conversions would be useful. The answer is to provide conversions that are normal member functions, without defining an implicit conversion:

```
class Rational {
// ...
public:
  Big_int cvt_Big_int();
  int cvt_int();
};
```

these allow users of `Rationals` to get the `Big_int` and `int` equivalents by making an explicit call.

We've avoided conversions between `Complex` and `int` or `Big_int` because we don't imagine this conversion happening very often (if it does, perhaps a version of Complex that uses `int` or `Big_ints` would be more appropriate), and can easily be done by converting to or from a `double` first. Since the conversion is not obviously necessary, we leave it out.

7. Summary

The moral of the story is: use implicit type conversions carefully, and with restraint. Every implicit type conversion has the potential for causing problems down the line. Think about the type structure as a whole, and look for ways to avoid fanout. The use of implicit type conversions from a simpler to a more complex type, even if the conversions are available, should be avoided; this is especially true if the simpler type is a built in such as `int`.

As software economics drives us toward more ambitious reuse of code, larger and larger parts of an application will consist of libraries written by someone other than the developer of the application. It is the responsibility of the library developer to make sure that the type structure provided by the library is as simple and small as is practical.

8. Acknowledgments

Helpful comments and suggestions were provided by Martin Carroll, Andrew Koenig, Stan Lippman, Barbara Moo, Jonathan Shopiro, and Bjarne Stroustrup.

REFERENCES

1. Stroustrup, B. *The C++ Programming Language*, Addison-Wesley, 1986.

2. *Draft Proposed American National Standard for Information Systems- Programming Language C*, 5/13/88 draft, Doc. X3J11/88-090, J. Brodie, chair.

3. Levy, L. *Taming the Tiger: Software Engineering and Software Economics*, Springer-Verlag 1987.

Porting from Common Lisp with Flavors to C++

Joseph Eccles

AT&T Bell Laboratories (Cap Gemini America)

ABSTRACT

While Lisp workstations provide a wonderfully productive development environment, they are expensive, often forcing the choice of another target machine for deployment. This paper describes some of the concerns and problems faced when porting a large (20000 line) system from the Texas Instrument Explorer workstation to a SUN/3. In each case an object oriented language was used, Common Lisp with Flavors on the TI and C++ on the SUN.

The emphasis here is on the differences between the two language systems. This paper addresses the translation of Flavors into C++ classes, the semantics of Lisp lists, and the mimicing of Lisp dynamic binding in C++. It deals only briefly with some other differences, such as multitasking and window systems.

1. Introduction

During the late 1970's and early 1980's researchers at the MIT Laboratory for Computer Science developed a Lisp based workstation that was later commercially developed by several companies. The highly integrated and tool rich environment provided by these workstations makes them attractive alternatives as development environments, especially for prototyping. These systems include capabilities for incremental compilation or interpretation of code, the intermixing of compiled and interpreted code, powerful tracing, debugging, and data inspection facilities, and flexible window systems. With all their advantages, though, the expense of these machines makes them less than ideal delivery vehicles. In addition, since several companies are now independently continuing development of the workstations, the programming environments are diverging, especially in the area of extensions to the windowing system. These divergences will tend to lock an application into a single manufacturer's hardware.

On the other hand, UNIX has become a standard software platform for graphics workstations, and there are several such systems with computing power and bit-mapped graphics capability comparable to the Lisp machines. With the addition of a portable graphics/windowing environment such as the X Window System and the object oriented capabilities of C++ these are becoming cheaper and more standardized alternatives.

For these reasons, we have undertaken to convert a large existing application from the Common Lisp/Flavors language to C++ running on a UNIX workstation. The application in question is a graphics based user interface for a network monitoring and control system, consisting of about 20,000 lines of code running on a Texas Instruments Explorer II color workstation.

The remainder of this paper will focus on the differences in the two language environments, and the difficulties encountered as a result of these differences.

2. C++ Replacements for Common Lisp Types

The standard data types of Common Lisp are easily replaced with C++ classes or basic data types. Simple types such as integers and characters are as easily handled by either language. Other important Lisp data types require class extensions to C++, either from existing class libraries, or specifically constructed to mimic the Lisp programming environment. Examples of such data types are

- strings
- hash-tables
- lists.

2.1 Strings

Common Lisp has a large set of routines available for handling string and other array like data, and user interface applications are likely use these routines heavily. C++ can easily provide such functionality by creating a sub-class of an existing string class[1] with the extra member functions added. Such an approach is preferable to adding more member functions into an existing class definition, since the changes are then cleanly layered on top of the public interface of a base class.

2.2 Hash-tables

Within Lisp applications hash tables are commonly used to map properties to key values. They are important in Common Lisp particularly when the key values are not integers, or when the key values are sparse. An example of the first case is the use of string keys, where a hash table will give a great performance advantage over exhaustive string comparisons. We are currently using a class of hash tables with string keys[2] to handle such mappings.

2.3 Lists

While there is existing C++ support for linked list classes[1], these are not consistent with the Lisp concept of a list. The biggest difference is that list classes mentioned above are implemented as doubly linked lists. This makes both nested lists and shared sub-lists, both of which are commonly used in Lisp programming, impossible. As a result, we have implemented our own singly-linked list class[3].

The singly-linked class that we created was a list of pointers, which was necessary because the objects in the lists could potentially belong to more than one list. Another consequence of this is that destroying a list or removing an element should not destroy the object being pointed to; it is left to the programmer to ensure that an object is destroyed before all references to it vanish.

Incorporated into the definition of the singly-linked list class are equivalents of many of the Lisp list functions, such as *car*, *cdr*, and so on. As in Lisp, making a copy of a list creates a new list pointing to the original contents: only the top level is normally duplicated.

3. An overview of Common Lisp/Flavors

Common Lisp/Flavors is an object oriented programming environment built on top of Common Lisp. In addition to the standard Lisp types, such as integers, characters, arrays, and lists, the Flavors extension allows for user defined abstract data types. Each abstract data type, or *flavor* has associated with it zero or more private data members and a public interface consisting of one or more associated functions, or *methods*. A flavor may be built on top of one or more *base flavors*, and will inherit all data members from these flavors.

The interaction of methods in the derived and base classes can be controlled by the programmer to a large extent. In the default, and most commonly used scheme, each method is divided into three components, the *before* method, the *primary* method, and the *after* method. For a method defined directly for a single class, the before, primary, and after methods are executed in order. For a derived class that overrides the base class methods, however, things are more complex, and the flow of execution proceeds as follows.

- The *before* method of the derived flavor is called, followed by that of the parent flavor if one is defined, and so on, until all before methods in the hierarchy have been called.
- Next the primary method for the derived flavor is executed. If none has been explicitly defined for this flavor, it will be inherited from the parent flavor.
- Finally, the after methods are executed in the reverse order of the before methods.

For flavors derived from several base flavors (multiple inheritance) the sequence is more complex, but again the primary method will be inherited from the first base flavor for which it is defined, unless it redefined for the derived flavor. The Flavors system also provides other schemes for invoking methods, giving the programmer control over how inheritance functions. For example, the *or* combination calls the method for the derived flavor if it exists, then calls the method for its base flavor only if the first call has returned false, and so on until some method finally returns true or it runs out of base classes. In addition to the standard combinations, a mechanism is also provided to allow the user to specify arbitrary combinations of inherited methods. While these combinations add to the power of the language, we found that they were not necessary for our application.

4. Differences between Flavors and C++

This section is not intended to be a complete discussion of C++, but rather a comparison of C++ with the features of Flavors discussed above. There are several differences in the semantics of Flavors methods and C++ member functions, but these have generally not been a problem.

- All methods act like C++ virtual member functions, that is the binding of the function is done dynamically at run time.
- Flavors allows methods to be broken into before, primary, and after components, while C++ has no such concept.

- When multiple base flavors would create an ambiguity in the definition of an inherited (primary) method, Flavors resolves the conflict by choosing the method from the base class that was lexically first in the flavor definition. In C++ such an ambiguity generates a syntax error at the spot where the reference was used, and all such usages must be explicitly qualified by the programmer using the :: scope operator.
- While C++ handles the inheritance of member functions by allowing a class to inherit an existing function from its parent class or to redefine it, Flavors has a wide variety of ways in which a class's methods can be combined with those from its base flavors.

In creating the Lisp version of the user interface, we found that full power of methods combination in flavors was largely unused. With rare exceptions only the default daemon combination was used, and the exceptions were invariably tied to the TI window system and mouse interface, which had to be rewritten in any case. Few of the methods used had before or after methods, and of these, most fell into three groups that are easily dealt with.

- :after :init — This is a special method that is called when a flavor is instantiated, after storage for the object has been allocated and after variables have been initialized. This is almost identical with a C++ constructor, except that a constructor is also responsible for explicitly initializing member variables.
- :before :delete — It is common to define cleanup routines that deal explicitly with a particular flavor within an objects inheritance hierarchy. This is like a C++ destructor, which is invoked before the space for the object is reclaimed.
- :before :set-variable / :after :set-variable — Methods are often placed around such variable setting routines to validate the operation or its results, or to do additional processing implied by the operation, for instance, after setting a background color, we might want to force a window to redraw itself. The reason such constructs are common in Flavors, however, it that the set methods are often automatically generated, and do nothing other than change the value of a single variable. In C++ such functions are always explicitly coded, and so it makes more sense to write the entire before/set/after operation as a single routine.

In these cases the processing done in the before or after method can be conveniently incorporated into the body of a C++ member function — into the constructor, the destructor or another routine.

5. Lisp Control Structures

Some of the control structures provided by Common Lisp proved to be particularly difficult to deal with, because of the basic differences with the C++ language. Of particular concern were several constructs that were heavily used in the Lisp implementation of our system, including

- error and condition handling
- UNWIND-PROTECT clauses

- dynamic variable scoping and binding (the *LET* clause).

5.1 Error and Condition Handling

The Lisp environment on the MIT derived workstations provides a powerful error handling facility based on Lisp's non-local branching, or *THROW* capability, which is similar to the longjmp() routine in C or C++. Within this scheme an error type is a flavor derived from the *error* or *ferror* (fatal error) flavors, which in turn are derived from the *condition* flavor. The programmer may declare a handler for a condition or set of conditions, which is in effect until the flow of control exits the stack frame in which it is declared. When an error is detected during the execution of a Lisp expression, an object of the appropriate flavor is instantiated, and the stack is unwound until an appropriate handler is found. The handler may do some local processing, and then resume execution just after the point where it was established, or use a non-local branch to unwind the stack further.

The main problem with trying to replicate this scheme in a C++ environment is that C++ does not interact well with the setjmp/longjmp mechanism. While the use of the stack for automatic variable storage ensures that the memory for local objects will be reclaimed by a longjmp, it does not allow destructors to be called, nor does it help for objects dynamically created with the *new* operator. In a recent article W. M. Miller has discussed this problem[4], and has proposed a partial solution.

The alternative to such a scheme is to propagate error conditions back from function calls to a level where they can be intelligently handled. To do this one must always be able to recognize an error value returned from any function. This can be done, but it is cumbersome and is a potential source of errors, and requires a reorganization of the code from that of the original Lisp program.

There are several problems with the approach described by Miller.

- Since destructors can not be directly called by the programmer, all destructors must simply call the virtual function *cleanup* that does the real work. This means that it is impossible to handle existing class libraries without modifying them. For example, the Strings class could not be used without changing its source.
- The constructor must be able to distinguish between automatic and dynamic objects, so that the destructor will know when the space is not on the stack, and so must be freed.
- The cleanup mechanism should probably not delete objects that were created dynamically. This can be left to the programmer, with the aid of the *protect* facility described in the next section.

To avoid these problems a class *Clean* was created as an alternative to Miller's cleanup_obj. It maintains a stack of objects to be cleaned up, but rather than acting as a base class for other objects, each Clean object keeps a pointer of type Base, which has a virtual destructor. The only function of Base it to allow delete to function on the pointer in the Clean object. The constructor for Clean accepts one argument of type Base*. Thus when an objects are discarded from the stack, the

objects pointed to are deleted, and the proper destructors are called. The destructor for Clean will also delete the object.

Objects of class Clean should always be automatic, since they do not know how to release their own storage. The objects to be cleaned are always created dynamically, but should not be deleted explicitly. They will be deleted when the clean object goes out of scope. It is up to the programmer to decide which objects should be cleaned up. Hopefully in the future error handling will be more completely integrated in with the C++ language.

5.2 UNWIND-PROTECT

The *UNWIND-PROTECT* mechanism is widely used in Common Lisp to handle cleanup of the running environment, especially in handling error conditions. When execution flow exits the scope of the UNWIND-PROTECT, a list of user supplied expressions are guaranteed to be executed, whether the exit is normal or by way of a non-local branch, including when a condition handler is invoked.

Protect is a simple class that provides a part of the UNWIND-PROTECT functionality. Its only public interface is through a constructor and a destructor, and thus the only legal operation is initialization. The class definition is

```
class protect {
        void (*function)(void *[]);
        void **params;
public:
        protect(void (*f)(void *[]), void *p[])
                {function = f; params = p; }
         protect() { (*function)(params); }
};
```

The destructor for this class calls a user supplied routine with a user supplied argument when the scope of the variable is exited. A simplified example of using class protect follows.

```
#include <iostream.h>
#include "protect.h"

char *p[] = {
        "The second parameter is the integer %d .
        (char *)5
};

// This program will declare a protect handler within a block,
// and then exit the block. This will cause the handler to
// execute.
main()
{
        {
                protect(doprotect, p);
                cout << "Ready to exit inner block ;
        }
        cout << "Done ;
}

void
protect(void *args[])
{
        char buf[80];

        sprintf(buf, (char *)args[0], args[1]);
        cout << buf;
}
```

This is of course overly simplified since in the absence of non-local jumps such cleanup could be handled much more directly. Obviously this mechanism is only useful with the condition handling mechanism described above or a similar one in place, since if the normal longjmp is used the destructor will never be called. The example above shows how the class would generally be used, that is, for anonymous declarations. This is possible since the object created will never be referenced by the programmer, and since keeping a temporary variable around to reference it would simply clutter the program and make it less readable. There would also never be a reason to allocate such objects dynamically using the new and delete operators, since the power of the construct lies in the automatic call of the destructor.

There are some deficiencies with this scheme when compared with the Lisp UNWIND-PROTECT mechanism. While UNWIND-PROTECT can execute an arbitrary collection of Lisp expressions with access to all the variables within the scope of its definition, the protect class executes a function with access to only those variables in its argument array or those that are global.

5.3 Dynamic Binding — LET

The most profound way in which Lisp differs from C or C++ is in its handling of variable binding. In Lisp a symbol is treated as a run-time object that can be manipulated, while for most other compiled language symbols are replaced in the execution environment with constant addresses. Thus for C++ it is not possible to change the *binding* of a symbol, that is, to cause a symbol to reference a different address. In Lisp code a symbol is in effect accessed by name, and its contents are accessed by evaluating it. The *LET* statement in Lisp allows the programmer to create a new value binding by placing a new symbol with the same name on the run-time stack, masking the original definition. The new binding is in effect until control exits from the level at which the new binding was made, and the stack is unwound.

Dynamic bindings are often used in Lisp for much the same purposes as pointers are used in C or C++, and the use of symbol bindings tends to make pointer (or *LOCF*) references rare in Lisp programs.

Other uses of LET are harder to deal with, though. Lisp supports scoping of variables dynamically as well as lexically. That is, while C or C++ define the scope of a variable to be the block in which it was defined, and all nested blocks within that block lexically (as read), a dynamically scoped, or *special* Lisp variable may be accessed by any code that runs while the symbol is on the stack. In a single-tasking environment this is equivalent to temporarily changing the value of a global variable, although the "global" variable may not be accessible to all parts of the program. In a multitasking system the effect is like having a separate set of global variables for any execution stack that declares one.

Frequently the programmer will use this to avoid passing extra arguments to functions. For example, when a task is associated with a particular window, that window is bound to the symbol **standard-output**, which is the default target for the common output routines. Often such a usage can be eliminated by replacing ordinary function definitions with method or member function definitions for an appropriate class.

It would certainly be possible to create a replacement for Lisp dynamic binding in a C++ environment — for instance using a *symbol* class with the appropriate properties — it is not clear that the gains would be sufficient to justify the added complexity.

6. Other Problems

There are two additional sets of differences that had to be dealt with in the process of accomplishing this port. The first is multitasking, which was extensively used in the Lisp version, and the second is the programming interface to the window system. These two areas are strongly interconnected.

6.1 Window System

For a user interface the window system is a primary concern, since it provides all interaction with the user — both output and input. Unfortunately, window systems also tend to be the least portable part of the environment for any program that uses one. While this is starting to change with the development of standards such as X11, the window system dependent code is still hardest part to move from one target environment to another. X11 is the target window system for this application, in part because of its growing status as an industry standard.

The window system provided by the Lisp machine environment is tightly integrated with the operating system and all standard applications. It is built on top of Flavors, with each type of window defined as a flavor, and options, such as title bars and scrolling, defined as a *mix-in flavor* which can be added to another window type. The set of predefined flavors is rich, and user defined window types are most often just trivial combinations of these.

Compared with this, X11 is poor in features, providing just the basic support for manipulation of simple windows. To use X11 effectively it is necessary to have additional layers of functionality built on top as a toolkit. Unfortunately the Xt toolkit provided with X11 has several problems. It is cumbersome to extend and use, largely because it tries to provide an object oriented interface using C. This results in code that is difficult to read and debug.

6.2 Multitasking

A multiwindow user interface requires something akin to multitasking operation, since there are multiple input sources, each of which can independently require processing. In the Lisp machine environment it is common to provide a separate running process for each window displayed, so that operations in one window won't interfere with the functioning of other windows. However, the Lisp concept of a process is different from the UNIX one. A Lisp process is *lightweight*, that is it shares a global address space with all other processes, and maintains a separate stack for local variables. In the UNIX case, processes have completely separate address spaces, and communication between processes occurs only through well defined facilities, such as shared memory and message queues.

The task library for C++ by J. E. Shopiro[5] provides the capability to create multiple tasks, or execution control threads, within a single UNIX process. Unfortunately, the real time task system did not interact well with X11, and was abandoned.

While multitasking seems like a natural approach to handling multiple sources of input, it is not really necessary for the implementation of a multiwindow user interface. In an event driven system like X11, handling routines can be called to process each event as it is detected. As long as no handler retains control for to long a period of time the effect is like multitasking. It is only necessary then to allow the reading of other input sources, such as IPC messages from other processes, to be intermixed with window events. This approach is now being investigated.

7. Summary

To date the port of the Lisp version of the user interface is currently about 30 percent complete. Those parts dealing with the creation and initialization of the major internal data structure — a directed graph[6] — and those for the parsing and execution of keyboard commands have been recoded in C++. If we ignore the extra C++ support needed to define Lisp lists, and so on, there seems to be little difference in source code size for the two versions. The most troublesome part remaining is the interface to X11.

Prototyping database applications with a hybrid of C++ and 4GL.

Ronan Stokes

Glockenspiel Ltd.
19, Belvedere Place
Dublin 1, Ireland

Phone 353-1-364515
Fax 353-1-365238

ronan@puschi.uucp
..!mcvax!iclitc!puschi!ronan

Abstract

There are many 4GLs on the market which allow interfacing with C in order to provide added functionality. This paper describes the author's experiences in developing a prototype for a shift management and time recording system (STMS), which has been completed and delivered, using the combination of C++ and a 4GL (in this case Seachange). The paper describes the use of classes and inheritance to abstract the functionality provided by the 4GL. The first part of this paper illustrates how the hybrid of C++ and a 4GL allows development of the final product by a system of incremental replacement. Incremental replacement is a systematic method of re-implementing the abstract classes which interface with the 4GL features, in C++ in order to remove dependencies on the 4GL. This allows the development of a final C++ product as a further development of the prototype rather than a lengthy rewrite. The next part of the paper describes the use of the concepts of services, sets, inheritance, interaction objects and interface classes to produce an abstraction of the 4GL. This section is illustrated with examples from the shift management system. The final part of the paper briefly discusses the performance constraints giving some metrics of using C++ with a 4GL.

1. INTRODUCTION

This paper is based on the prototype development of a rota management system STMS. STMS is a shift and time management system for use in a hospital working environment. The system is used for keeping employee information, planning and scheduling work rotas and also for daily updates of information such as absences, changes to shifts, overtime and pay calculation by way of data transfer to other systems. The target hospital environments are characterized by large numbers of employees with differing jobs, working schedules and in many cases, employees work in more than one department during a single day. In addition to this several working weeks were in operation for the hospitals concerned - Five and a half day week, six day week and five day week.

Several constraints were to be taken into account when choosing the means of development for the prototype.

- The prototype was to be completed and delivered in two months. This meant that development time was of prime importance.

- Also due to time constraints it would not be possible to produce a 'throw-away' prototype - a prototype with the look and feel of the end product but which would be re-coded for speed and space optimization.

- Secondly it should be possible to extend the system to work with various windows packages and other user oriented interfaces. It should also be possible to reimplement sections and add extra functionality without requiring changes to existing functionality.

- Finally certain implementation restrictions were applied. It should be portable among the range of machines used in the hospitals - mainly pc's running Dos or Unix.

Due to these restrictions and others to be described later a hybrid of C++ and a Fourth Generation Language (or 4GL), in this case Seachange from Thomson Computers of England, were chosen as the means of development for this prototype.

In the next section I will discuss the merits and demerits of using 4GLs and C++.

2. PROTOTYPING WITH 4GLS.

There are no standard definitions for the various generations of languages, however here is a simple explanation.

Martin [1] defines a Fourth Generation Language or 4GL as a non-procedural end-user oriented language. Programming is achieved by specifying the solution directly rather than as a set of functions or procedures giving an algorithm to find the solution. (For a good overview and comparison of 4GL's see ref. [2]). The knowledge required to find a solution is built into the 4GL itself and the effectiveness of this solution relies on the scope and efficiency of the 4GL's set of predefined functions, which tend to be directed toward a narrow range of problems. This form of programming was intended to give the power of a conventional language to non technical computer end-users allowing applications to be produced in fractions of the development time required by conventional languages.

4GL systems currently in commercial use tend to be a set of tools comprising a database management system, screen and report generators and some limited form of procedural language allowing procedures to be composed from the 4GL's set of functions.

4GL's usually allow some form of interactive task specification in addition to programming through the language supplied. Common examples of 4GL systems include dBase, Focus, Oracle, Ramis II and Seachange.

The main advantages of using a 4GL are speed of development time and small learning curves. Interactive generators allow the building of the overall structure of the application and its associated data representation in a matter of hours. The development time advantage however, quickly deteriorates as the programmer desires to achieve some effect outside of the range of the 4GL.

Other inherent disadvantages include :-

- Royalties and runtime license costs. Unlike conventional languages it is customary to for developers intending to market the application to have to pay a run time license fee to the 4GL distributor for each copy sold.

- Runtime Limitations. Applications produced by 4GL's are generally a lot slower than those produced from conventional languages.[2] Other runtime limitations are enforced by the 4GL system itself - the use of certain keys in the application, the enforcement of certain screen styles and in some cases (dBase & Seachange) only records from one file may be displayed on the screen at time.

- Language constraints. The 4GL systems supplied, due to their market, tend to be intuitive rather than concise, rigorously defined languages. Procedural components are often lacking in functionality or resort to old structure forms (such as single line conditional branches etc.) As the language is often proprietary to the company developing it, 4GL's don't have the advantages of conventional languages with their larger user bases, independent research and conferences, and the competitive multi-company language development. This reason also contributes to the performance failings of 4GL's. With a lower user base, there is less incentive to provide complex optimisations in the application generators provided with these packages.

- A major problem for developers is that of disposable prototypes. For commercial or performance reasons, software developers may wish to develop their final product in a general purpose programming language such as C or C++. There are two common approaches to this task. One alternative is to convert the 4GL application to the target language using one of the 4GL converter products available such as the 4GL to C converters. However the source code produced by these is not easily maintained.
  The other approach is to rewrite the system from scratch with the prototype being used solely as a guide to the appearance of the final product. This approach is wasteful of the time spent developing the original prototype.

In order to resolve some of these problems, many 4GL's allow interfacing with a conventional language such as C. Seachange is one such 4GL and this was a major consideration for its choice.

3. PROTOTYPING WITH C++

Object-oriented programming in general, and C++ in particular has many features which hasten the development time of large systems. Data hiding, data abstraction and inheritance allow a modular implementation and a close mapping of design to implementation. Using the data abstraction features allows rewriting and extension of features without a rippling of the changes required throughout the system. C++ in particular has the advantage of combining the philosophy of object-oriented programming with speed and space efficient executables. The advantages of using C++ are obvious to proponents of Object-Oriented programming.

However development time for C++ systems, although fast, is of several orders of magnitude greater than that using a 4GL.

4. PROTOTYPING WITH A HYBRID OF C++ AND 4GL.

Using a hybrid of C++ and a 4GL provides a compromise which removes many of the problems associated with prototyping solely by 4GL's or C++. By combining the speed of development of a 4GL and the object-oriented features of C++, a prototyping system is obtained which produces easily maintained and extended prototypes. The modularity and maintainability of C++ allows the production of an easily maintained and extended system. Production of a prototype can be tailored to available time and functional requirements. Time critical or frequently used portions of the application may be coded in C++ for efficiency, while large portions of the application may be developed speedily through the use of screen generators etc.

One may argue that interfacing a language such as C to a 4GL would produce the same effect. However the concepts of object-oriented programming allow the production of a system which is independent of the quirks of the particular 4GL.

By using data abstraction and inheritance, it is possible to design a set of interface classes which abstract the dependencies on the 4GL. Each service required from the 4GL is interfaced to C++ by a clearly defined class (or class hierarchy) representing the service. All dependencies on the 4GL are hidden in these interface classes, and subsequent changes require only the modification of these classes. All further usage of a 4GL feature is provided by the associated interface classes. Specialized handling of particular files and other application objects can be provided by inheritance from these classes.

This system of interfacing allows the extension or re-implementation of the application by incremental replacement. Any particular service can be re-implemented or extended in isolation without affecting the overall product. Services can be replaced on a service by service basis eventually , if so desired, leading to a production version system implemented totally in C++. In particular each change may be made while maintaining an up to date working product. Other possibilities are adding on different types of user interface (such as windows packages), natural language interfaces or replacing file systems or structure.

This system of incremental replacement ensures that prototypes can be developed as a working starting point for the final product rather than as an initial disposable imitation.

5. STMS DEVELOPMENT

STMS was implemented using the hybrid scheme described above. In order to explain how Seachange interfaces to C++ it is necessary to understand the organization of this 4GL.

5.1 Seachange organisation.

Seachange, following the typical organization for 4GL's, provides a set of tools in an integrated environment. These tools consist of a database generator, screen generator, menu generator and report generator with a Quick C style environment surrounding this. Each of the components for an application may be specified through interactive sessions with Seachange which in turn generates 4GL language files and compiles them, or by writing the 4GL language files directly. These files include ways of specifying fields in

records, options available in forms, actions to be taken on certain menu options being chosen, and screen layout. In addition to these components 'trigger' files may be specified - files of actions to be taken upon certain events happening. An example follows overleaf.

```
when removing with employee.s

remove from absences.f index absence_key =
  absence_key: employee_number

remove from hours.f index hours_key =
  hours_key: employee_number

adjust
  display "Removing cross references"
...
```

This set of actions causes the records of a particular employee for attendance hours and absences to be removed from their associated file whenever the employee is removed from the database.

At certain points C functions may be called from within Seachange either directly, as in menu options or form options, or indirectly - by way of triggers, upon moving form one field to another or as a set of actions to call when starting or finishing a form or menu. Within the C program the current menu, form, records etc. may be interrogated or modified through a set of C libraries provided with Seachange.

From the C or C++ end functions are called by way sending a message to a message handler associated with the given form or menu. The message, organized as three null terminated strings is dispatched to the associated handler with the strings representing the name of the form or menu, the name of the function to call and a single string as a parameter to the function. From the C++ or C programmers point of view, this is quite similar to message handling in MS-Windows.

From the C end it is also possible to modify a function table which contains pointers to functions for the standard operations - validation of data types in fields, adding a record etc.

5.2 Interface Classes.

The C++ interface to Seachange was organized as a system of several fundamental services, each being represented as an interface class hierarchy. These services represented as a file interface, a database interface, a screen interface (at low level - e.g. fonts, boxes etc.), a form interface and menu and trigger interfaces. The database interface, as distinct from the file service provides means to locate global Seachange variables, to which file a certain key refers and other non-file specific actions and operations. All interface classes use constructors and destructors where necessary. This avoids the need to call any library functions directly for cleanup or initialization. The following example is taken from the menu service.

```
class menu {
    char name[13];
    int is_loaded;
    DB_VALUE (*old_dispatcher)(char*,char*,char*);
    ...

    public:
    menu(char *);   // Create menu from menu file.
    ~menu();
    void run();
    void set_dispatcher(auto DB_VALUE (*)(char*,char*,char*));
    void restore_dispatcher();
    };
...

StartUp()
{
menu main_menu("title");        // Load top level menu
...
main_menu.set_dispatcher(main_processing);
main_menu.run();                        // Run menu
}
```

This simple service provides all menu handling required by applications. The menu is created using the 4GL menu file, the dispatcher is set up and the menu is run. By re- implementing the menu constructor it would be possible to replace the user input with a windowing system or a natural language interface without requiring changes to the rest of the program.

In another example, this time taken from the form service, it is possible to see the role of inheritance in interface classes. Note: types prefixed with the letters "SC" refer to Seachange underlying types.

```
class form {
 protected:

 struct SCF_subform fdef;
 struct SCF_functions *ffns;
 struct SCF_footnote *fft;
 ...
 public:
 form(char *, dispatch_fn_ptr);
 form();                // Current form being run
 ~form();
 void set_rmfn(remove_fn_ptr);
 char *run();     // Run form
 ...
 };
```

```
// Inside the menu dispatcher
...
if (strcmp(menu_message,"run_form") == 0)
 {
 form frm(param,form_processing);
 frm.set_rmfn(delete_func);
 frm.run();
 return;
 }
...
```

In this example all Seachange constructs are hidden within the private and protected parts of the class. Two constructors are provided :- the first constructor supplying the pointer to the dispatcher function and a string representing the environment in which the form is to be run, which is subsequently parsed by the form constructor. (This is necessary as Seachange only allows one parameter to be passed through the dispatcher.) The second constructor, when called within the dispatcher environment, gives a object representing the current form. A more specialized form is derived from this providing one constructor, only for use as within the dispatched environment. This autosrc_form forces the form, upon startup to locate all the records in the associated file and display the first one.

```
class autosrc_form : form {
    ...
    public:

    autosrc_form();
    };

// Inside the form dispatcher

...
if (strcmp(form_message,"automatic_search") == 0)
    {
    autosrc_form cr_form;
    return 1;
    }
...
```

5.3 Interaction objects

In many cases communication between distinct interface classes of a given service, that is interface classes belonging to the sane service which are not derived from a common base class, takes place through the use of interaction objects based on underlying Seachange types.

What are interaction objects ?

It is often desirable for two or more classes to communicate some of their private information. This can be achieved by making one class a friend of another or by providing member access functions for one or more of the classes. However under either of these two methods, the consumer of the information must know about all the types which will communicate with it. An alternative to this is to allow the creation and consumption of interaction objects. One class produces an object for consumption by another class as a means to communicate information. The objects themselves have no functionality except to serve as packets for information passing. The scheme basically goes as follows:-

- Object Producer knows how to produce objects of type I.
- Object Consumer uses objects of type I to produce some service.
- Therefore any new class introduced to the system can use Consumer's services by providing objects of type I.

In the case of STMS these interaction objects can be provided by some of Seachange's underlying types. Some of the underlying types provide ready-made interaction objects which are created and returned by the 4GL C interface functions (and hence available to the interface classes). As long as the restriction that only the interface classes manipulate these objects is applied , system flexibility is maintained.

The following examples are taken from the file service which consists of a record, file and index classes. In each case the interaction objects SC_record and SC_file are passed as pointers. Each class can be constructed from the appropriate interaction object and may also be cast to the interaction object.

```
class record
{
protected:
SC_record *rec_ptr;
short create_flag;
public:
record(SC_file*);
record(SC_record *);
~record();
void set_field(int , long );    // Overloaded functions
void set_field(int , double );  // To set field values
void set_field(int , char* );
...
void get_field(int , long& );    // Overloaded functions
void get_field(int , double& );  // To get field values
void get_field(int , char* );
...
operator SC_record*()  { return rec_ptr; }
};
```

A record may be constructed from the underlying Seachange types SC_record, or SC_file, and may be cast to a SC_record*. Constructing a record from a (SC_record*) creates an object representing the database record and allows easy access through the set_field and get_field functions. Creating a record with an SC_file* creates a record (and allocates space for it) compatible with the file or index which generates the SC_file*.

```
class file
{
 protected:
struct SC_file * file_ptr;
int reference_count; // Number of references to this file
int *reference_addr;
file( file &);    // create a reference to existing file
SCreckey this_key;
 public:
file(SC_file *);
file(char * , iomode );
~file();
operator SC_file*();
};
```

In the case of files, a file can be constructed from the interaction object SC_file, or by specifying the filename and mode of i/o. A third constructor is provided for use by objects derived from file.

An index is considered to be a file with a particular index. This index corresponds to searching a file in a given order based on some key. A file object may have several indexes, hence the need for reference counting within the file object.

Each of these objects can be constructed from a SC_file*, or cast to a SC_file*.

```
class index : public file
{
 SC_scanstate *scst;
 SC_index *inx;
 ...
 public:
 record kval;              // record for key value
 index(file& rtf,int inxno  = 0);
 index(char* nm,iomode iom,int inxno = 0);
 index(SC_file * fl, int inxno = 0);
 ~index();
 int find(FINDCOND);    // finds record for reading,updating etc.
 int firstrec();
 int nextrec(); // failure returns zero
 ...
 operator SC_file*()  { return file_ptr; }
};
```

In these cases interaction objects narrows the dependencies between the interface classes of a service. Construction of files, indexes and records takes place on the basis of a subset of the available information. In addition to this messages received and returned by the dispatchers are passed in the form of these interaction objects. This allows for a more efficient solution for the application. I.e. files, indexes and records only need to be created if the information and services given by the interface class are to be used. In all other cases the smaller interaction object pointer is passed around.

5.4. Sets.

The concept of sets allows collections of anonymous objects to be manipulated, either individually or as a collection.

In STMS a variant on sets is used to manipulate a group of records retrieved from searching an index, entry into a form or querying a form. Sets of records can be opened for navigation, application of functions to the current item or all items, and for inspection. In STMS an initial action is supplied to the working set indicating whether the current set is to be used, discarded or a copy of the current set is to be used. The set may be extended or reduced by addition or deletion of keys to the respective records.

A function can be applied to the current object with a "vararg" style parameter list. This parameter list is a list of arguments which are passed to every invocation of the function on a set object. The return value from this function is dictates the movement within the set - whether the next invocation of the function is applied to the next, previous, first or last item. If the movement cannot be achieved or a SET_QUIT message is passed back, the collective processing finishes.

```
class working_set {
...
public:
working_set(wks_init wki);  // Initial action
~working_set();
void to_all(auto set_move (*f)(form&,record&,SCreckey*,PRMLIST), ...);
void exec(auto set_move (*f)(form&,record&,SCreckey*,PRMLIST), ...);
// Insertion and deletion
void add_all(index&); // Add all keys retrieved by a index
void add(SCreckey& rk);
void del();
// Current status
void current_record(record&);
...
// Navigation
int first();
int last();
...
};
```

For example displaying all records in the set in sequence on a form could be accomplished as follows.

```
show_item(form& cr_form, record& rec, SCreckey *key_ptr)
{
...
cr_form.show( rec );
...
}

{
...
form current_form;
working_set wk;
...
wk.to_all( show_item );
}
```

In producing a prototype for STMS other C++ features such as references and function and operator overloading also proved to be of value, resulting in a neater product.

6. PERFORMANCE CONSTRAINTS AND METRICS

The hybrid of C++ and 4GL proved to be effective for prototyping this type of application. The resulting design is maintainable, flexible and extendible. The following are a few points worth mentioning.

- Code size. The sources ran to 90K of C++ and 88K of 4GL script for the completed application.

- Development time. The project was completed and delivered with two person months, with no overrun.

- Learning curve. The learning curve from scratch for the 4GL and its C interface ran to about one week.

- Performance. The gain in performance was slight in many cases. This is attributable to two things. Firstly the interface classes all eventually use 4GL features resulting in the overall system being constrained by the 4GL performance. Secondly the application itself is mainly i/o bound both in terms of disk manipulation and user i/o.

- Extensibility. In several cases, forms and reports were re-implemented in C++. These cases proved to be speedy to re-implement (several hours) and gave large performance gains in the respective cases.

7. CONCLUSIONS

This project was by no means perfect. Several areas could be improved. One particular area is sets. These are currently restricted to Form environments, however by generalising

the concept of sets, a further data abstraction could be provided for filing systems etc. However these misgivings aside, I believe this strategy provides a speedy, efficient means for providing extensible, flexible prototypes.

It is worth noting that it proved to be quite trivial to interface C++ to systems designed for C. By implementing all package dependencies through interface classes, all further coding was achieved in C++ with no dependencies on outside libraries and packages.

Finally experiences showed that C++ with its rich set of constructs is as much at home producing applications which were formerly the domain of languages such as COBOL, as it is for producing systems software. This hybrid development provides a useful compromise between using application generators for prototyping and prototyping in a conventional language.

8. REFERENCES

[1] J.Martin, "Fourth-Generation Languages", Vol. 1, Prentice Hall.

[2] S.K.Misra and P.J.Jalics, "Third-Generation versus Fourth-Generation Software Development". IEEE Software July 1988.

[3] John Carolan, "Object oriented Design and C++ syntax", UNIX Forum III, Vienna, October 1987.

[4] John Carolan, "Techniques for Object-Oriented Design applied to Windows and OS/2 applications in C++", Boston Computer Society seminar, June 29, 1987.

[5] William E. Hopkins, "Experience in Using C++ for Software System Development", USENIX proceedings and additional papers, C++ Workshop, Sante Fe, June 1987.

Open Dialogue:
Using an Extensible Retained Object Workspace to Support a UIMS

Andrew Schulert and Kate Erf
Apollo Computer Inc.[1]

Introduction

Open Dialogue (TM) is a User Interface Management System (UIMS) written in C++. In general, its object-oriented design maps well onto the features of its implementation language. However, the goals of Open Dialogue required us to introduce features outside of the language that would more naturally have been incorporated into the language. The most significant of these were the ability to save and restore a collection of objects (*retained object workspace*) and the ability for application developers to add behavior to objects without recompiling existing binaries (extensibility). This paper discusses the goals of Open Dialogue, explains the difficulties in realizing these goals, and describes how they were addressed in the design. It concludes with a summary of possible implications for C++.

Open Dialogue Overview

Open Dialogue is based on a previous product, Domain/Dialogue (TM), that is written in Pascal, and runs only on Apollo systems [7]. The two primary of goals of Open Dialogue above and beyond Domain/Dialogue were that it be portable and that it be extensible. We considered C as an implementation language because of the need for portability. However, C was not adequate. Both Domain/Dialogue and Open Dialogue have an object-oriented design. We were able to maintain that design when implementing Domain/Dialogue, but only because there was a small group of developers, all of whom understood the conventions to be followed. This was not the case with Open Dialogue, since customers had to be able to extend it themselves. We chose C++ as an implementation language because of the additional support it provided for object-oriented programming.

A user interface management system allows a user interface to be described separately from its associated application. Encapsulating the interaction between the user and the application in the UIMS has several advantages as follows:

- It is possible to provide tools for defining the user interface that are more appropriate than conventional programming languages.
- It is possible to define multiple interfaces to a single application.

[1] Andrew Schulert is now at On Technology, Inc.

o Rapid prototyping is encouraged by allowing the user interface to be changed without affecting the application.

Other advantages of this approach and issues related to it are given in other papers [5, 9].

Open Dialogue allows a user interface to be described declaratively, as a set of interrelated objects that cooperate to interact with the user, transform the data passing between the user and application, and coordinate the sharing of control with the application. Developing an application with Open Dialogue involves following a sequence of steps to design, implement and refine the interface. We will illustrate this process with a simple example called "square."

The first step in creating the interface is to determine how it will look and behave. Square displays a field into which the user can enter a number, an area called a label where the square of the number is displayed, and a button that can be selected to exit the application. Figure 1 shows this interface.

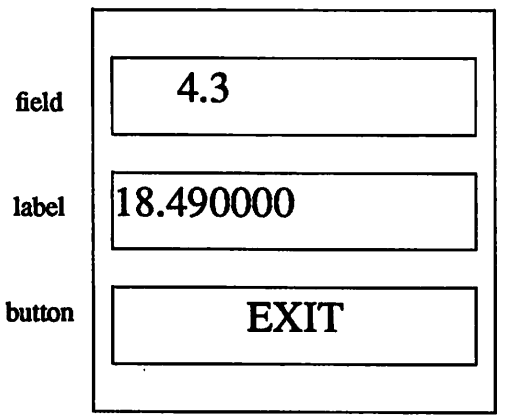


Figure 1. SQUARE -- A Program for Computing Squares.

The second step is to define the set of objects that are needed to support this interface. It is useful to divide Open Dialogue objects into three categories. *Interaction* objects interact with the user to acquire commands and data, and to display results. The interaction objects in square are the field into which the user types, the label that displays the result, and the exit button. *Application* objects manage the passing of control and data between Open Dialogue and the application. Square has two application objects: one of these objects calls an application subroutine to compute the square of a number, and the other returns to the application when the user wishes to exit from it. Finally, it is often the case that the form of data that is most convenient for the user is not the form that is most convenient for the application. *Transformation* objects transform data from a type convenient for the user to a type more

suited for the application. In the case of square, the user enters and expects to see strings, but the application expects floating point numbers. There are two transformation objects to convert the data appropriately-- one to convert a string to a double and one to convert a double to a string.

Each Open Dialogue object takes a set of input values and provides a set of output values. An interface is constructed by connecting appropriate input and output values. Figure 2 shows a schematic of the set of objects used in the interface for square.

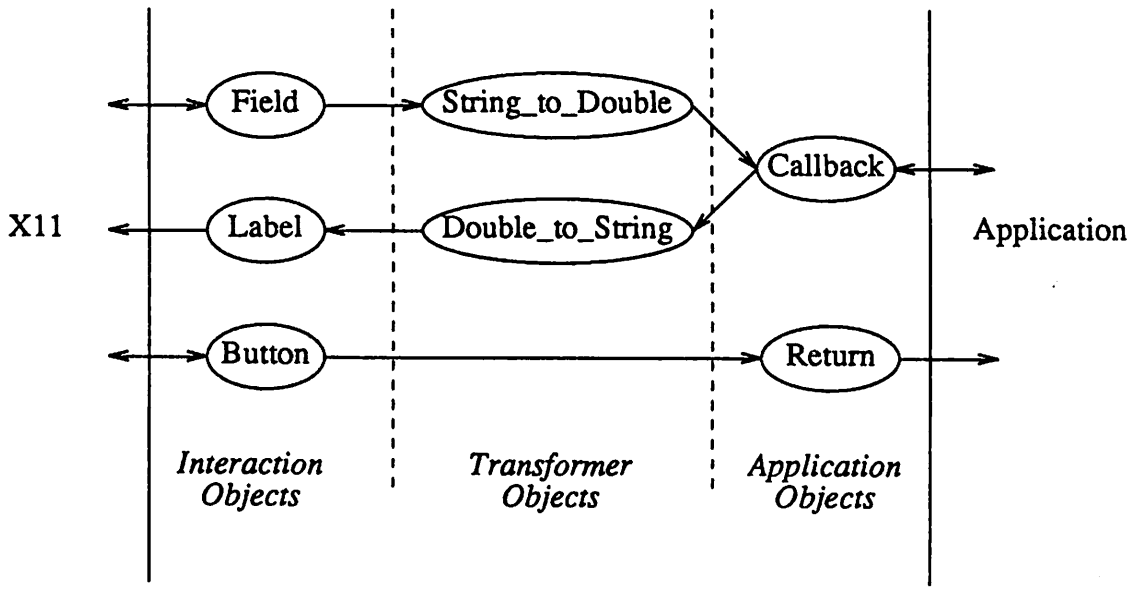


Figure 2. Schematic Representation of Square.

Figure 3 shows the development environment provided by Open Dialogue. This figure is drawn to show the similarity between Open Dialogue and the model given by Tanner and Buxton [9].

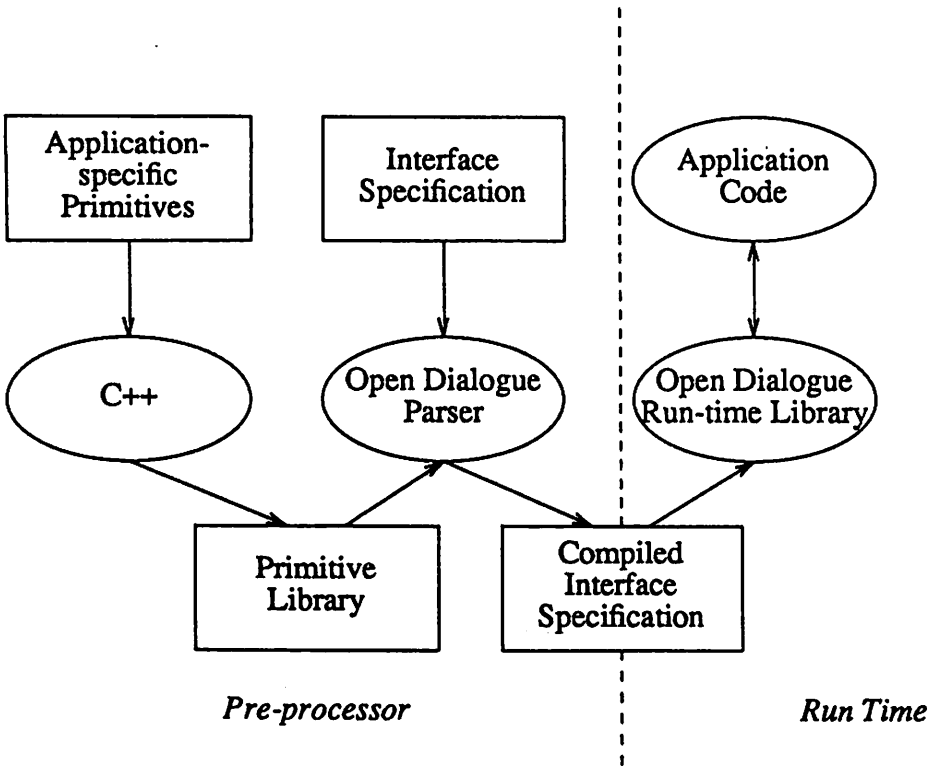


Figure 3. Development Environment Provided by Open Dialogue.

Open Dialogue comes with a standard set of primitives (*primitive library*) implemented as C++ classes. The user interface designer describes the primitives needed for a specific interface in a textual *interface specification*. This is compiled by the Open Dialogue *parser* into a *compiled interface specification*. At run time the application uses the *run-time library* to display and manage the user interface.

The set of primitives provided by Open Dialogue may not be sufficient for all applications. For example, in square the application writer might prefer an alternate way of having the user enter a number, such as a graphical dial. The application could achieve this by calling X library routines directly, but that would preclude the advantages of a UIMS. Consequently Open Dialogue allows customers to extend the standard set of primitives by creating new classes. So, for instance, a dial primitive can be added by the developer and used not only in square, but also in other interfaces.

In addition to defining new classes it may be desirable in some cases to implement new behavior for existing graphic object classes. For instance, Open Dialogue graphic objects, such as menus and fields, have member functions that return the amount of screen space they would like. However, they cannot request that the space be in a particular aspect ratio. One could imagine adding a new layout manager that took aspect ratio into account. In addition to adding the new layout manager class, one would also like to implement a default implementation of the aspect ratio member function for all existing classes.

A second type of extensibility within Open Dialogue is the ability to add new interface definition tools. One can imagine many alternatives to the parser for defining interfaces. For this reason, Open Dialogue allows the construction of new tools that can either generate new or modify existing compiled interface definitions. One possibility is a schematic editor that displays an interface in a form similar to Figure 2 and allows the user to establish connections visually.

A new interface definition tool might also require adding behavior to existing classes. For instance, in the case of a schematic editor, one might want a class-specific visual representation of the object. The editor can provide defaults for existing classes (e.g. a box with input and output arrows) and then allow new classes that are aware of the editor to implement their own representations that look more realistic.

There are two aspects of Open Dialogue that were difficult to implement in C++. Allowing an interface to be defined separately from the application requires a retained object workspace. Second, allowing developers to add new primitives and tools requires extensibility, including the ability to add new behavior for existing classes without access to the source code for those classes.

Open Dialogue Design

This section discusses how Open Dialogue was designed to address the issues raised above.

Retained Object Workspace

We considered three alternatives for saving and restoring objects. The first was the use of *save and restore* member functions. This requires each class to supply its own procedures for saving to a file and restoring from a file. This allows the member function for a specific class to recursively copy all referenced objects by invoking their respective save and restore member functions. This is the approach taken by Andrew [6] and by OOPS [4]. It is also the approach we took with Domain/Dialogue.

The advantage of this approach is that it is very general. It places no restrictions on the format of object data. It also allows the save and restore procedures to take into account the semantics of the data. Gorlen, in his discussion of OOPS, gives the example of a hash table that compacts itself when being saved. The disadvantage of this approach is that it requires these two member functions to be implemented for all classes and to be updated with each change to the object structure. Adding a new

member variable requires updating three different places: the class definition, the save member function, and the restore member function. This is tedious and error-prone.[1]

The other alternatives we considered both avoid the need for class-specific save and restore procedures by proposing that a single procedure be written to handle all objects. This can be done if the procedure can determine the length of the object and the type and location of its member variables.

The second alternative considered was *self-describing object data.* With this approach, all objects are represented in such a way that no information other than an object reference, not even the object's class, is necessary to determine its structure. This is true of most implementations of Smalltalk and Lisp. For instance, in most Smalltalk implementations [3] each object contains a length field. Within each object, each member variable has a bit that indicates whether it is an in-line value or a pointer. On conventional machines (those without tag bits and with only even addresses) this is done by shifting in-line values left one bit and setting the low bit. This allows a machine-dependent object workspace to be easily created by saving the in-line values directly to the file. A machine-independent object workspace would require further information about the type of the in-line values (e.g., floating point vs. char) so that they could be stored in a machine-independent form.

The advantage of self-describing object data is that it requires no class-specific information, simplifying the definition of new classes. However, there is a problem with taking this approach with C++. It requires adding a lot of mechanism for the manipulation of in-line values, since the values must be decoded (e.g., shifted right) before use and encoded before storage. This added mechanism is also inefficient if the target machine does not have tag bits.

The third approach was the use of *class-specific object descriptors.* This is similar to self-describing object data, except that the object layout is determined not by examining the object directly, but by accessing a class-specific data structure that describes the layout. Objective C uses this approach [1].

Class-specific object descriptors retain the disadvantage of save and restore member functions, since adding a new instance variable requires editing both the class definition and the object descriptor. It does minimize the problem, though, since only two places must be updated instead of three.[2] As with self-describing object data, there needs to be some representation of the type of in-line values if there is to be a machine-independent save and restore.

If our only concern were a retained object workspace, then we probably would have used save and restore member functions. However, there are many additional advantages to having a standard object layout. Stroustrup gives the examples of debugging and printing routines. Others include:

[1] Stroustrup suggests that these member functions could be generated automatically by the compiler or a preprocessor [8]. We discuss this briefly in the section on implications for C++.

[2] Like save and restore member functions, the data descriptor could be generated automatically by the compiler or a preprocessor.

o garbage collection -- A fundamental part of garbage collection is copying a live object from memory that is about to be reclaimed into a fresh area. This is similar to saving an object to a file. In most cases a single procedure could perform the copy automatically if it knew which member variables referred to other objects.

o analysis of space usage -- For performance reasons, it would be useful to know which objects were most heavily used and how much they contributed to the overall size of an object workspace. This analysis could be done automatically with the use of object descriptors.

For this reason, we took a combination of the first and third approaches above. Open Dialogue classes are required to have save and restore member functions. However, we provide a default implementation of these member functions that makes use of a class-specific object descriptor. Most classes simply define the object descriptor. A few of them, like hash tables, override the default behavior, implementing their own save and restore member functions.

The object descriptor contains:

o The name of the object (for debugging purposes).

o The length of the object.

o A record for each member variable within the object.

The record for each member variable contains:

o The name of the member variable (for debugging purposes).

o The length of the member variable.

o The kind of data in the member variable. This is one of:

-- Explicitly typed data -- the data is a "real" object, and has its type embedded within it.

-- Implicitly typed data -- the data is a primitive object. Its type is not embedded within it, but is indicated within the object descriptor.

-- Data described elsewhere - the data is described by some other object descriptor that is referenced by this object descriptor.

o How the data is referenced. This is one of:

-- an in-line variable - the object contains the data directly.

-- a pointer - the object contains a pointer to the data.

-- an array - the object contains a pointer to an array of data elements.

(This field is only for efficiency and simplicity. We could have introduced new primitive types for pointers and arrays.)

The object descriptors rely on the existence of what we term *primitive objects*. These are not to be confused with the primitives that the user interface designer works with. These are the primitive values out of which other objects are composed, and have no embedded object type. Examples of primitive objects include strings and integers. Something like this is needed as the basis for any object-oriented system; C++ relies more heavily on them to avoid the overhead of typed objects. While the type of a primitive object is not stored with the object, it *is* stored in the object descriptor of any object that contains the primitive object. This allows a form of polymorphic function invocation on primitive objects. In other words, primitive classes can have their own save and restore member functions. A procedure that is processing an object by making use of the object descriptor can invoke the appropriate function for a primitive object by using the type stored in the object descriptor.

In addition to using object descriptors for save and restore, we use them for a simple form of garbage collection. Open Dialogue does not have a general purpose garbage collector. However, we have implemented a separate "copy to new heap" member function that we use to collect garbage and compact the retained object workspace before saving it to a file.

Extensibility

As described above, adding new primitives and tools requires the ability to add new behavior to existing classes. Because all extensions are not developed by the same person or institution there is the additional constraint of not requiring existing classes to be recompiled. In other words, we give customers a header file containing a class definition and an object (.o) file that implements its behavior. The customer has to be able to define extensions to the class behavior in a separate header file and provide an implementation of those extensions for that class, for subclasses that s/he develops and for subclasses that are developed elsewhere.

Virtual functions work well when all behavior in the class hierarchy can be predetermined. However it is not possible to add new virtual functions for existing classes without recompiling all modules that use these classes. We use *traits* to achieve this capability. Our trait model is adapted from the Apollo trait system, which was designed to handle polymorphic operations on files. (Apollo files are typed objects.) We adapted this design to apply to smaller grained objects, objects within files as opposed to the files themselves.[1]

[1] Our traits are also similar to the trait facility used in the implementation of the Xerox Star. The emphasis in that work

Traits are a collection of related operations that describe a particular behavior. The operations for a trait are implemented as member functions. A given class either implements all of the operations for a given trait or none of the operations. For instance, the *graphic object* trait is supported by all classes that can be displayed on the screen. This trait has operations for accepting input events, requesting screen real estate and drawing to the display. Some operations, such as drawing an object, need to be implemented for every graphic object class because all objects appear differently. However, as with virtual functions, other operations, such as dealing with mouse input, may be inherited from the base class if the default implementation is sufficient.

Unlike virtual functions, traits are not correlated with the class hierarchy, i.e. two classes can support the same trait even though their common parent does not.[1] For instance, the *context* trait is supported by all classes that accept keyboard input. This trait has operations for accepting and giving up the typing focus. It is supported by a subset of the graphic objects classes, but those classes are scattered throughout the class hierarchy.

The implementation of traits uses a class member variable in each object. This contains a small integer that identifies an object's class.[2] Each Each class has a trait binding record for each trait it supports. The trait binding record is a table of pointers to member functions for the operations required by that trait. The trait binding records are accessed through trait vectors, one for each trait. A trait vector is indexed by class type. For example, if menu had a class type of 4, then the 4th index in the graphic object trait vector would hold a pointer to menu's trait binding record for the graphic object trait. This is shown in Figure 4.

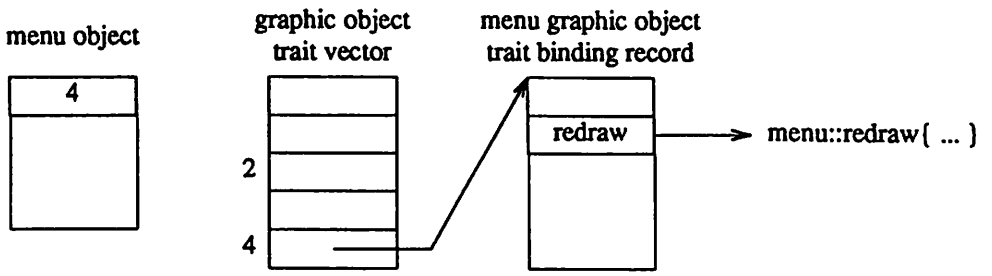


Figure 4. Implementation of Trait Bindings.

was on supporting multiple inheritance rather than extensibility [2].

[1] Traits describe behavioral abstractions. They say nothing about how that behavior is implemented. This is different from a conventional single-inheritance object oriented system such as C++ or Smalltalk. In those systems a class cannot inherit the behavior of another class without also inheriting its member variables.

[2] To maintain uniqueness across classes, class has a universal unique identifier which is 128 bits-- this gets hashed into the small integers stored in each object. This allows an application writer to use classes developed at two different locations without risk of collision.

Traits provide all the features we need to support extensibility. New behavior can be added to an existing class by creating a new trait and a new trait binding record. By default, a class's entry in a trait vector is set to the same as its base class. This can be overridden for both new and pre-existing classes. Invoking a trait operation is not as simple as invoking a virtual function, (by convention macros are used) but it does maintain type checking.

Implications for C++

This section discusses the implications of the design of Open Dialogue on the design of C++. We realize that, while the issues discussed previously might be better addressed within the context of a programming language, it isn't necessarily the case that what is best for Open Dialogue is best for C++. This section makes no explicit recommendations for C++. It simply proposes some alternatives that could be considered, briefly discussing their implications.

Retained Object Workspace

There are three alternatives for better integrating save and restore into the language. The first is to give developers access to an automatically generated object descriptor (either through self-describing object data or class-specific object descriptors). While this would have utility beyond just save and restore, it is difficult to argue for its inclusion into the language, since it subverts type abstractions. More importantly, it requires the reification of classes (classes as objects) to allow the type of a member variable to be given as part of the object descriptor. This would have a major impact on the language.

A second alternative is to take Stroustrup's suggestion and automatically generate member functions that recurse over an object's member variables, passing each member variable the same arguments that the member function has received. This fits the language model much better, but has two drawbacks. The first is that it requires that all member variables support the member function being invoked. This means, for instance, creating explicit types for all pointer variables. The second problem is that it isn't clear how generally applicable the approach is. It is clear how it would work for save and restore, where the arguments passed to each member variable are the same (e.g., a file descriptor). It is less clear how this would work for, say, a deep copy operation used as part of garbage collection. In this case, each member variable is passed the address of the variable it is copying itself to; this will be different for each member variable.

The third alternative is to embed save and restore explicitly into the language. This could be done by introducing standard member functions that are analogous to constructors and destructors. The advantage of explicit language support is that it allows C++ to automatically generate reasonable default behavior. The only apparent disadvantage to this alternative is that it is not more generally applicable. It requires separate solutions for other problems, such as garbage collection and debugging.

Extensibility

The problem with integrating extensibility is not with the language model, as one can imagine extensions to the language for expressing this. The problem is how to implement it efficiently. Extensibility means that there is no way to know the complete set of virtual functions at compile time. There has to be an extension-specific database that can be used at run time to map from some representation of the object's type to the appropriate implementation of the extension. Open Dialogue does this by using the trait vector to map from the object's class member variable to its trait binding record. As currently implemented, C++ relies on there being only a single record of virtual functions, and short circuits the database by having a field in each object point directly to that record. Changing to a model like that of Open Dialogue would make virtual function invocation less efficient. Another possibility would be to retain the current model for behavior defined with the class, and embed an object type in the virtual function record for use by extensions. This would impose no penalty on existing systems, but would impose a greater penalty on extensions.

Summary

There are two major facilities that we incoporated into Open Dialogue that would have benefited from greater language support. These are saving and restoring objects and the ability to extend the behavior of the system by adding new behavior to existing classes.

Saving and restoring objects is a fundamental operation of any object management system. Because this feature is so tightly linked to the layout of objects as they are defined in C++, this could be most naturally implemented within the objects themselves as a standard member function similar to or combined with constructors.

Extensibility of behavior cannot be completely implemented through virtual functions. There needs to be a mechanism within C++ that allows new virtual functions to be added to existing base classes without having to recompile the code. Open Dialogue was able to work around this with the trait mechanism, but this required additional work that could be handled within the implementation language.

Bibliography

[1] Brad J. Cox. *Object-Oriented Programming*. Addison-Wesley, 1986.

[2] Gael Curry, Larry Baer, Daniel Lipkie, Bruce Lee. Traits: an approach to multiple-

inheritance subclassing. *Proc. SIGOA Conference On Office Information Systems*, (Philadelphia, June 21-23, 1982), ACM, New York. pp. 1-9.

[3] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation*. Addison-Wesley, 1983.

[4] Keith E. Gorlen. An object-oriented class library for C++ programs. In *USENIX Proceedings and Additional Papers C++ Workshop* (Santa Fe, 1987), pp. 181-207.

[5] Dan R. Olsen, Jr., William Buxton, Roger Ehrich, David J. Kasik, James R. Rhyne, and John Sibert. A context for user interface management. *IEEE Computer Graphics and Applications* 4(12):33-42, 1984.

[6] Andrew Palay et al. The Andrew Toolkit-- An overview. In *USENIX Conference Proceedings, Winter 1988* (Dallas, February 9-12, 1988). pp. 9-21.

[7] Andrew J. Schulert, George T. Rogers, and James H. Hamilton. ADM -- A Dialog Manager. In *Proc. CHI '85 Human Factors in Computing Systems* (San Francisco, April 14-18, 1985), ACM, New York. pp. 177-183.

[8] Bjarne Stroustrup. Possible directions for C++. In *USENIX Proceedings and Additional Papers C++ Workshop* (Santa Fe, 1987), pp. 399-416.

[9] P.P. Tanner and W.A.S Buxton. Some issues in future interface management system (UIMS) development. In Gunther E. Pfaff, editor, *User Interface Management Systems*, pages 67-79. Springer-Verlag, 1985.

A C++ Class Hierarchy for Building UNIX-Like File Systems*

Peter W. Madany, Douglas E. Leyens
Vincent F. Russo, and Roy H. Campbell

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801

Abstract

Class hierarchical object-oriented programming languages like C++ facilitate the construction of organized libraries of related data structures and algorithms. In operating systems research, it is convenient to build such libraries to support system abstractions. In our Choices [3] parallel operating systems research, we have been experimenting with new and existing file system facilities in an attempt to design an object-oriented file system implementation.

This paper describes a classification of the data structures and algorithms used in UNIX-like file systems and an implemention of them using C++. We present a class hierarchical organization for the System V [8] and 4.2 BSD [4] file systems that reflects the common subcomponents, abstractions, and interfaces that these systems share. Because of the flexibility afforded by designing such systems in an object-oriented language, new specializations of the abstract file system can mix and match components from existing implementations forming hybrid systems.

We conclude by discussing the performance of our system and the influence of C++ on our design and organization.

1 Introduction

This paper describes an experiment in the classification and implementation of data structures and algorithms used in UNIX-like file systems. Our long-term purpose is to provide

*This work was supported in part by NSF grant CISE-1-5-30035, by NASA grant NSG1471, and by AT&T ISEP.

a foundation for further research into object-oriented file systems; however, the immediate goal is to combine the UNIX philosophy of file systems and the Choices philosophy of object-oriented operating system components.

In the following three subsections we will describe Choices, the System V file system, and the 4.2 BSD file system extensions.

1.1 Choices

Choices is a family of operating systems that can be customized to a particular multiprocessor or parallel application [1] [2]. Object-oriented programming and class hierarchies are used to facilitate the building and customization of the family. C++ was adopted as the programming language because it provides an efficient implementation of objects and classes [7].

A Choices system is an object-oriented operating system that uses persistent objects to provide facilities and services to client processes. Choices persistent objects have lifetimes independent of user processes. Many of these subsystem facilities and services would belong in the kernel of a more "traditional" operating system. However, persistent objects allow an application to load only those subsystems that it needs. Persistent objects can provide secure services because Choices uses virtual memory protection mechanisms to restrict access to the objects.

The file system is one of the more important subsystems provided in an operating system. In Choices, we have chosen to implement the file system as a collection of persistent objects; each persistent object implements an independent component of the file system. Using this technique, an application may use a file system composed of many different components, each tailored to improve the performance of the application, to optimize the use of the storage technology, or to provide compatibility with file systems of other operating systems.

Currently, we have completed two different UNIX file systems: the 4.2 BSD file system [4] and the System V [8] file system. The classes of the two file systems are specialized from one, abstract, UNIX-based file system class hierarchy. However, many of the concrete classes realizing the two systems are very different from one another. Further, it is possible to combine file system components from UNIX BSD and System V implementations to produce hybrid systems that combine the features of both. For example, the efficient BSD disk allocation methods and larger block sizes can be combined with the System V directory structure to yield a system with higher throughput without having to rewrite any user level code that relied on a System V record structure for directories. Alternately, individual features such as symbolic links or disk quotas may be added to the System V file system, as needed.

C++ has been a aid in developing our file system implementations. The language was useful because it directly supported the development of the abstract classes that formed the framework for our systems. The abstract interfaces permitted concurrent development and debugging of the various components of each file system. The virtual function feature allowed us to simplify much code. The ease of developing and reusing C++ classes led to

much code reuse, both within a file system and between different file systems.

1.2 System V

The System V [8] file system is the standard file system model found in today's commercially available UNIX systems. Its design is dominated by simplicity.

Basically a user program can view a UNIX file as a sequence of randomly accessible bytes. All files can be accessed via the same standard interface: `read`, `write`, and `lseek`. This interface conceals hardware device dependencies and hides block allocation and block mapping. Because the operating system does not impose record structures on files, the output of most UNIX tools can be the input of others. Nevertheless, any tool can impose a structure on a file. Efficient implementation of random access allows even complex record structures, such as ISAM, to be imposed on specific files when needed.

Disk drives in UNIX systems are divided into logical sub-devices, called partitions, each of which contains one file system. A file system consists of a header for the system called a superblock, information about which disk blocks are available for allocation, and an array of inodes that describe individual files. While file systems cannot span disk partitions, a single directory tree contains all the files on all the file systems. The directory tree hides individual disks and partitions from the user.

The inode is a structure that describes an individual file and and manages access that file. Within a UNIX system, a file can be uniquely identified by specifying its partition and the inode array index number, called the inumber. An inode contains its file's size, reference count, ownership, access rights, timestamps, and the numbers of the blocks which hold the file's data.

Directories are sequences of records that contain (*name*, *inumber*) pairs. Because directories contain inumbers instead of complete inodes, files can appear in more than one directory at a time. Files are only deleted when their reference count reaches zero.

The System V file system's performance is marked by two impressive characteristics: high disk space utilization and low CPU overhead per block transferred. However, there are some deficiencies in both its performance and feature set that have been addressed by the design of the 4.2 BSD file system.

1.3 BSD

The 4.2 BSD file system [4] maintains the same basic interface as System V and adds optimizations and extensions.

The penalty incurred by the System V file system per individual block transferred is small. Its overall throughput is dominated by disk latency. To minimize the disk latency and thereby improve overall throughput, the 4.2 BSD file system increased file block sizes and improved inode and disk block allocation policies.

An 8192 byte block improves throughput almost sixteen times when compared to a 512 byte block. To maintain the high disk space utilization of System V, 4.2 BSD added the

capability to fragment the last block in a file. The improved disk block and inode allocation policies minimize both disk head seek time and rotational latency.

Three of the major extensions provided by the BSD file system are symbolic links, long file names, and per-user disk quotas. Symbolic links allow users to create directory entries which refer to files on different file systems. In System V, file names are restricted to 14 characters because of the fixed-size record structure used for directory entries. The 4.2 BSD file system uses a variable-size record which allows file names to be up to 255 characters long. Disk quotas allow system administrators to restrict individual users to using only a portion of the space in a file system.

The following sections discuss the class and instance hierarchies in our system and are followed by discussion of performance and directions for future work.

2 A Class Hierarchy for File Systems

The use of *class hierarchies* has been proposed as a solution to some of the traditional design and engineering problems in today's software development lifecycle [6] [5] [7]. In particular, class hierarchies support code reuse and the sharing of common interfaces among different implementations. A class in a class hierarchy encapsulates an interface and a possibly empty implementation. The interface, or *signature*, of a class is defined by the set of *methods* or *operations* the class defines for its instances. The implementation of the methods of a class can either be defined by the class itself or can be defined by other classes that are derived from the class through class inheritance. [1] A class in a hierarchy can define or augment an interface, an implementation, or both. Classes that define only an interface and have subclasses that supply implementation are *abstract* classes. Subclasses that define an implementation for a particular interface are termed *concrete* classes. Most classes are neither concrete nor abstract; they often redefine only a portion of an implementation or augment an interface with a few additional methods. A subclass can customize an implementation of a superclass for specific applications and may share all, some, or none of its implementation with its superclass. Class derivation provides a framework for changing specific parts of a system without altering the whole structure.

The following sections describe the majority of the classes in a hierarchy to implement UNIX-like file systems. Figure 2 shows this hierarchy.

[1] The classes that have methods that are inherited are usually termed *parent* or *super* classes. The classes that inherit methods are usually termed *derived* or *sub* classes.

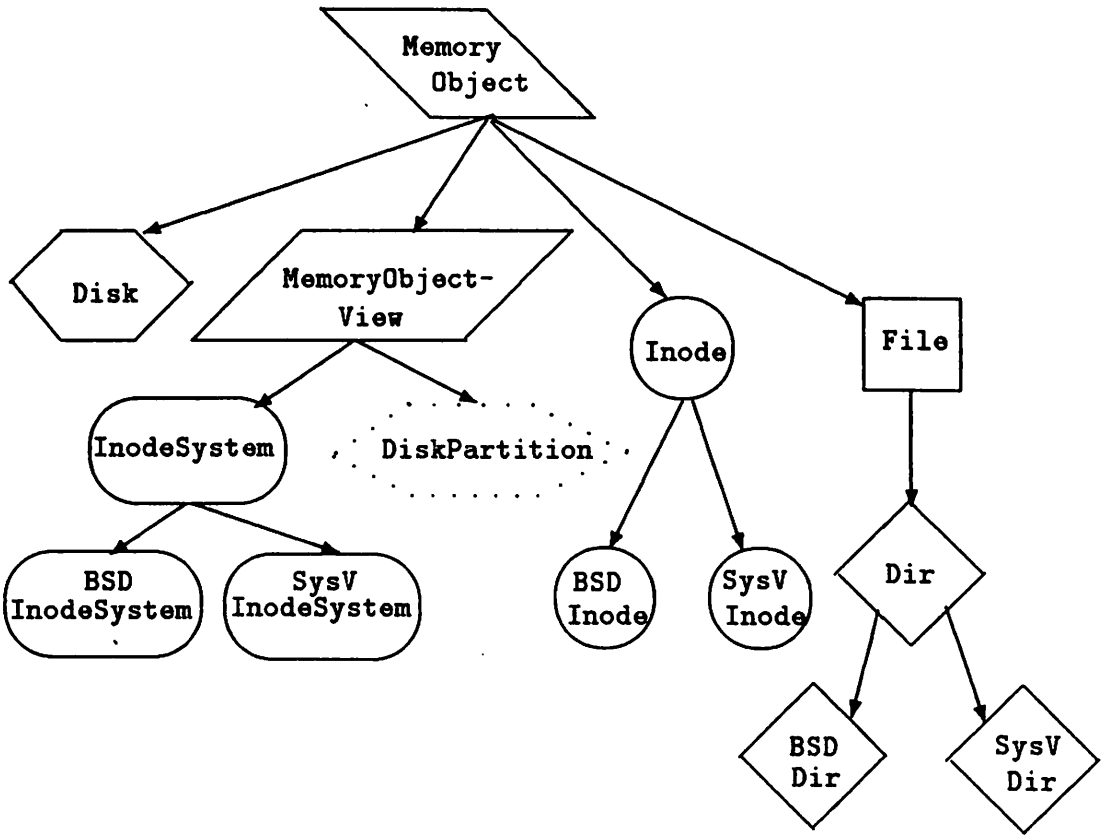


Figure 2: MemoryObject Hierarchy

2.1 MemoryObject

The superclass *MemoryObject* abstracts both the Choices file system and memory management systems. It defines an interface which permits access to a block of data that may either reside on permanent storage or be generated dynamically. The interface uses a read-unit/write-unit protocol. The units used for reading and writing are all the same size within an individual MemoryObject, and this size must be an integer power of two. Subclasses of MemoryObject augment the protocol and provide various implementations of the methods involved.

In Choices, MemoryObjects are most often accessed by mapping them into a process' virtual address space. The system caches portions of the MemoryObject into physical memory and provides the address translation mechanisms necessary for the process to address it with the read/write instructions of the CPU. A MemoryObject can, however, be accessed directly by its `read/write` interface.

Class	Common Public Methods					
MemoryObject	*read*	*write*	*close*	*open*	*create*	*synchronize*
↑ Disk	read	write	–	–	–	–
↑**Inode**	read	write	close	–	–	–
↑↑BSDInode	↑	write	↑	–	–	–
↑↑SVInode	↑	↑	↑	–	–	–
↑**MemoryObjectView**	read	write	–	–	–	–
↑↑DiskPartition	read	write	–	–	–	–
↑↑**InodeSystem**	↑	↑	–	*open*	*create*	*synchronize*
↑↑↑BSDInodeSystem	↑	↑	–	open	create	synchronize
↑↑↑SVInodeSystem	↑	↑	–	open	create	synchronize

Class	Protected Methods				
↑**Inode**	mapUnit	*getDirect*	*getIndirect*	*setDirect*	*setIndirect*
↑↑BSDInode	↑	getDirect	getIndirect	setDirect	setIndirect
↑↑SVInode	↑	getDirect	getIndirect	setDirect	setIndirect

Class	Protected Methods						
↑↑**InodeSystem**	get	put	*free*	*allocate*	readDinode	writeDinode	*getFreeInode*
↑↑↑BSDInodeSystem	↑	↑	free	allocate	↑	↑	getFreeInode
↑↑↑SVInodeSystem	↑	↑	free	allocate	↑	↑	getFreeInode

Legend	
Symbol	Meaning
Boldface	Abstract class.
Italics	Abstract definition of method.
Roman	Concrete class or method.
↑	Subclass or inherited method.
–	Undefined method.

Table 1: MemoryObject Class Hierarchy.

2.2 MemoryObject Subclasses

The following paragraphs discuss individual subclasses of MemoryObject and their particular functionality. Table 1 and Table 2 show the class hierarchy using the format introduced in [3].

The *Disk* subclass of MemoryObject represents the physical disk devices in a system. It provides an abstract interface and access protocol to these disks. It is further subclassed for specific hardware architectures and devices.

It is usually inconvenient or inefficient to copy a MemoryObject into virtual memory. The *MemoryObjectView* subclass of MemoryObject provides a window into another MemoryObject. The size of this window can range up to the size of the MemoryObject being viewed. The window may be offset from the start of the MemoryObject. Its purpose is to restrict access to the MemoryObject under the window. Several MemoryObjectViews may exist for the same MemoryObject.

A *DiskPartition* is simply an instance of MemoryObjectView that windows a sub-range of a Disk. The size and offset of the window is defined by the Disk's hardware partition table.

The *InodeSystem* class is derived from MemoryObjectView and inherits its `read` and `write` methods. One InodeSystem exists per DiskPartition and contains a UNIX file system. The InodeSystem is an abstract class definition that provides the framework for UNIX-like file systems. It contains the code for all methods that have the same implementation in the derived classes. All common methods are implemented in this class to reduce the overall code size and programming effort. The other methods defined here are needed by the subclasses, but since they will be different, they cannot be inherited.

The two major subclasses of InodeSystem implemented are *BSDInodeSystem* and *SVInodeSystem*. Many of the methods of BSDInodeSystem and SVInodeSystem perform identical functions but use different data structures or algorithms. The class InodeSystem contains the code common to both the BSDInodeSystem and the SVInodeSystem. It also provides virtual functions for methods that are implemented differently in these subclasses. For example, inumbers must be mapped to physical blocks by the `readDinode` and `writeDinode` methods. A `mapInumber` method is defined as a virtual function in the InodeSystem. Each subclass implements this method in a different way. However, both the `readDinode` and the `writeDinode` methods can be implemented in the InodeSystem and this implementation can be inherited by the subclasses. The `readDinode` and `writeDinode` methods of BSDInodeSystem and SVInodeSystem use the implementation of `mapInumber` that is appropriate to the subclass of the instance upon which the methods are invoked. Similarly the `get` and `put` methods are inherited but need a variable containing the fragment-to-sector conversion factor to be appropriately initialized by the derived class. Such techniques move general code up into the base class where it can be reused instead of requiring it to be rewritten for each new implementation.

Those methods that are sufficiently different between various subclasses of InodeSystem (types of UNIX file systems) are simply defined as empty virtual functions in InodeSystem and redefined by *all* subclasses. For example, the System V superblock contains a free list for

File Class Hierarchy							
Class	Methods						
↑ File	read	write	seek	close	–	–	–
↑↑ **Directory**	*read*	–	–	–	*put*	*locate*	*remove*
↑↑↑BSDDirectory	read	–	–	–	put	locate	remove
↑↑↑SVDirectory	read	–	–	–	put	locate	remove

Table 2: File Class Hierarchy.

both free data blocks and free disk inodes while the BSD system uses bitmaps. `Allocate`, `free`, and `getUnusedInode` have sufficiently different implementations that they cannot share code, only an interface.

The *Inode* is an abstract class that provides a framework for a UNIX-like in-memory inode object. As in the InodeSystem, common code is moved into the base class and inherited by the BSD and System V derived classes. These methods are mostly private methods used to calculate disk block pointers and and manage internal caches of indirect blocks. There is also a method, `mapUnit`, that maps logical block numbers to physical file system block numbers and can be inherited by both derived classes. The remainder of the class defines the framework to be used by the derived classes.

The *BSDInode* and *SVInode* subclasses implement the Inode framework according to their particular needs. The differences are due to the ways in which data block pointers are stored, and the other fields in the disk inode structure. For example, methods to set and retrieve the direct and indirect pointers are implemented by each subclass. System V has 10 direct pointers, a single, a double, and a triple indirect pointer. Each of these is stored in three bytes in the disk inode and must be converted to and from an integer. BSD, on the other hand, has 12 direct pointers, a single, a double, and a triple indirect pointer. Each of these is stored as a four byte integer requiring no conversion.

From the user's perspective, an important subclass of MemoryObject is *File*. The File class is both a concrete class used for interaction with any UNIX disk file and an abstract class from which the *Directory* class is derived. The unit size for the File class is one byte. The File class adds the concept of a current file location pointer to the MemoryObject interface and adds the `seek` method to position this pointer. The `read` and `write` methods update this file pointer as well. These methods together provide a byte-oriented interface to user level programs. Each instance of File communicates with a corresponding Inode object which reads and writes blocks instead of bytes.

The Directory class is an abstract subclass of File which adds a directory-entry record structure on top of the blocks supplied by the Inode object. It also provides methods to simplify the insertion, retrieval, and removal of directory entries. Since directories in BSD and System V are different, the methods of this class: `read`, `put`, `locate`, and `remove`, must be defined by each subclass and cannot be inherited.

3 An Instance Hierarchy for File Systems

In this section, we describe the instance hierarchy for a complete working file system.

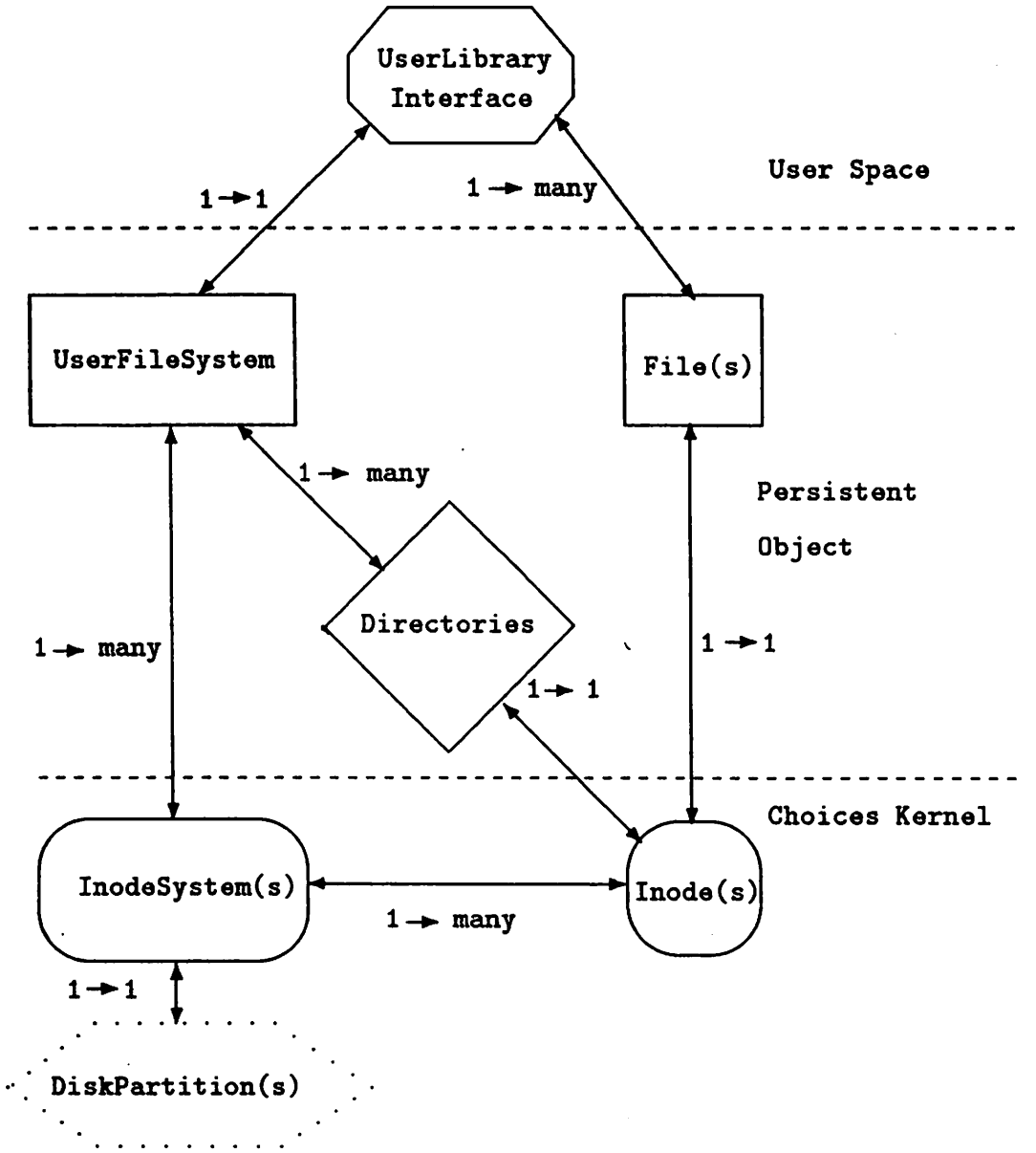


Figure 3: File System Instance Hierarchy

When performing operations on files, user programs must invoke the methods of the User Library Interface. These methods will in turn invoke methods on several other objects in order to perform the requested action. Figure 3 shows the objects involved in these operations and the basic data flow between them. Some sets of objects have a one to one

correspondence; for example, there is one Inode object for each File object. Other sets of objects have a one to many relationship, such as the UserFileSystem which can communicate with several InodeSystem objects.

A user program gains access to all file systems and files via persistent object calls. This can be likened to the system calls used to gain access to a UNIX file system. These calls will be translated into the appropriate method invocations in either the UserLevelFileSystem or a File object. In Figure 3 the set of these calls is referred to as the User Library Interface.

The *UserFileSystem* object views all active file systems as a single tree. A reference is maintained for each InodeSystem in this tree. The UserFileSystem also contains instance variables for pathname resolution that maintain references to the root directory and the current directory. These are needed to correctly implement the operations required of the UserFileSystem.

The public methods of the UserFileSystem are similar to several of the UNIX file system calls including: **open**, **creat**, **link**, **unlink**, **mkdir**, **chdir**, and **stat**. These methods operate on and return references to File objects and Directory objects which may in turn be used by the User Library Interface to perform operations on File objects.

The File object corresponds to a UNIX open file table entry and provides a generic interface to open files for user level programs. Its methods support operations similar to the set of UNIX system calls that operate on open files including: **read**, **write**, **seek**, and **close**. The File object communicates directly with its corresponding Inode object and maintains a current byte offset for implementing **seek** and sequential **read** and **write**.

An instance of Directory is used to impose the directory record structure on a file. Directory methods include **read**, **put**, **locate**, and **remove**. Read returns a directory entry and is used by programs such as **ls**. Directory entries are added and removed from the Directory object's underlying file via the **put** and **remove** methods. The **locate** method finds the inumber of an indicated file name in a directory. All of these methods are invoked by the directory methods of the UserFileSystem. User programs are prevented from executing the **put** and **remove** methods on a directory. The protection is provided by setting the file access mode as opposed to using protected C++ functions.

An instance of an InodeSystem is used for each active file system. Creation of a new instance is similar to the UNIX **mount** system call. The UserFileSystem communicates with the InodeSystem when requesting operations on new and existing Inodes. The InodeSystem communicates with the *DiskPartition* to read and write disk blocks. It also manages the superblock fields, disk inode allocation, and disk data block allocation. It creates and provides the Inode objects when requested and keeps track of in-memory versions of the corresponding disk inode structures.

The public interface to the InodeSystem includes methods that operate on and return references to Inodes. These are **open** and **create**. The interface also includes methods to maintain the data blocks of the DiskPartition for use by the Inodes. These methods are **allocate**, **free**, **get** and **put**. The **synchronize** method is used to write the modified superblock and in-memory versions of disk inodes to the DiskPartition to maintain consistency.

The Inode object contains the UNIX disk inode structure and the methods used to operate on it. These include all information needed to access the file such as size, mode, protection, ownership, and disk block pointers. Once this object is created by the InodeSystem, it may be referenced by a File or Directory object to perform actions on blocks of data stored within the InodeSystem's DiskPartition.

The DiskPartition object maintains the size and starting block location of the partition it represents. It performs disk read and write requests for its corresponding InodeSystem object and checks these requests to ensure that they only access blocks within the range that it manages.

3.1 Choice'ing a File System

Choices supports the concept of customizable operating systems. The file system class hierarchies presented allow a system designer to choose and easily integrate existing, modified, or new concrete components to create new customized file systems.

This mix-and-match approach leads to the following orthogonal "choices" when designing a new file system:

- Fixed or variable-sized file names.
- Per user disk quotas.
- Optimized inode and disk block allocation.
- Large block sizes and fragmented blocks.
- Symbolic links.

Some of these choices involve the selection of a complete specialized subclass, while others simply require creating new concrete subclass with methods from two existing subclasses.

The following section presents performance data measured from our implementation.

4 Performance

The performance of our systems can be characterized in three ways. First, we measured the overhead incurred by all of the Inode and InodeSystem methods as opposed to raw disk reads and writes of the same disk blocks. Second, we checked to make sure that our BSD implementation did not reduce or remove the effectiveness of the BSD optimizations. Third, we observed the effect of altering certain parameters and algorithms used in the Inode class methods.

To calculate the amount of CPU-time needed by all of the Inode and InodeSystem method code, we measured the time to copy a 17 Megabyte file, and then measured the time it took to do a raw disk copy of the same blocks using a simple iterative loop. Both copies were

performed in the kernel. The raw disk copy took 127 seconds, whereas the copy that used the Inode and InodeSystem code took 133 seconds. Therefore, only 5% of the time spent in the kernel while copying a file accounts for all of the block allocation and block mapping code in the Inode and InodeSystem class methods. This corresponds to the System V design goal of low overhead per block transferred.

When designing the interface between the Inode and InodeSystem classes, we took care to ensure that both the BSD block and inode allocation policies were fully supported. We also fully implemented the large block size and the block fragment features of the BSD file system. Therefore, the performance improvements that the BSD optimizations brought to UNIX will also be realized when using the BSD specialized classes under Choices.

After developing the file system code, we measured the effects of altering the block size on the time it took to copy files. Each time the block size was doubled from 512 bytes up to 8192 bytes, the time to copy a file was almost halved. These results confirm those found by the developers of the BSD file system.

Since Choices currently has no disk buffer cache, we added an index block cache to the Inode class. For copies of large files, those between one and sixteen megabytes long, we found the index block cache tripled the speed of file copying operations.

5 Experiences with C++

While building the file system class library, the use of C++ not only enabled but also encouraged an object-oriented programming style. This style in turn helped us to specify object interfaces and to enforce data encapsulation, *which usually allowed us to perform independent development and debugging.* While all the authors contributed as a group to the designs of each class, we were able individually and simultaneously to work on the implementation of the disk class methods, the BSD and System V details, and the user level file and file system methods. Furthermore, by classifying the objects into hierarchies, we were able to achieve both code and design reuse.

The features of C++ that we found most useful were classes, inheritance and virtual functions. A good example is the Inode class, and more specifically its private method called `mapUnit`. The implementation of `mapUnit` needs no information about whether either the Inode object or its containing InodeSystem object conforms to the BSD or System V standard. Once its code has been debugged, it automatically functions equally well for either of Inode's concrete subclasses; in fact, the file containing the code for the abstract class doesn't even need recompilation in order to support additional concrete classes of Inodes. The primary difference between a BSDInode and a SystemVInode is the details of the disk inode representation. In order to allow functions like `mapUnit` to be inherited by concrete subclasses of Inode, virtual functions were defined for disk inode access routines. At run-time, calls to these methods are translated into appropriately redefined concrete subclass methods.

We did find it necessary to make restricted use of friends. Sometimes objects belonging to different classes need more access to information stored in an object of yet another class.

For example, Inodes use the protected InodeSystem methods: **get**, **put**, **allocate**, **free**, and **close**, while the UserLevelFileSystem uses only the public InodeSystem methods: **open**, **create**, and **synchronize**. Even though InodeSystem declares Inode as a friend, an Inode object still never directly accesses any data member of an InodeSystem object. Hence, we do have a suggested enhancement for C++: instead of giving another class access to all the private data and methods of a class via the friend mechanism, it would be useful to make just certain private or protected methods accessible to another class.

In retrospect, the MemoryObject hierarchy suggests the need to use the multiple inheritance feature of C++. Some MemoryObjects, such as InodeSystems, are collections of other MemoryObjects. They should inherit methods **open**, **create**, and **synchronize** from an abstract class MemoryObjectCollection instead of class MemoryObject.

Our systems also benefited from other, somewhat unrelated C++ features such as inline functions and type-checking. When procedure call overhead is eliminated, one no longer has to consider a tradeoff between code modularity and speed.

The **lint**-style type-checking of C++ invariably flags either coding errors or questionable practices, we do not recall it ever getting in the way of code development.

In general, we always felt that the use of C++ provided the same speed and more expressive power than would the use of C.

6 Future Work

We plan to add support for additional existing file systems, including other UNIX file systems such as that of the Ninth Edition UNIX system, and some non-UNIX file systems, such as the MS-DOS file system. Adding MS-DOS classes to the hierarchy will be more challenging, but they will still fit into our existing class hierarchy.

We also plan on implementing experimental file system components to further support our research. In particular, we are developing an object-oriented file system that propagates its object-oriented structure up into the user interface.

7 Conclusions

In Choices, we have used C++ to develop new operating system mechanisms and policies based on object-oriented design. However, C++ may also be used to recode existing systems in an object-oriented manner. In this paper, we discussed the development of a class hierarchy that captures the design of two existing, well-known file systems. Although data encapsulation has been used in the design of these systems, the ease with which we have been able to design a class hierarchy to capture the similarities between the systems also reflects the adherence of the implementation of those systems to the UNIX standard file interfaces.

Our implementation contributes to our understanding of the design of class hierarchical, object-oriented systems in several ways.

- We demonstrated that system programs can be coded as efficiently in C++ as in C.

- We showed that by careful choice of the methods defined and inherited in the class hierarchy, much code and design can be reused even though the implementations may at first sight, appear to be different.
- The class hierarchy we described in this paper defines a family of file systems, and this family provides an insight into new file systems that are not only constructed as object-oriented systems but are also object-oriented in operation.
- The library of file system components that we built allows hybrid file systems to be constructed that use particular components to provide a customized file system.
- The System V and BSD implementations we built are independent of UNIX and could, potentially, be ported to many other systems in addition to Choices.

Throughout the implementation we have been impressed with the ease with which object-oriented design can be expressed in C++ code. This had many major benefits, particularly in code maintenance, debugging, and modification.

To conclude, this paper describes a complete, efficient implementation of 4.2 BSD and System V file systems as a portable package written in C++. Our next step is to build object-oriented file systems for Choices based on our experience of building UNIX-like file systems.

References

[1] Roy H. Campbell, Gary Johnston, Kevin Kenny, Gary Murakami, and Vince Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *Operating Systems Review*, 21(3):9–17, July 1987.

[2] Roy H. Campbell, Gary Johnston, Kevin Kenny, Gary Murakami, and Vince Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). In *Fourth Workshop on Real-Time Operating Systems*, pages 12–18, Cambridge, Mass., July 1987.

[3] Roy H. Campbell, Vince Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, Santa Fe, NM, November 1987.

[4] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[5] Bertrand Meyer. Reusability: the case for object-oriented design. *IEEE Software*, 50–64, March 1987.

[6] Lawrence Snyder. Using types and inheritance in object-oriented programming. *IEEE Transactions on Computers*, March 1981.

[7] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 1986.

[8] K. Thompson. Unix implementation. *Bell System Technical Journal*, 57(6):1931–1946, July 1978.

Applying Object-Oriented Design to Structured Graphics

John M. Vlissides and Mark A. Linton
Center for Integrated Systems, Room 213
Stanford University
Stanford, California 94305
E-mail: vlis@lurch.stanford.edu, linton@lurch.stanford.edu

Abstract

Structured graphics is useful for building applications that use a direct manipulation metaphor. Object-oriented languages offer inheritance, encapsulation, and runtime binding of operations to objects. Unfortunately, standard structured graphics packages do not use an object-oriented model, and object-oriented systems do not provide general-purpose structured graphics, relying instead on low-level graphics primitives. An object-oriented approach to structured graphics can give application programmers the benefits of both paradigms.

We have implemented a two-dimensional structured graphics library in C++ that presents an object-oriented model to the programmer. The **graphic** class defines a general graphical object from which all others are derived. The **picture** subclass supports hierarchical composition of graphics. Programmers can define new graphical objects either statically by subclassing or dynamically by composing instances of existing classes. We have used both this library and an earlier, non-object-oriented library to implement a MacDraw-like drawing editor. We discuss the fundamentals of the object-oriented design and its advantages based on our experiences with both libraries.

1 Introduction

Many software packages have been developed that support device-independent interactive graphics [1,3,4,6,7]. These packages provide various ways to produce graphical output. In *immediate-mode*, a graphical element such as a line appears on the screen as soon as it is specified. Several packages provide procedures for adding graphical elements to a *display list*; the elements appear on the screen after an explicit call to draw the display list. Graphical elements in the list can be stored as data or as procedural specifications. *Structured graphics* packages allow elements in a display list to be lists themselves, making it possible to compose hierarchies of graphical elements.

Application programs designed for workstations make extensive use of graphics in their user interfaces. Many programs such as drawing and schematics editors let the user manipulate graphical representations of familiar objects. Structured graphics can simplify the implementation of such applications because much of the functionality required is already implemented in the graphics package. For example, drawing editor operations for translating and scaling geometric shapes, enlarging and reducing the drawing, and storing its representation are supported by most structured graphics packages. Graphical hierarchies could be used to compose and manipulate groups of notes on staves in a music editor. A project management system could define the elements of bubble charts using graphical primitives and allow structural changes to be made interactively using display list editing operations.

However, there are drawbacks to using structured graphics. The library of procedures that comprises such packages is often large and monolithic, rich in functionality but difficult for the programmer to extend. Extensibility usually requires access to and manipulation of internal data structures, but

such access is dang\ ıs and can compromise the reliability of the system. Also, it is often difficult to edit and manipul the display list, particularly when its elements are represented procedurally, because there is no \ ıy to refer to graphic and geometric attributes directly. Editing the display list may be inefficient a: well. For example, if the display list is compiled into a more quickly executed form, then the list r ust be recompiled following editing before it can be drawn. These deficiencies make it likely that .he structure provided by the package will not map well to that required by the application, forcing the programmer to define data structures and procedures that parallel the library's.

An object-oriented design offers solutions to these problems. Intrinsic to object-oriented languages are facilities for data hiding and protection, extensibility and code sharing through inheritance, and flexibility through runtime binding of operations to objects. However, existing object-oriented programming environments [5,9] rely on immediate-mode graphics, and object-oriented user interface packages [2,11] do not support general-purpose structured graphics. Ida [15] uses an object-oriented framework that decomposes structured graphics into a set of building blocks that communicate via message passing. Ida supports high-level functionality such as scrolling, though it does not provide some graphical capabilities that structured graphics systems usually have, such as rotations and composite transformations.

We have developed a C++ [12] library of graphical objects that can be composed to form two-dimensional pictures. The library is a part of the InterViews graphical interface toolkit [8] and runs on top of the X window system [10]. Our aim was to learn how inheritance and encapsulation could be used in the design of a structured graphics library. A base class **graphic** is defined from which all other structured graphics objects are derived. We show how a hierarchy of these primitives can be composed to form more complex graphics and how features such as hit detection and incremental screen update are incorporated into the model. We also compare this library to an earlier, non-object-oriented structured graphics library implemented in Modula-2, relating experiences we had in using each library to implement a MacDraw-like drawing editor.

2 Class Organization

The graphic class and derived classes collectively form the Graphic library. The class hierarchy is shown in Figure 1. Its design was guided by the desire to share code as much as possible without compromising the logical relationships between the classes.

The derived classes define the following graphical objects:

- **Point, Line, MultiLine**: a point, a line, and a number of connected lines
- **Rect, FillRect**: open and filled rectangles
- **Ellipse, FillEllipse**: open and filled ellipses
- **Circle, FillCircle**: open and filled circles
- **Polygon, FillPolygon**: open and filled polygons
- **BSpline, ClosedBSpline, FillBSpline**: open, closed and filled B-splines
- **Label**: a string of text
- **Picture**: a collection of graphics
- **Instance**: a reference to another graphic

All graphics maintain graphics state and geometry information. Graphics state parameters are defined in separate base classes. These include **transformer** (transformation matrix), **color**, **pattern**

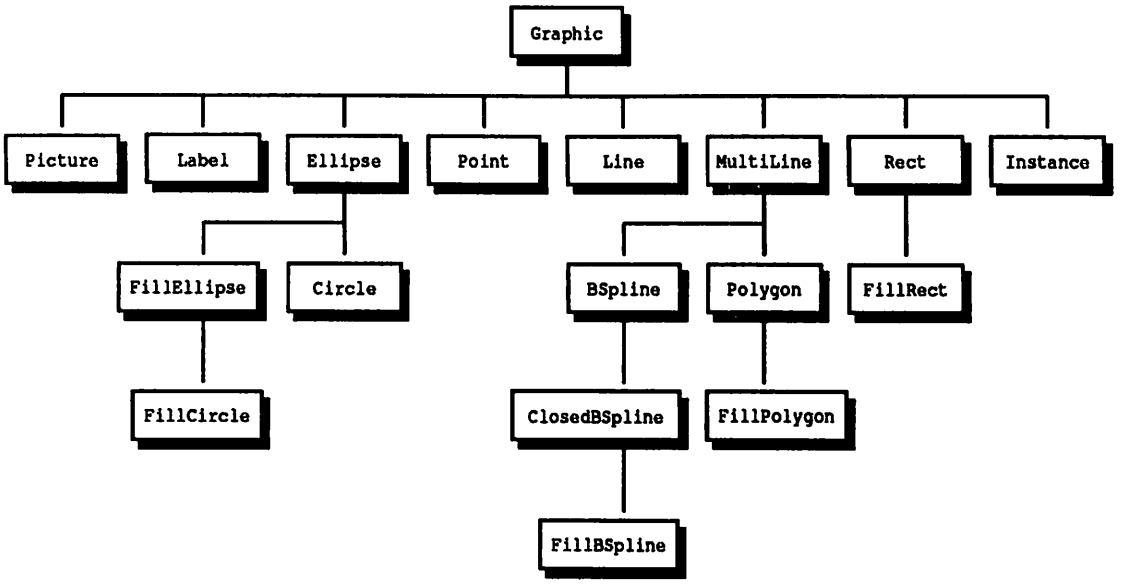


Figure 1: Graphic library class hierarchy

(for stippled area fills), **brush** (for line drawing), and **font**. Each graphics state class implements operations for defining and modifying its attributes. For example, transformers have translation, scaling, rotation, and matrix multiplication operations, and colors allow their component intensities to be varied.

A structured graphics package should be able to transfer its graphical representations to and from disk. GKS uses "metafiles" for this purpose. The files PHIGS uses are called "archives." Both packages provide procedures for saving and retrieving structures, for querying structures by name, and for deleting structures from the file.

The approach used by these packages requires the programmer to save and retrieve structures explicitly. The Graphic library uses persistent objects to automatically manage the storage of graphics. The graphic class and graphics state classes are derived from a **persistent** class that provides transparent access to objects whether they are in memory or on disk. Persistent objects are faulted in from disk when they are first referenced, and "dirty" objects are written to disk when the client program exits.

3 Graphic

The graphic base class contains a minimal set of graphics state including a transformer and foreground/background colors. Derived classes maintain additional graphics state according to their individual semantics. For example, the label class includes a font in addition to inherited state, filled objects maintain a pattern, and outline objects include a brush.

3.1 Operations

All graphics implement a set of operations defined in the base class. These include operations for

- drawing and erasing, optionally clipped to a rectangle,
- setting and retrieving graphics state values,
- translating, scaling, and rotating,

```
virtual boolean Contains(PointObj&);
virtual boolean Intersects(BoxObj&);
```

Figure 2: Interface to operations supporting hit detection

```
virtual void Draw(Canvas*);
virtual void Draw(Canvas*, Coord, Coord, Coord, Coord);
virtual void DrawClipped(Canvas*, Coord, Coord, Coord, Coord);

virtual void Erase(Canvas*);
virtual void Erase(Canvas*, Coord, Coord, Coord, Coord);
virtual void EraseClipped(Canvas*, Coord, Coord, Coord, Coord);
```

Figure 3: Interface to drawing operations

- obtaining a bounding box, and
- ascertaining whether the graphic contains a point or intersects a rectangle.

The `Contains` and `Intersects` operations are useful for hit detection. Their definitions are shown in Figure 2. `PointObj` and `BoxObj` are classes that serve as shorthand for specifying a point and a rectangular region, respectively. `Contains` can be used to detect an exact hit on a graphic; `Intersects` can be used to detect a hit within a certain tolerance.

3.2 Drawing Operations

Figure 3 lists the set of drawing and erasing operations defined on graphics. InterViews defines canvas objects and the coord type. A canvas represents a region of the display in which to draw. Canvases are rectangular and may overlap. A coord is a integer coordinate.

The graphic base class implements each erasing operation in terms of the corresponding drawing operation. An erase operation first sets the foreground color to the background color, then calls the drawing operation, and finally resets the foreground color to its original value.

The operations taking a single parameter draw and erase the graphic in its entirety. The coordinate parameters are used to specify a rectangular region. *Bounded* `Draw` and `Erase` operations use the rectangular region as a hint to the graphic's visibility. Graphics may perform optimizations based on this information. For example, because canvases do not permit drawing outside their boundaries, bounded draw and erase operations can cull parts of the graphic that fall outside the canvas.[1]

`DrawClipped` and `EraseClipped` clip during drawing or erasing. They are useful when drawing must be strictly limited to a portion of the canvas. For example, `DrawClipped` is often used to redraw portions of a graphic that had been obscured by an overlapping canvas.

[1] The bounded operation could obtain the rectangular region directly from the canvas. For generality, however, the region is specified explicitly.

4 Composite Graphics

Picture and **instance** are composite graphics. A picture composes other graphics into a single object, while an instance is a reference to another graphic. Both rely on a notion of *graphics state concatenation* to define how they are drawn.

4.1 Graphics State Concatenation

Composite graphics are like other graphics in that they maintain their own graphics state information, but they do not have their own geometric information. Composition allows us to define how the composite's state information affects its components. The graphic base class implements a mechanism for combining, or *concatenating*, graphics state information. The default behavior for concatenation is described below. Derived classes redefine the concatenation operations as needed.

Given two graphics states A and B, we can write their concatenation as $A \oplus B = C$, where C is the resultant graphics state. Concatenation associates but is not commutative; B is considered "dominant." C receives attributes defined by B. Attributes that B does not define are obtained from A. An exception is the transformation matrix; C's transformer is defined by postmultiplying A's transformer by B's. B thus dominates A in that C inherits B's attributes over A's, and C's coordinate system is defined by A's transformation with respect to B's.

A graphic might not define a particular attribute either because it is not meaningful for the graphic to do so (a filled rectangle does not maintain a font, for instance) or because the value of the attribute has been set to nil explicitly. Defined attributes propagate through successive concatenations without being overridden or modified by undefined attributes. For example, suppose graphics state A defines a font but B does not. Moreover, C maintains a font but its value has been set to nil. Then $D = A \oplus B \oplus C$ will receive A's font attribute. If A's transformer is nil but B and C's are non-nil, then D will receive a transformer that is the product of B's and C's. If $D = C \oplus A \oplus B$, then D will receive a transformer that is the product of C's and B's.

The semantics for concatenation as defined in the base class are useful for describing how composite graphics are drawn, but derived graphics can implement their own concatenation mechanism. This creates the potential for concatenation semantics that are more powerful than the default precedence relationship. For example, the concatenation operation could be redefined so that concatenating two colors would yield a third that is the sum or difference of the two. Two patterns could combine to form a pattern corresponding to an overlay of the two. This behavior could be used to define how to draw overlapping parts of a VLSI layout.

The ability to redefine concatenation semantics demonstrates how inheritance lets the programmer extend the graphics library easily. Flexibility is thus achieved without complicating or changing the library.

4.2 Picture

Pictures are the basic mechanism for building hierarchies of graphics. Each picture maintains a list of component graphics. A picture draws itself by drawing each component with a graphics state formed by concatenating the component's state with its own. Thus, operations on a picture affect all of its components as a unit. `Contains`, `Intersects`, and bounding box operations are redefined to consider all the components relative to the picture's coordinate system. The picture class defines the operations shown in Figure 4 for editing and traversing its list of components. Pictures have a notion of a "current" component, which aids in the traversal by acting as a position marker in the list of components.

Pictures also define operations for selecting graphics they compose based on position. These operations are shown in Figure 5. The ...`Containing` operations return the graphic(s) containing a point;

```
void Append(Graphic*);
void Prepend(Graphic*);
void Remove(Graphic*);

void InsertAfterCur(Graphic*);
void InsertBeforeCur(Graphic*);
void RemoveCur();
void SetCurrent(Graphic*);
Graphic* GetCurrent();

Graphic* First();
Graphic* Last();
Graphic* Next();
Graphic* Prev();
boolean IsEmpty();
boolean AtEnd()
```

Figure 4: Picture editing operations

```
Graphic* FirstGraphicContaining(PointObj&);
Graphic* FirstGraphicIntersecting(BoxObj&);
Graphic* FirstGraphicWithin(BoxObj&);

Graphic* LastGraphicContaining(PointObj&);
Graphic* LastGraphicIntersecting(BoxObj&);
Graphic* LastGraphicWithin(BoxObj&);

int GraphicsContaining(PointObj&, Graphic**&);
int GraphicsIntersecting(BoxObj&, Graphic**&);
int GraphicsWithin(BoxObj&, Graphic**&);
```

Figure 5: Picture operations for selection

...Intersecting operations return the graphic(s) intersecting a rectangle; **...Within** operations return the graphic(s) falling completely within a rectangle.

Pictures draw their components starting from the first component in the list. The **Last...** operations can be used to select the "topmost" graphic in the picture, while **First...** operations select the "bottommost." The **Graphics...** operations return as a side-effect an array of all the graphics that satisfy the hit criterion. These operations also return the size of the array.

The following example demonstrates how concatenation can be used and extended using pictures. Consider a what-you-see-is-what-you get text editor that implements paragraphs using a subclass of picture called **paragraph** and words using a subclass of label called **word**. Both pictures and labels maintain a font attribute. Thus, each word can define its own appearance, and the paragraph can override the appearance of all the words through concatenation. For instance, defining a font attribute on the paragraph would cause all words to appear in that font independent of their individual attributes.

```
void Incur(Graphic*);
void Incur(BoxObj&);
void Repair();
void Reset();
boolean Incurred();
```

Figure 6: Interface to damage class

By deriving paragraph from picture, we can change the concatenation semantics; for example, the concatenation of an italic font with a bold font could yield a bold italic font. Defining an italic font attribute on the paragraph would thus italicize the paragraph without ignoring the font of individual words. Alternatively, paragraphs could rely on words to define the concatenation semantics. Thus, instances of different word subclasses could respond differently to formatting changes within the same paragraph.

4.3 Instance

An instance is a reference to another graphic (the *target*). Graphic library instances are functionally equivalent to instances in Sketchpad [14]. The concatenation of the instance's and target's graphics states is used when the instance is drawn or erased. An instance can thus redefine any aspect of the target's graphics state, but it cannot change the target's geometric information.

Instances are useful for replicating "prototype" graphics. Once the prototype is defined, it can appear at several places in a drawing without copying. Also, structural and graphics state modifications made to the prototype will affect its instances, thus avoiding the need to change instances individually.

5 Incremental Update

Structured graphics can be used to represent and draw arbitrarily complicated images. Many images (and most interesting ones) cannot be drawn instantaneously. Incremental techniques can be used to speed the process of keeping the screen image consistent with changes in the underlying graphical structure. Such techniques will be effective if the user makes small changes most of the time, and experience with interactive graphics editors shows this to be the case.

To support incremental update, the Graphic library includes a **damage** base class. A damage object is used to keep the appearance of graphics consistent with their representation. Damage objects try to minimize the work required to redraw corrupted parts of a graphic. The base class implements a simple incremental algorithm that is effective for many applications. The algorithm can be replaced with a more sophisticated one by deriving from the base class.

5.1 Interface

The interface to the damage class appears in Figure 6. When a damage object is created it is passed a graphic (usually a picture) for which it is responsible. The `Incur` operation is called by the client program whenever the graphic is "damaged." The graphic is incrementally updated when `Repair` is called. `Reset` discards accumulated damage without updating the graphic. Clients can determine whether any damage has been incurred using the `Incurred` operation.

5.2 Implementation

The damage class implements a simple algorithm for incremental update. Each damage object maintains zero, one, or two non-overlapping rectangles. A damage object must be notified whenever the graphic's appearance changes by calling the `Incur` operation with either a region of the canvas or a graphic as a parameter. If a graphic is supplied, its bounding box determines the extent of the damaged region.

`Incur` either stores the new rectangle representing the damaged region or *merges* it with one or both of the rectangles it has stored. Merging replaces a stored rectangle with the smallest rectangle circumscribing the rectangles being merged. `Repair` calls `DrawClipped` on the graphic for each stored rectangle.

The number of rectangles maintained by damage objects is limited to two because successive increases in the number of rectangles bring diminishing returns. This is a result of the overhead associated with drawing a graphic clipped; for complicated graphics this involves significant computation. We found that the limiting value of two yielded subjectively the quickest screen update on average in an object-oriented drawing editor based on the Graphic library. Typically the user either transforms an object in place (producing a single damaged rectangle) or moves an object (producing one or two rectangles). Assuming that drawing editors represent a fair benchmark for interactive graphics applications, the two-rectangle limitation offers advantages in both performance and implementation simplicity.

6 Experience

The design of the Graphic library was based on experience with an earlier structured graphics library we implemented in Modula-2. The Modula-2 design emphasized high drawing speed over low latency. It also tried to handle incremental update completely automatically; that is, it had no operation comparable to `Incur`. The extent of damage was inferred from the operations performed on each graphical object. Though the package attempted to provide an object-oriented interface, the implementation language's lack of inheritance resulted in a monolithic library that could not be extended easily.

We have developed two versions of an object-oriented drawing editor called `idraw`, shown in Figures 7 and 8. The first version uses the Modula-2 graphic library, while the second version uses the C++ Graphic library. This gives us a good opportunity for comparing the two libraries based on actual usage.

6.1 Graphics State Propagation versus Concatenation

A difference between the Graphic and Modula-2 libraries is in the way they manage graphics state. Modula-2 graphical elements propagate their graphics state to the leaves of the graphics hierarchy as part of the modification operation. Graphic library objects defer the propagation until they are drawn, relying on the concatenation mechanism to do the job. The rationale behind propagation was to make drawing as fast as possible. It was believed that on-the-fly concatenation would slow drawing unnecessarily. Thus, as much work as possible was done before the drawing routine was called.

We realized that propagation was a mistake as we used the Modula-2 library to implement `idraw`. Propagating graphics state each time an operation is called precludes amortizing many changes over a few draws. That is, if several state-modifying operations are made before the graphic is drawn, we can avoid traversing the structure if we defer propagation to draw time, when we must traverse it anyway.

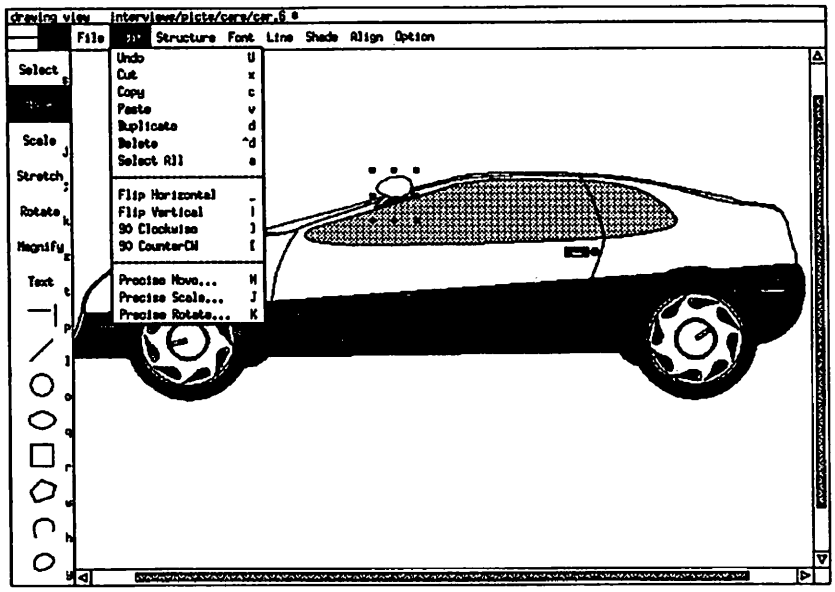


Figure 7: The idraw drawing editor, Modula-2 version

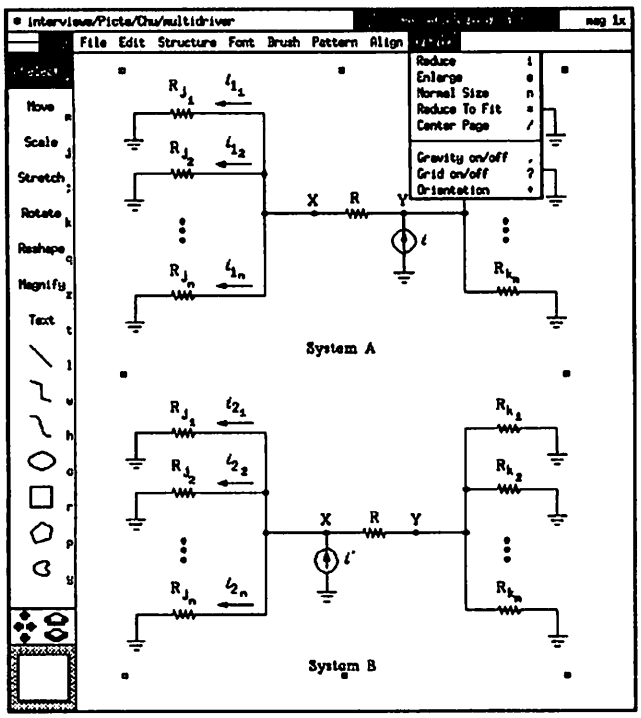


Figure 8: The idraw drawing editor, C++ version

Having made propagation an integral part of the Modula-2 library, there was no practical way for users to modify the library to use concatenation. An object-oriented design would have used inheritance to facilitate the modification of the library to use concatenation. In comparison, it would be straightforward to derive a new sort of picture and redefine its graphic state modification operations to propagate attributes immediately.

6.2 Incremental Update

The Modula-2 graphics library implemented an automatic incremental update feature. The library kept track of changes to objects by storing lists of rectangles with each object. Newly-added rectangles were merged with any rectangles in the list they intersected. The list of rectangles was ultimately limited by the object's bounding box; when a rectangle in the list became large enough to subsume the bounding box, the incremental update mechanism was disabled and the object would be drawn in its entirety.

The `Redraw` procedure was used to initiate incremental redraw of a graphical object. `Redraw` erased the regions defined by the rectangles in the object and redrew the object clipped to each rectangle. Any nested objects would be redrawn recursively.

This approach worked—the screen was never left in an inconsistent state following incremental redraw—but it did not always perform the update in an efficient way. The generality of the algorithm coupled with the lack of a way for the programmer to influence the redraw mechanism often rendered the facility useless; the programmer would bypass the mechanism and redraw damaged objects explicitly.

To illustrate, consider the case where a drawing is restructured so that an object obscured by other objects is brought to the top. A simple way to update the screen is simply to draw the object; nothing else need be redrawn. However, the incremental algorithm did not consider this optimization, and `Redraw` proceeded to redraw all the obscured objects as well.

The more serious problem arose because damaged rectangle information was *always* accumulated, since `Redraw` could be called at any time. This added overhead to every appearance-modifying operation. The overhead remained even if the programmer decided to bypass automatic redraw and perform the update manually. The addition of a `Disable` procedure that turned off rectangle accumulation complicated the use of the package and presented problems of its own: What should happen when automatic redraw is enabled again? Should old damage information be eliminated? How do we know the screen is still consistent?

The lesson we learned was that it is important not to exclude the programmer from the update process. Damage objects do not in any way interfere with the normal operation of graphics. They incur no overhead unless they are used, and they encapsulate the incremental update algorithm, making it easy to enhance or replace. In contrast, the update mechanism pervaded the older library. Damage objects give programmers the option of performing tricks of their own when updating the screen without paying for mechanisms they do not use.

6.3 Persistence

We have mixed feelings about having used persistent objects in the Graphic library. On one hand they are convenient because they free the programmer from worrying about storage. On the other hand, objects created by a program live in their own world analogous to the address space in which they were created. Thus, objects cannot communicate across program or machine boundaries easily, nor is there provision for moving objects from one world to another.

Persistent objects are useful for preserving the state of a program transparently across executions, but they are not suitable for communicating the state between processes. We expect that a later version of the Graphic library will incorporate a more conventional storage mechanism.

		Modula-2	*C++*
structured graphics library	common code	3600	3500
	incremental update	500	100
	hit detection	400	1500
	persistence	600	1300
	comments	700	300
	total lines	5800	6700
idraw	common code	13000	14000
	user interface	2000	0
	comments	1000	2000
	total lines	16000	16000

Table 1: Comparison of Modula-2 and C++ source code (in lines)

6.4 Cached Bounding Boxes

To improve performance, the more complex graphics such as multilines, polygons, splines, and pictures cache their bounding box once it is calculated. Caching can save substantial time, especially for large pictures, because the bounding box is needed whenever a graphic is drawn clipped or bounded.

The object-oriented approach makes it easy to add this optimization to classes that can use it without penalizing other classes. The graphic base class declares operations for caching, invalidating, and retrieving a bounding box. These are null operations by default; derived classes can redefine them if they use caching. Thus, individual graphics can define their own caching and invalidation criteria. Furthermore, since the base class does not allocate storage for the bounding box, no overhead is incurred on subclasses that do not require caching.

6.5 Quantitative and Qualitative Comparisons

This section presents quantitative and qualitative comparisons of the Modula-2 and C++ structured graphics libraries and versions of **idraw**. Note that any direct comparisons are necessarily crude because of differences in design criteria, in our experience level at the start of each library's implementation, and in the implementation languages themselves. Nevertheless, we offer these comparisons to add insight into the relative merits of the Modula-2 and C++ implementations.

Table 1 shows the source code sizes for both libraries and both versions of **idraw**. The library code is divided into five components: common code (that is, code that implements the same functionality in both libraries), code for incremental update, code for storing graphical objects on disk, code for hit detection, and comments. The **idraw** code is divided into common code, user interface code, and comments.

This partitioning lets us take into account different capabilities and levels of commenting when comparing code sizes. For example, the Graphic library has a general persistent object facility, whereas the Modula-2 library supports only manual read/write of graphical objects. Graphic subclasses implement fine-grain hit detection, while the Modula-2 library can detect hits only within an object's bounding box. The Modula-2 library uses a more complicated incremental update mechanism and is commented more heavily than the Graphic library. Modula-2 **idraw** implements scroll bars, pull-down menus, and rubberbands explicitly, while InterViews provides this functionality in the C++ version.

From the information in Table 1, we conclude only that the C++ and Modula-2 code is comparable in size. The amount of common code in the structured graphics libraries is about the same, and

Drawing	*Test*	*Modula-2*	*C++*
`car.6` (82 objects)	zoom #1	18	8.3
	zoom #2	12	6.3
	rotation	15	4.5
`multidriver` (361 objects)	zoom #1	24	12
	zoom #2	18	8.3
	rotation	11	6.7

Table 2: Comparison of Modula-2 and C++ `idraw` drawing performance (in seconds)

the C++ version has proportionally more code to implement added functionality. The Modula-2 `idraw` is somewhat smaller than the C++ version, taking into account that C++ `idraw` relies on InterViews to implement its user interface. However, C++ `idraw` provides more functionality, including arbitrary-level undo (versus single-level for Modula-2 `idraw`), more sophisticated text editing, and user customizability.

A possible disadvantage of an object-oriented implementation is a runtime performance penality because of overhead such as method lookup. In the implementation of C++ we used, the overhead amounts to three or four extra memory references per virtual function call [13]. To see whether this overhead has a significant impact on the performance of `idraw`, we measured how long it took each version of `idraw` to do three different operations on two different drawings, `car.6` (shown in Figure 7) and `multidriver` (shown in Figure 8). These are representative of two common types of drawings: artistic drawings with many complex, overlapping polygons and splines, and technical drawings consisting mainly of rectangles, lines, and text with little or no overlap. We timed the following operations:

1. In the "zoom #1" test, the drawing is zoomed from half size to quarter size and back. The drawing is fully visible throughout the test.

2. In "zoom #2," the drawing is zoomed from half size to full size and back. The drawing is clipped when drawn at full size so that only half is visible.

3. In "rotation," the (top-level) object in the drawing is rotated 90°.

Table 2 shows the average of ten trials for each test. The C++ version outperforms the Modula-2 version in every test. The difference in speed is greatest for the rotation test on `car.6`, but this difference is exaggerated because of a bug in the Modula-2 library's incremental update routine that caused redundant redraws of two subcomponents. In general, the Modula-2 library is handicapped by the extra traversals associated with graphic state propagation and incremental update. The results would be more comparable if the Modula-2 library were modified to use concatenation and the simpler incremental update algorithm of the damage class.

The last quantitative comparison involves the object code sizes for each library and `idraw` version. These values are shown in Table 3. The C++ sizes are larger mainly because of the added functionality of both the Graphic library and C++ `idraw`, constructor, destructor, and inline code, and the overhead associated with virtual pointer tables.

From a qualitative standpoint, the Graphic library and the corresponding version of `idraw` are both significantly better structured, more understandable, and "cleaner" overall than their Modula-2 counterparts. One could argue that the lessons learned in the Modula-2 implementation efforts led to superior C++ versions. However, the versions of `idraw` were developed by two different people. In fact, the Modula-2 version was its author's second attempt at a drawing editor, while the C++ version was its author's first attempt. The object-oriented paradigm simply invites good program

	Modula-2	*C++*
structured graphics library	40	110
`idraw`	130	280

Table 3: Comparison of Modula-2 and C++ object code sizes (in kilobytes)

structuring through inheritance, encapsulation, and late binding, all of which promote modularity and flexibility.

7 Conclusion

A striking aspect of graphics packages such as CORE, GKS, and PHIGS is their size and complexity. These packages are intended as standards that provide machine independence, extensive functionality, and generality, and they largely succeed in these respects. However, all reflect their procedural implementation in their interface. Programmers cannot extend primitives through inheritance to modify their semantics. The result is a substantial complexity penalty for every increase in flexibility.

For example, some packages bind graphics state attributes statically to graphical objects when the objects are created. Others provide a simple form of state inheritance by allowing graphics to reference other graphics in a manner similar to instances in the Graphic library. These facilities are significantly less flexible than the graphics state concatenation mechanism, the semantics of which can be changed on a per-class basis. In an object-oriented package, generality can be achieved through class inheritance instead of supporting a broad range of behaviors explicitly.

Another advantage of the object-oriented approach is the ability to treat graphical objects generically, relying on the runtime system to determine the correct method for a particular object. The `virtual` mechanism accomplishes this in C++. Thus, functionality such as hit detection can be implemented in a simple way without identifying objects with element pointers and labels. Furthermore, escape mechanisms for exploiting special hardware facilities are unnecessary; subclasses can be derived that reimplement key operations such as `Draw` to take advantage of unique capabilities.

In our experience, structured graphics is useful for applications that allow the user to manipulate graphical objects interactively. Structured graphics is less useful for implementing the appearance of the user interface. It is unnecessary to define scroll bars, menus, and buttons using structured graphics because they are simple to draw procedurally and their structure rarely changes. Thus, structured graphics is not a replacement for immediate-mode graphics.

We are interested in using the Graphic library for animating graphics. Structured graphics is appropriate for animation if the hardware is fast enough to support it. Also, the current implementation does not provide three-dimensional capabilities. Extending the library to support three dimensional graphics would require significant additions to base class functionality, for example, to incorporate operations governing lighting models and point of view, three-dimensional analogs of `Contains` and `Intersects`, and additional information when clipping.

Of more immediate interest is the introduction of version 2.0 of C++ [13] with multiple inheritance, among other enhancements. Though single inheritance is very useful, it often forces the programmer to derive from one of two equally attractive classes. This limits the applicability of predefined classes, often making it necessary to duplicate code. For example, there is no way to derive a graphic that is both a circle and a picture; one must derive from one or the other and reimplement the functionality of the class that was excluded.

The availability of multiple inheritance will undoubtedly change the class hierarchy shown in Figure 1. Classes such as **filled** and **open** could be defined to simplify the relationships between filled and non-filled graphics, which are currently derived as they are to maximize code sharing. Persis-

tence could be implemented as a separate class from which to inherit. Thus, non-persistent classes can avoid the small space overhead caused by deriving graphic from a persistent class.

Acknowledgments

This work was supported by the Quantum project through a gift from Digital Equipment Corporation. John Interrante implemented the C++ version of **idraw**. Paul Calder and Craig Dunwoody provided helpful comments on earlier drafts of this paper.

References

[1] *IRIS User's Guide.* Silicon Graphics, Inc., 1984.

[2] P.S. Barth. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[3] P. Bono et al. GKS: The first graphics standard. *IEEE Computer Graphics & Applications*, 2(5):9–23, July 1982.

[4] Status report of the graphics standards planning committee of ACM/SIGGRAPH. *Computer Graphics*, 13(3), Fall 1979.

[5] Adele J. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA, 1984.

[6] W.T. Hewitt. Programmers Hierarchical Interactive Graphics System (PHIGS). In G. Enderle et al., editors, *Advances in Computer Graphics I*, Springer-Verlag, 1986.

[7] Keith A. Lantz and William Nowicki. Structured graphics for distributed systems. *ACM Transactions on Graphics*, 3(1):23–51, January 1984.

[8] Mark A. Linton, Paul R. Calder, and John M. Vlissides. *InterViews: A C++ Graphical Interface Toolkit.* Technical Report CSL–TR–88–358, Stanford University, Stanford, CA 94305–2192, July 1988.

[9] Patrick D. O'Brien, D.C. Halbert, and Mike F. Kilian. The Trellis programming environment. In *ACM OOPSLA '87 Conference Proceedings*, pages 91–102, Orlando, FL, October 1987.

[10] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[11] Kurt J. Schmucker. *Object-Oriented Programming for the Macintosh*, pages 83–270. Hayden, Hasbrouck Heights, NJ, 1986.

[12] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1986.

[13] Bjarne Stroustrup. The evolution of C++: 1985 to 1987. In *Proceedings of the USENIX C++ Workshop*, pages 1–21, Santa Fe, NM, November 1987.

[14] I.E. Sutherland. *Sketchpad: A Man-Machine Graphical Communication System.* PhD thesis, MIT, 1963.

[15] Robert L. Young. An object-oriented framework for interactive data graphics. In *ACM OOPSLA '87 Conference Proceedings*, pages 78–90, Orlando, FL, October 1987.

A C++ Interpreter for Scheme*

Vincent F. Russo and Simon M. Kaplan

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801

Abstract

This paper reports on a project to port an interpreter written in C for a subset of the Scheme language, to C++. The primary advantage of such a migration is the greatly improved internal structure of the interpreter, increasing modularity and decreasing the effort required to maintain and extend it. A secondary advantage is that the C++ interpreter is somewhat faster. We briefly overview Scheme, discuss the implementation of the two interpreters, and give performance data. We compare the two systems from the ease of maintenance, extension, and performance viewpoints.

1 Introduction

Scheme [7] is a dialect of Lisp, with static scoping, full function abstraction, and the ability to treat all objects in the language (including functions and continuations) as first-class objects. These features, along with its simple syntax, make it an excellent language both for teaching and research. As part of the Garp [6] [5] project, we are investigating extensions to Scheme that support explicit parallelism. Implementation of such extensions require that we modify an existing Scheme interpreter. Scheme interpreters are usually written in C [1]. Because of C's poor abstraction support, the resulting code is usually difficult to modify and maintain.

To try to solve these problems, we have reimplemented the SIOD Scheme interpreter [3] in C++. This paper reports on our experience with the reimplementation. For the purpose of this paper, we are not concerned with Scheme *per se*; rather we view Scheme as an application, reimplemented in C++, and comment on the advantages of migrating an interpreter from C to C++.

*This work supported in part by NSF grants CCR-88-09479 and CISE-1-5-30035 and grants from AT&T.

A Scheme interpreter is an interesting candidate for such a migration because of its internal structure. The interpreter loops continuously through the following three phases:

- *Read.* Issue a prompt. Read an expression from the terminal.
- *Eval.* Evaluate the expression just read. This may mutate the state of the interpreter by introducing new or modifying existing bindings.
- *Print.* Print out the result of the evaluation.

The read phase produces a data structure representing the expression read; the evaluator then traverses this data structure, and produces a new structure representing the result. This is in turn traversed by the printer.

In a traditional C implementation, the cornerstones of the implementation are a discriminated union representing components of expressions and a large evaluation procedure which interprets instances of this structure depending on the tag value of each instance, i.e. the procedure body is essentially a giant switch statement. This means that the logical structure of the interpreter is lost, and one ends up with two large objects, the union and the code for the evaluator, both of which must be consistent during modification or extension of the code. This makes the interpreter difficult to write, maintain and understand. By switching to an object-oriented implementation in C++ we *distribute* both the union and the evaluator over the classes representing the elements of the union. This:

- *makes the code modular.* Classes are orthogonal, and the interpreter can be extended by adding new (sub)classes.
- *reduces maintenance effort.* A particular part of the interpreter (for example closures or cons nodes) can be understood independently of other parts.
- *speeds up the interpreter.* We were not looking for an efficiency improvement, but found one anyway. This has to do with the change from a discriminated union and switched evaluator to an explicit object-based approach and using C++ virtual functions to implicitly store the type information.

We should emphasize at this point that the performance figures reflect a simple change from a C to a C++ based implementation. There were some poor design decisions in the original C implementation which affect performance. To give a fair comparison, we have not corrected these defects in our C++ implementation; therefore our speedup reflects differences between C++ and C, not "superior hacking skill". Of course, correcting the design defects will result in an even greater speed improvement.

The remainder of the paper is structured as follows. Section 2 discusses the Scheme language, and identifies the subset which we have implemented. Section 3 first overviews how a Scheme implementation is structured, then discusses the C and C++ implementations chosen, and then compares the implementations. Section 5 gives comparative performance figures and discusses them. We end by offering conclusions and directions for future work.

2 Scheme

This section of the paper discusses the Scheme language, and the subset of Scheme implemented in the SIOD interpreter.

2.1 Scheme Overview

Scheme is a dialect of Lisp. Its full, formal definition can be found in [7]. An excellent introduction is [2]. The salient features of the language are:

- Few syntactic features. These include binding (`define` and `let`), assignment (`set!`) conditional (`if` and `cond`), function abstraction (`lambda`) and continuation access (`call/cc`) constructs.
- Library of standard functions. These include mathematical, logical, symbol, string and list operators. There is also a collection of array manipulation operators. Note that in Scheme operators are *not* a part of the syntax of the language, but part of a standard library.
- All objects in the language are *first-class*, including procedures and continuations. This means that these may be passed as arguments to or returned from other procedures.
- Arithmetic is performed using infinite-precision numbers represented according to the rules in [7], not in the native machine floating point representation.
- Quasiquotation. This is a way of elegantly constructing data objects where some parts are literals and some are results of function applications. It is most often used in macro definitions and syntactic extensions.

Following are three sample Scheme function declarations. The performance of them is analyzed later in the paper. The forms shown bind a name (for example `fact`) to a value - the procedure, declared by a `lambda` expression. Note the prefix polish notation for function (including predefined operator) application. The form of an `if` expression allows three expressions: condition, then and else parts. Parentheses surround function applications and the `lambda`-expressions.

The first computes the fibonacci number of its argument:

```
(define fib (lambda (x)
        (if (< x 2) x (+ (fib (- x 1)) (fib (- x 2))))))
```

The second and third compute the factorial and fictorial of their arguments.

```
(define fact (lambda (x) (if (= x 0) 1 (* x (fact (- x 1))))))
(define fict (lambda (x) (if (= x 0) 1 (* 1 (fict (- x 1))))))
```

Fictorial is like factorial, but always multiplies by one in order to prevent numeric overflow for large arguments. We will discuss these examples in more detail below.

2.2 SIOD Subset of Scheme

SIOD is a simple experimental interpreter and as such does not implement the entire language. Significant exclusions and differences are:

- Omission of quasiquotation and the continuation construct.
- Use of native machine floating point arithmetic instead of the formal arithmetic model.
- Omission of array types.

We should note that we do want to extend the interpreter to handle the full language, and that in our C++ version this is easy to do, given the natural abstraction barriers induced by the object-oriented paradigm.

3 Interpreter Structures

This section of the paper describes the implementation of the interpreters. It is broken into four subsections. We first discuss an abstract model of the interpreter, then its realization in SIOD and the C++ interpreter, and finally compare the two implementations from a software engineering viewpoint. Discussion of performance implications is deferred to the following section.

3.1 An Abstract View

The interpreter consists of three major components: the *reader*, which reads an expression and builds an internal representation for it; the *evaluator*, which traverses this internal structure and "interprets" it, possibly modifying the state of the interpreter in the process; and the *printer*, which displays the result of the evaluation.

The building block of any dialect of LISP is an *S-expression.* Numbers, symbols and lists (of numbers, symbols and other lists) are all S-expressions. From the viewpoint of the interpreter there are four classes of data object:

- Numbers.
- Symbols.
- Null. A literal constant analogous to a `NULL` pointer in C.
- Pairs. A pair is the building-block for constructing lists of data objects; the first element, or `car` is a data object, and the second element, or `cdr` is also a data object. Any complex structure can be constructed out of collections of such pairs. Pairs are sometimes referred to as `cons` nodes.

The reader builds an internal tree representation of an S-expression and passes it on to the evaluator.

The evaluator traverses the tree and takes appropriate action depending on the contents of each node of the tree. It has two arguments: the root of the (sub)tree being traversed (which allows recursive traversing), and an environment pointer that points to a structure holding bindings of variables to values. An environment is created for each scope (`lambda`-expression or `let`-expression). Each environment points to its enclosing scope, creating a tree of environments. The environment pointer points to some element of a tree that can be traversed to identify variables visible in that environment.

If the node is a number node, the evaluator just returns the number stored there.

If the node is a symbol node, the evaluator looks up an identifier using the environment pointer, and returns the value bound to the identifier. If no binding is found the evaluation fails.

If, however, the node is a **cons** node, then the following action depends on the **car** field of the node. We evaluate the contents of the **car** field, and then switch:

- If the **car** field is a reserved symbol (eg `if`, `define`, `set!`, `lambda`, etc) then we interpret that form. For example, for an `if` this would mean evaluating the **car** of the **cdr** field (the location of the *condition* part), and if that is true then evaluating the **car** of the **cdr** of the **cdr** (the location of the *then* part) and otherwise the *else* part.

  For a `lambda`-expression, its evaluation results in the construction of a *closure*, which consists of the code for the expression (the arguments and body of the procedure) together with a pointer to the environment in which the expression is being evaluated so that the evaluator can resolve free variables correctly according to the rules of static scope. The closure is returned as the result of the evaluator for the `lambda`-expression.

- If it is not a reserved symbol, then the expression must be a function application and the result of the evaluation must have been a closure or a primitive operator.

  Each argument is evaluated and its result stored on a runtime stack. A new environment is created, and linked to the environment pointer in the closure for resolution of free variables. The formal parameters are bound to the argument results on the runtime stack in the new environment just created and the function body is evaluated in this environment.

  The application of most functions requires the allocation of more space for the results of the applications. For example, the result of the application of the primitive procedure **cons** is a new pair holding the two arguments to the function.

- There are special cases, especially concerning quotation and quasiquotation, that complicate this model and are beyond the scope of the paper.

As the evaluator executes, it destroys references to existing data and creates new data structures. Another part of the evaluation process is to *garbage collect* space that is no longer accessible when memory fills up.

Once execution of the evaluator is complete, the printer traverses the structure representing the result of the computation and displays the result in human-readable form.

The following sections discuss the realization of this model in the C and C++ interpreters, highlighting their differences and the advantages of the object-oriented approach.

3.2 SIOD Implementation Details

In SIOD a discriminated union, named *obj*, is used to represent all the data objects. This union contains an enumerated type field indicating whether the object in question is a `cons`, `number`, `symbol`, `primitive` or a `closure` node. If it is a `cons` node, the union contains two pointer to the `car` and the `cdr` of the list. If it is a `number`, the union stores the value of the number in a C `double`. If it is a `symbol`, the union stores a C `char` pointer to the symbol's string representation and a pointer to the symbols value. If it is a primitive, the union stores a pointer to the name of the primitive (for debugging and printing purposes), and a pointer to the C builtin function that is the body of the primitive. Finally if it is a `closure`, the union contains pointers to the environment of the `closure` and to the body of the code. The body of the code is stored just like any other S-expression. The reason for using a single data structure is mainly to simplify garbage collection by making all elements in the heap the same size.

3.2.1 The Reader

In SIOD, the reader scans individual tokens from the input stream and builds up the internal representation of the input S-expressions, numbers and symbols as trees of *obj*s. The reader sets the type code for each *obj* that it creates. Symbol names are added to the global environment and their values are set to NIL. Numbers are parsed and stored internally as C `double`s. All other expressions are stored internally in the forms described above.

3.2.2 The Evaluator

The evaluator is the most complex phase of the interpreter. Overall it inspects the type code of each *obj* passed into it, evaluates the *obj* (possibly involving recursive calls to itself), and then returns the value in the form of another *obj*. For all arguments except symbols and cons nodes, the evaluator simply returns the argument. If the argument is a symbol, the symbol is looked up in the environment and its value is returned. If the argument is a cons node the car is split off and evaluated in the environment. Usually it will be a symbol and its value will resolve to a primitive or a closure. Other values are errors. If the symbol resolves to a primitive, a builtin C function is called to evaluate the primitive and return a result in the form of another *obj*. This result is then returned from the evaluator. If the symbol resolves to a closure, the environment is extended to include the closures environment, and the code is evaluated in the new environment by calling the evaluator tail recursively.

3.2.3 The Printer

Of all the phases of the interpreter, the printer is the simplest. Cons nodes are printed as regular LISP S-expressions, numbers are printed as their values, symbols are printed as their string representations, primitives print as #<SUBR *primname*> and closures print as #<CLOSURE *body environment*> (where both *body* and *environment* are also S-expressions).

3.2.4 Miscellaneous

The remaining portions of the interpreter consist of the code to evaluate all the Scheme primitives implemented, the initial startup code which constructs the initial global environment containing the names of all the primitives, the top level read-eval-print loop and the garbage collector.

3.3 C++ Reimplementation Details

When recoding the SIOD interpreter in C++, every attempt was made to keep the essential algorithms of the C and C++ implementation identical. The C++ implementation attempts to replace all the uses of the type information and case statements in the interpreter with C++ virtual functions. For example, in the C implementation, the printer is coded with the following skeleton.

```
print( o )
struct obj * o;
{
        switch( o->type ) {
           case ConsNode:
                ...
           case Symbol:
                printf( "%s", ((Symbol *) o)->name );
                break;
           case Number:
                printf( "%g", ((Number *) o)->value );
           case Primitive:
                ...
           case Closure:
                ...
        }
}
```

The more complex cases are elided to save space. Given a pointer to an object o, the **print** routine would be invoked simply as:

```
print( &o );
```

In the C++ version of the interpreter this can be restructured by creating class Object (much like the obj struct in the C version) and subclassing it to represent the various members of the discriminated union in SIOD. Printing an object would be performed by the simple syntactic transformation of the above to:

```
o->print();
```

But, by making `print` a virtual function in class Object, each subclass of object (i.e. Symbol, Number, ConsNode, etc) can redefine how to print its value.

For example:

```
Symbol::print()
{
        printf( "%s", name );
}
```

or

```
Number::print()
{
        printf( "%g", value );
}
```

The obvious advantage to this methodology is the removal of the need for explicit type information to be kept with each Object and the grouping of all the code related to a particular subclass of object together. The net effect is that the C++ complier becomes responsible for keeping the type field (in the form of the pointer to the object virtual function table) and for selecting which routine to execute for each action.

3.3.1 Major Classes in the Interpreter

The major class in the C++ implementation of the interpreter is the class *Object*. Object defines virtual functions to implement all the major portions of the read, eval and print phases of the interpreter. Object is subclassed into 5 subclasses. The class *Number* represents numbers, the class *Symbol* represents symbols, the class *ConsNode* represents pairs, the class *Primitive* represents builtin Scheme primitives, and finally, the class *Closure* represents Scheme closures.

In Scheme, any operation can be applied to any data object. For this reason all interpreter operations are defined as virtual functions which perform no function, or are errors, by class Object and redefined, by the subclasses in which they make sense, to do the proper actions. For example, class Object defines the `car` and `cdr` operations to return an empty pair. The ConsNode subclass redefines them to return, respectively, the `car` or `cdr` of the data element. The list below names each of the virtual functions defined by class Object and gives a brief description.

- `Object * print()` – Print the value of the object in a human readable form.

- `Object * gcRelocate()` - Relocate the object to a new heap (used in garbage collection).
- `void gcRelocateComponents()` - Relocate any objects the current object references to a new heap by invoking their `gcRelocate()` function.
- `Object * eval( Object * environment )` - Evaluate the object in the environment passed as an argument.
- `Object * car()` - Return the LISP `car` of the object.
- `Object * cdr()` - Return the LISP `cdr` of the object.
- `Object * setcar( Object * newcar )` - Set the `car` of the object to `newcar`.
- `Object * setcdr( Object * newcdr )` - Set the `cdr` of the object to `newcdr`.
- `Object * pairp()` - Test if the object is a ConsNode or not.
- `double value()` - Return the numeric value of the object in a C `double`.
- `Object * symbolp()` - Test is the object is a Symbol or not.
- `Object * symbolBound( obj * environment )` - If the object is a Symbol, test whether it is bound in the environment.
- `Object * symbolValue( obj * environment )` - If the object is a Symbol, return its value in the current environment.
- `Object * nullp()` - Test whether the object is the null pair.

4 Comparison of the Implementations

The original interpreter contained no data abstractions per se; rather, all data types were implemented as instances of a discriminated union with an enumerated type field as discriminant. Most internal functions in the interpreter consist of a large case statement to interpret the various types. Thus there was no modularity in type handling. Adding a new type is difficult, and requires modifying many different parts of the code. In the re-implementation of the interpreter, abstractions are introduced and implemented by C++ classes that are subclassed from a new "Object" class. By using virtual functions defined for "Object" to implement the primitive operations on each type, all explicit type information was removed from the interpreter's implementation. This eliminated the case statements, and simplifies adding new abstractions. The implementation demonstrates the expressive power of class hierarchies and object-oriented programming.

As an example, consider an attempt to modify both interpreters to extend *Number* to *Integer* and *Real*. In the original C interpreter, almost every part of the interpreter would need to be explicitly rewritten. New cases would have to be added to the

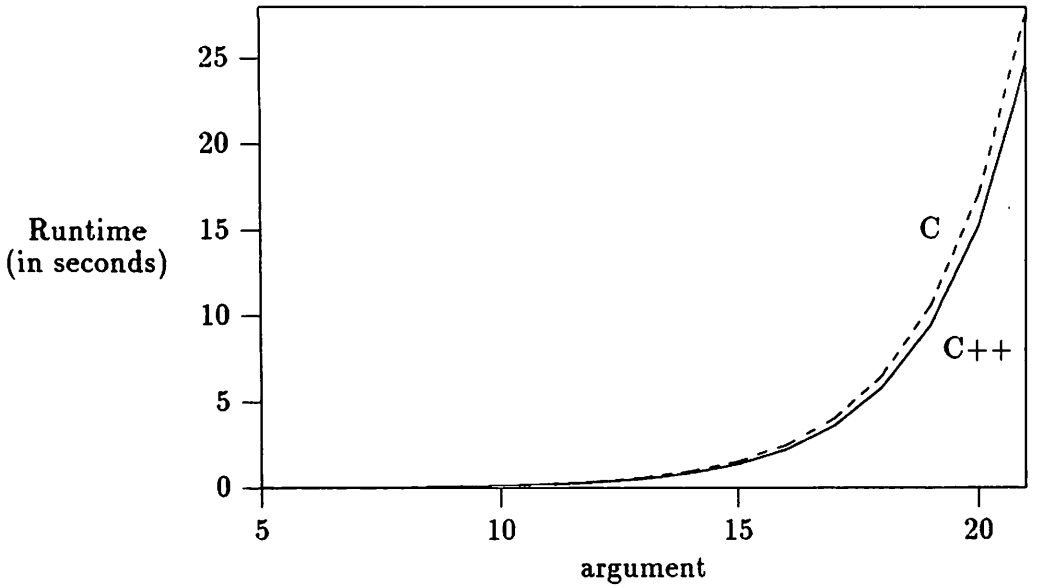


Figure 1: `fib` function

case statements where ever *Number* was used. In the C++ version, we would simply subclass *Number* into *Integer* and *Real*, and redefine the relevant operations on the subclasses.

The print routines would become:

```
class Integer : public Number {
        void print() { printf( "%d", value ); }
        ...
}

class Real : public Number {
        void print() { printf( "%g", value ); }
        ...
}
```

5 Performance

The three scheme functions shown earlier were used to compare the two interpreters. Data were gathered on an Encore Multimax using NS32332 processors running UNIX in single user mode and averaged over 10 separate runs.

Figure 1 shows the runtime of `fib` versus its argument. As can be seen from the graph, the C++ interpreter outperforms the C version. Figures 2, and 3 show the relative performance of the two interpreters on the `fact` and `fict` functions. Both functions show a linear relation between their arguments and their runtimes. The `fict` function multiplies repeatedly by 1 rather than by `x`, to get the effect of the multiplications without overflows. This allows it to run for a much larger argument

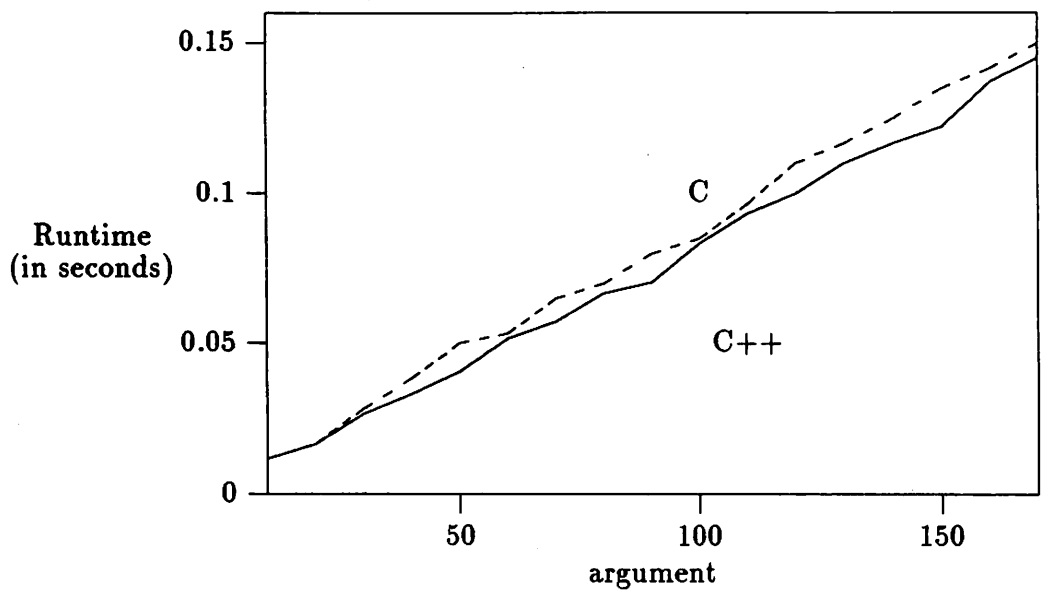


Figure 2: `fact` function

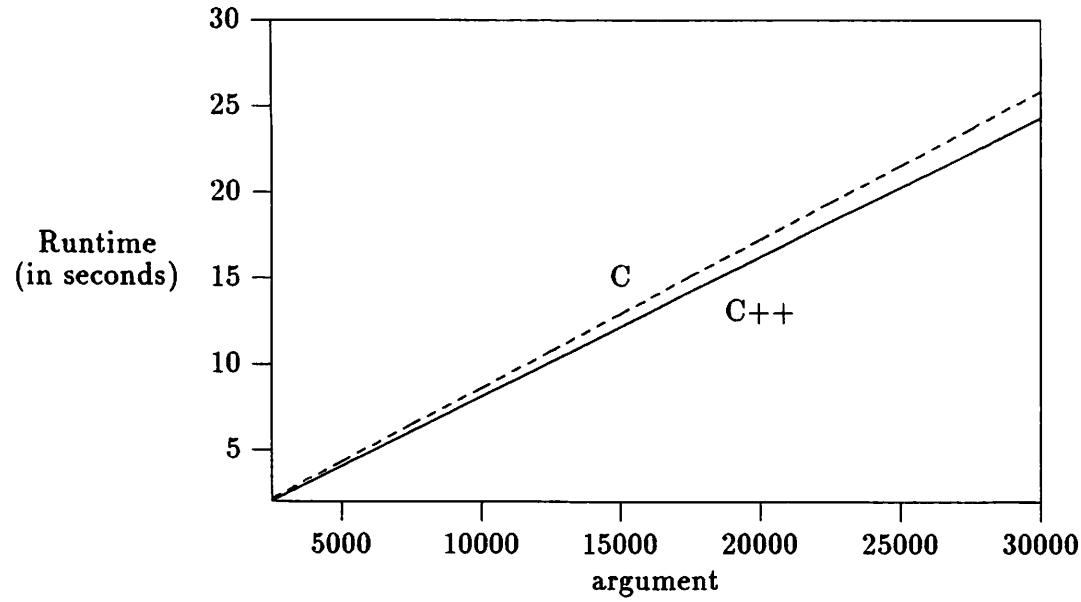


Figure 3: `fict` function

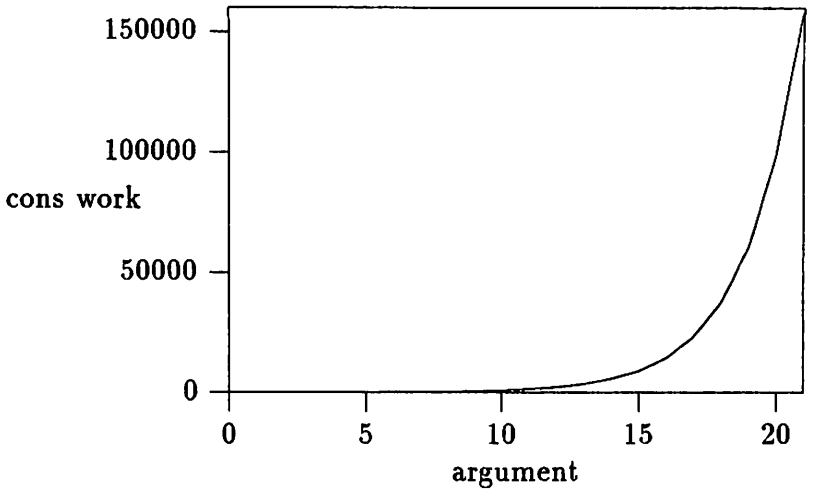


Figure 4: `fib` cons work

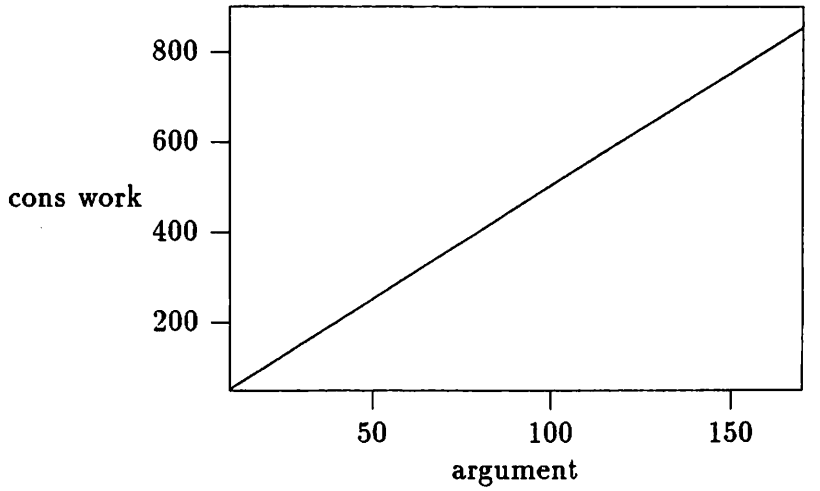


Figure 5: `fact` cons work

value. The performance data show that the larger the argument the greater the speed difference between the two interpreters.

5.1 Analysis of Performance Data

The C++ version is consistently faster than the C version of the interpreter. This can most likely be credited t· the use of virtual functions. There invocation is faster than checking an explicit ty] code and executing a case statement. This is encouraging because it indicates th: there is actually a performance bonus to be gained by properly structuring code acc ·rding to accepted object-oriented program development guidelines [4].

In all three of the test cases, the amount of *cons work* (a rough measure of the number of data structure traversals and evaluations performed by the interpreter), rises proportionally to the argument (see figures 4, 5 and 6), but the runtime for the C version rises faster than that of the C++ version.

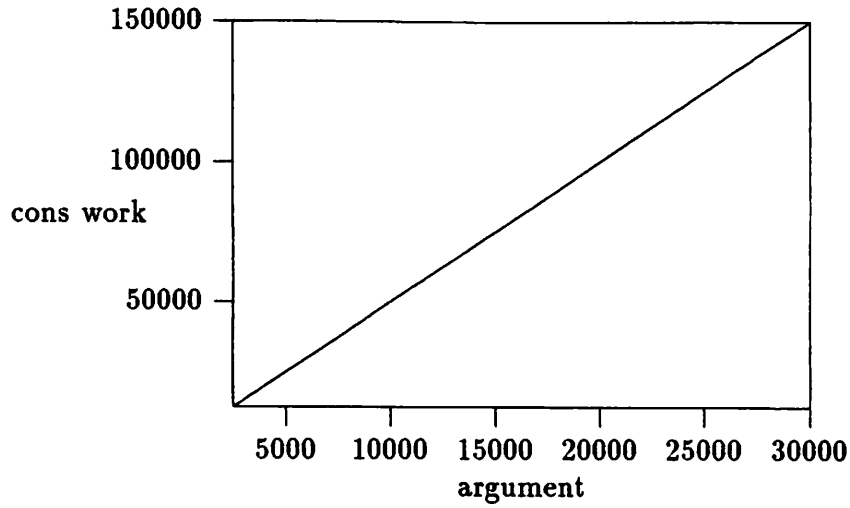


Figure 6: `fict` cons work

6 Conclusions

We have reported on the reimplementation of an interpreter for Scheme in C++. We found that the C++ version is:

- *More modular.* The structure of the code is improved because, rather than having a monolithic discriminated union and equally large switch to evaluate instances of the union, we use a collection of small, orthogonal objects.
- *More easily extended.* Because each type in the system is implemented as an independent (sub)class, code modification during extension to the system is minimized.
- *More reliable.* Better code structure leads to a more robust implementation.
- *Faster.* Simply restructuring a traditional C program into a class hierarchy, and using an efficient object-oriented language for the implementation eliminates costly switches based on discriminants and speeds up the interpreter.

Object-oriented programming is well-known for its improvement to program structure, but programming mythology suggests that there is a performance price to be paid for this improvement. Our results debunk this myth.

Our performance figures are not the last word on the speedup that can be gained from a change to the object-oriented approach. There are several design flaws in the initial C implementation which were highlighted by the change to C++; we have ignored fixing these in this paper in order to get a fair comparison of performance data, but we are currently working on extensions to the interpreter to correct these flaws.

Acknowledgements

We are deeply indebted to George Carrette for his effort in implementing the SIOD Scheme interpreter. We would also like to thank Roy Campbell for his comments.

References

[1] The mit c scheme interpreter. *MIT public domain software*, 1987.

[2] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[3] George J. Carrette. The siod (scheme in one defun) scheme interpreter. *comp.source.unix newsgroup*, 1988.

[4] Ralph E. Johnson and Brian Foote. Designing reusable classes. *The Journal of Object-Oriented Programming*, 1(2), 1988.

[5] Simon M. Kaplan and Roy H. Campbell. Designing and prototyping in grads. In *Proceedings IEEE Software Engineering Conference*, Liverpool, England, July 1988.

[6] Simon M. Kaplan and Gail E. Kaiser. Garp: graph abstractions for concurrent programming. In *ESOP '88 Proceedings*, March 1988.

[7] J. Rees and W. Clinger (Editors). Revised (3) report on the algorithmic language scheme. *Sigplan Notices*, 21(12):37–79, December 1986.

GPIO:
Extensible Objects for Electronic Design Tools

Roger Scott
Prakash Reddy
Russel Edwards*
David Campbell†
Data General Corp.
433 N. Mathilda Ave.
Sunnyvale, CA 94086
roger@panther.sv.dg.com

August 31, 1988

Abstract

GPIO is a library of object-oriented data structures developed at Data General for use in building internal Computer Aided Design / Computer Aided Engineering (CAD/CAE) tools. The goal of GPIO was to provide a software platform on which to build these tools that included: efficient storage and retrieval of design objects into/from a filing system; the ability for application programs to extend the data in the objects without modifying GPIO itself; and the ability for application programs to determine the implementation of GPIO's most important data structuring entity - the **Collection**. GPIO is implemented entirely in C++ [1] and makes extensive use of virtually all C++ features including inheritance, virtual functions, data hiding, inline functions, constructors and destructors, overloading, pointers to (virtual) member functions, static members, and global objects with constructors and destructors.

This paper will discuss the system of classes that comprise GPIO, the techniques GPIO uses for storing and retrieving structured data, the techniques used to allow applications to extend GPIO objects, and the details of the collection abstraction used within GPIO. Along the way, applications of various C++ features will be noted, ongoing open issues will be discussed, and the benefits of hindsight will be applied. A number of perceived C++ deficiencies will also be discussed, including the need for 'parameterized' types, multiple inheritance, and class 'meta objects'. Developers have built several application programs using GPIO and this paper will describe their experiences using this sophisticated class library.

1 Introduction

GPIO was originally conceived as a means of assuring that different application programs in our internal CAD/CAE system would agree on both the format and the meaning of the files

*now affiliated with Valid Logic Systems, Inc. San Jose, CA
†now affiliated with Valid Logic Systems, Inc. San Jose, CA

stored in our design database. The problem of sharing data among an assortment of CAD tools is among the most important that CAD system builders face, and is not new. Traditional approaches to this problem fall into two categories. The first is 'no solution', where each application independently locates, reads, parses, and interprets the data files. With this approach only discipline and luck provide coherence. The second is a procedural interface to the database, often in the form of an 'access library'. This scheme provides much greater assurance of coherency, but still requires that each application design data structures from scratch and then manually bridge the gap between those data structures and the procedural interface. GPIO is an attempt to apply the full power of Object Oriented Programming [2] to the CAD database problem. The result is the logical next step from procedural interfaces: a library of object oriented data structures for use in building CAD/CAE applications—an *object interface.*

The design of the file formats and the design of the data structure abstraction were very closely tied. Both follow a very simple model of fixed form aggregates, or structures, and collections. The grammar of the file format is completely predictive, so, given an object as a starting point, generating the file (when writing) and parsing the file (when reading) are both trivial tasks. In fact, throughout most of GPIO reading and writing are unified operations. Because the structures are fixed form and the collections are homogeneous, we have managed to keep 'synchronization' overhead to a minimum in GPIO files.

Because different applications place different requirements on their data structures, typically choosing widely spaced points on the time-space tradeoff continuum, we had to defer to GPIO's 'customers' as many data structure design decisions as possible if it was to be widely accepted. We initially adopted a very conservative design style which called for the GPIO data structures to closely mirror the file formats, producing, in effect, an unannotated parse tree. The intention was that any annotation of the the tree would be done by each application as it saw fit. We recognized that this might turn out to be lower than the lowest common denominator of the applications, but we reasoned that it would be easier to build up GPIO over time to meet common needs than it would be to remove excesses that resulted in unacceptable overhead for some applications. Over time we have recognized additional common needs and incorporated these in GPIO; this paper describes its current state.

2 The Class System

GPIO is actually an application-specific class library built on top of a general purpose class library. Because the general library is used extensively in the implementation of GPIO, it will be described first.

2.1 DG Library

The DG C++ library is an assortment of general purpose classes that provides convenient ready-made solutions to many common programming problems. It currently includes a rudimentary string package, a sophisticated hash table/set package, two sizes of bit array based sets-of-integers, an exception handling system, and an easy-to-use symbol table class.

2.1.1 String

Our **String** class's main purpose is to eliminate one of the most common sources of errors in C programs—string storage mismanagement. By starting with a very conservative 'allocate space and copy' strategy we have nearly eliminated problems with shared strings and permanent data structures pointing to temporary strings or literal strings. The inclusion of appropriate constructors and type conversion operators makes **String** objects interchangeable with '`const char *`'s.

2.1.2 Set

Our **Set** class provides a very general set of pointers to objects capability. An object can simultaneously be an element of multiple **Sets**, and the same object can participate in more than one *kind* of set. **Sets** support a full compliment of set-theoretical operations, plus the ability to look up an element given a pointer to an object.

Class **Set** is implemented with a pointer to a (variable sized) **HashTable** object and a counter that holds the set's cardinality. Class **HashTable** contains an array of pointers to **HashTableElement** objects. Class **HashTableElement** contains nothing but a trivial virtual destructor[1]. `HashTable::Slot(HashTableElement *, PHashMember, PEqMember)` is the core of the hashing and set implementation. It returns the address of the slot in a hash table that either does contain or would contain a (pointer to a) given **HashTableElement**. The key to the generality of this method[2] is found in the second and third arguments, which are method variables (pointers to member functions of **HashTableElement**) that are used to compute a hash function of an object and to compare two objects for 'equality'. In practice, these are never actually members of **HashTableElement**, but rather members of classes derived from it.

To date the only direct user of class **HashTable / HashTableElement** is class **Set / SetElement**. **SetElement** is derived from **HashTableElement**, and adds virtual functions `Hash()` and `Eq()`, whose default implementations are based solely on object addresses. Class **Set** includes two virtual member functions, `HashFct()` and `EqFct()`, that return method variables that are supplied to `HashTable::Slot(...)` by `Set::Slot(...)`. The default implementation of `Set::HashFct()` [`Set::EqFct()`] is to return `SetElement::Hash` [`SetElement::Eq`]. Because `SetElement::Hash()` and `SetElement::Eq()` are virtual, many different kinds of **Sets** can be created by deriving from **SetElement** and reimplementing `Hash()` and `Eq()`. `HashFct()` and `EqFct()` only need to be reimplemented if you want to have the same object occurring in more than one kind of set at a time. This degree of generality in **Set** is a fairly recent addition whose positive implications have not really been pursued in GPIO.

Sets use a dynamic hash table sizing scheme similar to that described in [3] except that our table sizes are always prime and we do not allocate tables for empty **Sets**. **HashTables** include reference counting features that are used to implement copy-on-write optimization of assignment and initialization of **Sets**.

---

[1] we almost always make destructors virtual so that `delete` can safely be used in cases where a pointer may point to a derived-class object

[2] this paper will use the terms *method* and *member function* interchangeably

Because **Sets** do not have a natural ordering to their elements, we provide a **SetEnumerator** class that can be used to enumerate the elements of a **Set**. By use of the reference counting feature of **HashTable** we have made it safe to modify a **Set** while enumerating it—a feature whose earlier absence was the source of some obscure bugs in application programs.

2.1.3 Set16, Set32

Class **Set16** provides for sets of integers between 0 and 15 (typically enums) that are implemented as bit vectors stored in short words. **Set32** is similar except for being larger (0 to 31, stored in long words). Both support a full compliment of set theoretical operations, almost all of which are implemented with inline functions that generate code identical to that resulting from the 'equivalent' C 'bit twiddling' expressions.

2.1.4 Exception, Handler

Our exception handling package is pretty much yet another `setjmp()/longjmp()`-based implementation. We allow for nested handlers and re-raising of exceptions. We do not address the problem of destructors for local objects being skipped, except with a caveat in the documentation.

2.2 GPIO Proper

GPIO itself consists of a group of non-application specific classes, and a few groups of application specific classes.

2.2.1 Core classes

Class **Class** performs a function similar to the Smalltalk-80 [4] class **Class**—it describes GPIO classes that can be redefined by application programs. It consists of virtual functions to identify the class, allocate an object of that class, allocate a collection of objects of that class for general use, and to allocate a collection of objects of some other class suitable for use as a member of an object of this class (this last one will be explained in section 3.2). Unlike Smalltalk-80 (or OOPS [5]) GPIO does not want to know about all application-defined classes—it only wants to know which GPIO classes have been derived from by an application—thus it is not fully general like these other systems.

Class **Object**, as in so many systems, is the root of the application-level GPIO class hierarchy. In the absence of multiple inheritance (hereafter, m.i.) **Object** is derived from **SetElement** so **Set** can be used to implement some GPIO collections. **Object** contains a static member that points to the current quasi-global **IOHandle** that is to be used to write or read GPIO objects. Since many GPIO objects participate in collections that distinguish objects by a **String** 'name', **Object** includes a virtual function `KeyString()` that can be implemented in derived classes to return this 'name'. **Object** reimplements `Hash()` and `Eq()` from **SetElement** based on `KeyString()`. **Object** also defines virtual functions `IdIO()`, `PageIO()`, `IO()`, and `GIO()` which are used to implement the non-virtual functions `OIO()` and `OGIO()` in a 'parameterized' manner. `IdIO()`, whose default implementation is a noop, is

implemented by those classes whose instances are pointed to by other objects, in conjunction with class **PointerMap**. **PointerMap** is an associative array (again derived from **Set**) that allows us to store actual pointer values in files for those object relationships that are not reasonably expressible as collections. This is achieved by arranging the file formats such that objects *pointed to* always occur before objects that point to them and then having the 'pointed to' objects store their memory address when written. When such object are read, their file 'address' and their current *memory* address are stored in the pointer map in the current **IOHandle**. When an object that points to it is read it reads the old pointer value from the file, looks it up in the pointer map, and stores the resulting memory address in the pointer member. Both sides of this operation are encapsulated in member functions of **Object**: `IdIOAction()` for implementing `IdIO()`, and `IOAndBind(...)` for doing I/O on pointers. While it is a nuisance for objects to have to 'know' that they may be pointed to, it saves a lot of space in the files and in the pointer map if most objects do not have to store their address, and this has been observed to be the case in GPIO.

`IO()` and `GIO()` are the non-graphical and graphical object I/O routines, respectively, that are reimplemented in each derived class. Implementations of `GIO()` always call the same class's `IO()` routine; in the case of fundamentally non-graphical objects, this is all they do.

Class **IOHandle** is a vaguely **Stream**[1]-like entity that encapsulates all of the low-level details of how files are written and read. It understands some simple types and allows for a fairly compact file format. It is currently based on stdio operations, because we were most familiar with them. In addition to maintaining the state necessary for file I/O, **IOHandle** also contains a **PointerMap** for use when reading an object and a method variable that is the appropriate kind of collection I/O for objects to do. Encoding this state in a method variable is another example of the 'data-driven' style that GPIO is written in—we tried to minimize the use of explicit flow-of-control constructs. The result, in conjunction with extensive use of virtual functions, is code with very few `if` statements and almost no `switch` statements.

Collection and **CollectionEnu** are abstract classes that define the interface GPIO expects all derived collection and collection enumerator classes to support. **Collection** specifies virtual functions to add and remove **Objects**, to remove all elements, to determine the cardinality of the collection, to generate an appropriate enumerator object, and to get the **Class** object that describes the objects that are contained in a collection. In addition, **Collection** provides a number of (for now) non-virtual functions built from these operations to do such things as add and subtract collections and `delete` all of the elements of a collection. The most important **Collection** methods from GPIO's perspective are `IO()` and `GIO()`, which perform I/O (non-graphical and graphical) on each element of a collection. Both are implemented by calls to the 'parameterized' inline function `_IO()` as follows [most global identifiers in GPIO start with GP; error checking has been omitted for brevity]:

```
inline
void
GPCollection::_IO(int G) {
    void (GPObject::*piof)();
    piof = G ? &GPObject::OGIO : &GPObject::OIO;
    GPIOHandle *h = GPObject::IOHandle;
```

```
    GPClass *klass = Class();
    GPeRecord br = klass->Identifier();
    if (h->writing()) {
        // if writing non-graphical objects, map the collection
        // 'type' to its non-graphical equivalent
        GPeRecord rec = G ? br : GPUnG[br];
        GPBaseRecord base(rec, Cardinality()); // a collection descriptor
        h->Write(&base, BASERECORD);
        GPCollectionEnu *en = newEnu();
        GPObject *p;
        while (p = en->next())
            (p->*piof)(); // call via pointer to member function to write it
        delete en;
    } else {
        GPBaseRecord base;
        h->Read(&base, BASERECORD);
        int i;
        for (i = base.RecordLength; --i >= 0; ) {
            // allocate an object of the appropriate class
            GPObject *p = klass->NewObject();
            (p->*piof)(); // to read it
            add(p);
        }
    }
}

void
GPCollection::IO() {
    _IO(0);
}

void
GPCollection::GIO() {
    _IO(1);
}
```

Class **CollectionEnu** defines virtual `next()` and `init()` functions that are used to advance through a collection and to reset an enumerator, respectively. The **CollectionEnu** returned by `Collection::newEnu()` will always be set up to enumerate the **Collection** that generated it.

Class **GPSet** and class **SList** are implementations of **Collection** employing **Set** and singly-linked lists of pointers, respectively. Unfortunately, neither of these are purely implementations of the **Collection** interface—both affect semantics in subtle and not-so-subtle ways. For instance, **GPSets** also support `operator[String]` for looking up objects by name, and GPIO assumes this capability in some places that it really should not (without making the need for this feature explicit).

One of these two classes is the default implementation of every application-level collection in GPIO. Because we firmly believe in type-secure interfaces we derive type-specific collection classes from one of these two for all application-level GPIO classes. This process of adding an implementation to a functional interface and then adding a type-specific interface to that is one area where the reality of C++ does not live up to the promise; we will revisit this issue in the 'problems encountered' section. All GPIO collections are homogeneous; we did not feel that our objects were sufficiently purely object-oriented for heterogeneous collections to be workable.

The design objects that are stored as files are referred to as *named objects* and are represented in the GPIO class hierarchy by class **NamedObject**. Examples of named objects are schematic drawings, schematic symbols, and connectivity files, or 'netlists'. **NamedObject** is derived from **Object** and contains a member of class **Name** that is a structured, application-specific name for that design object. This **Name** is used by the *named object locator* to locate the file representation of the object in our filing system. The details of the operation of the named object locator system, an interesting GPIO-based C++ application itself, are outside the scope of this paper. From an application perspective, classes derived from **NamedObject** are the lowest-level entities involved in I/O operations; **NamedObject** provides `Read(...)` and `Write(...)` methods that entirely encapsulate the lower-level details of object and file I/O. Both routines call a common, private `RW(...)` method that sets up exception handling, asks **IOHandle** to open the file for the object's **Name**, calls the object's (virtual) `IO()` function, and cleans up. `RW()` also takes care of stacking **IOHandles** when nested I/O operations are required. `NamedObject::Read/Write()` take an optional method variable argument that is actually the function called by `RW()` and is either `Object::IO` or `Object::GIO`. The purpose of this generality will be apparent when we discuss **Conn** and **Schematic**.

2.2.2 Overview of Application Level Objects

The application level objects in GPIO represent the components of one level of a hierarchical electronic design. GPIO currently supports three levels of representational detail: *abstract*, where only the interface to a block or cell is specified; *connectivity*, where the electrical connections among the interfaces of subcomponents of a block and between the interface to the block itself and its subcomponents is specified 'graph theoretically'; and *schematic*, where connectivity information is augmented with graphical information and cosmetic data to produce a representation suitable for viewing and editing with a graphical editor. An interesting feature of GPIO is that it recognizes the distinction between graphical and non-graphical versions of objects at a very low level. This allows it to process graphical objects as if they were non-graphical, a capability whose usefulness will be demonstrated below.

2.2.3 abstract

The most abstract of the three representations of an electronic design is what we call the *abstract*. It consists of class **Abstract**, which is a **NamedObject** and contains a collection of **Ports**. An **Abstract** is much like the declaration of a function in a programming language, and a **Port** is much like a formal parameter. Class **PageItem** is a semi-abstract class that is derived from **Object** and is the base class for all application-level classes whose instances

(can) occur on individual pages of a schematic drawing. Class **Port** is a **PageItem** (again, only due to lack of m.i.) and consists of a name, a type, a width (for multi-bit signals), and a collection of **NameVals**, or name-value pairs, which is essentially a property list.

2.2.4 connectivity

The intermediate level representation is *connectivity*. It is analogous to an abstract syntax tree for a function in a programming language in that it contains all of the essential information and no incidental information. It is embodied in class **Conn**, which is derived from **Abstract** and contains a collection of **Abstracts**, a collection of **SymInsts**, a collection of **PortInsts**, and a collection of **Nets**. The **Abstracts** are essentially declarations of other blocks, or designs, that are used in this design. The **SymInsts** are *sym*bolic *inst*ances of these other designs. **Abstracts** are the first example of an object that is pointed to by other objects and thus implements `IdIO()` with `IdIOAction()`.

Class **SymInst** is a **PageItem** plus a pointer to the **Abstract** of the design of which it is an instance, an instance-specific name, a time stamp, a collection of **PortInsts**, and a collection of **NameVals**. **SymInsts** are the first example of an object that points to another object and thus uses `IOAndBind()`. Class **PortInst** is derived from **Object** and contains a pointer to the **Port** of which it is an instance (either on an **Abstract**, when a **PortInst** is within a **SymInst**, or on a **Conn**, when it is directly part of a **Conn**), a pointer to the **Net**, if any, that is connected to this port, a 'back' pointer to the **SymInst** this **PortInst** is part of (or nil), and a collection of **NameVals**. As is apparent, **PortInsts** serve different purposes depending upon whether you are looking 'down' the hierarchy (from **Conn** to **SymInst**) or 'up' (at the interface to a **Conn**, from within it). **PortInsts** are similar to actual parameters in a programming language.

Class **Net** is a **PageItem** plus a name, a 'back' collection of **PortInsts** that it connects to, and a collection of **NameVals**. The 'back' collection of **PortInsts** associated with a **Net** is an example of annotation of the 'parse tree'—it is redundant information that is not stored in the files, but rather it is constructed on-the-fly as a **Conn** is being read. This is a natural consequence of the fact that **PortInsts** automatically maintain this 'back' relationship whenever their **Net** pointer is modified. This collection, and the 'back' pointer from **PortInsts** to **SymInsts**, are both essential for efficient traversal of connectivity, thus their inclusion in the basic GPIO data structures.

2.2.5 schematic

The least abstract representation currently supported by GPIO is *schematic*. **Schematic** objects are used in our CAD system to store schematic drawings for both logical and circuit level electronic designs. Continuing the programming language analogy, schematics are analogous to the source code for a function. They are a human-readable representation of information that is meaningful to a computer program plus cosmetic details that may be helpful to humans but are ignored by computer programs. Many of the classes at this level are graphical versions of classes at the connectivity level. Classes that are in this relationship have names beginning with **'G'**, such as **GAbstract**, **GSymInst**, **GPort**, **GPortInst**, and **GNet**. Along these same lines you might think of a **Schematic** as a **'GConn'**. All of these

'G' classes implement *both* `IO()` and `GIO()` The schematic level contains two named objects derived from **Abstract**: **Schematic** and **Symbol**.

Class **Schematic**, which is derived from **Conn**, redefines all of **Conn**'s collections and adds two of its own. The **Ports** from **Abstract** become **GPorts**; the **Abstracts**, **SymInsts**, **PortInsts**, and **Nets** from **Conn** become **GAbstracts**, **GSymInsts**, **GPortInsts**, and **GNets**, respectively. Additionally, there is a collection of **Shapes**, which are cosmetic graphics such as lines, polygons, and arcs, and a collection of **Texts**, which are graphical strings with an assortment of display attributes.

Class **GPort** is derived from **Port** and implements the port's name as a **Text** object rather than a **String**. **GPort** adds a **Location** member to describe the port's position on a page and a collection of **NameValInsts**, which are graphical instances of **NameVals** containing a pointer to the **NameVal** of which it is an instance and a collection of **StringInsts**, which are instances of **Strings** containing a pointer to the **String** of which it is an instance and a few members that hold display attributes. Class **GAbstract** is derived from **Abstract** and adds only a pointer to the graphical **Symbol** of whose design it is the abstract. `GAbstract::GIO()` performs the additional function of maintaining a global **Symbol** table that is shared among all **Schematics**. Class **GSymInst** is derived from **SymInst**. It redefines **SymInst**'s **PortInsts** to be **GPortInsts**, redefines the **Abstract** pointer to be a **GAbstract** pointer, adds a **Location** member that specifies the position at which an instance of a **Symbol** is to be drawn on a page, and adds a collection of **NameValInsts**. Class **GPortInst** is derived from **PortInst**. It redefines the **Port** pointer to be a **GPort** pointer, the **Net** pointer to be a **GNet** pointer, and the **SymInst** 'back' pointer to be a **GSymInst** pointer. It also adds a collection of **NameValInsts**.

Class **GNet** is derived from **Net**. It redefines the **PortInsts** to be **GPortInsts** and adds a collection of **Wires**. Class **Wire** is derived from **PageItem** and contains a collection of **Segments** and a collection of **NameValInsts** that are instances of the **NameVals** that are attached to **GNets**. Class **Segment** is derived from **Object** and consists of two **Points**.

Class **Symbol** is derived from **Abstract**. It, like **Schematic**, redefines the **Ports** to be **GPorts** and adds collections of **NameVals**, **NameValInsts**, **Shapes**, and **Texts**.

Classes **NameValInst**, **StringInst**, **Location**, **Text**, **Shape**, **Wire**, and **Segment** are purely graphical and do not even bother to implement `IO()`. Consequently, they are completely avoided by the `IO()` routines of the other schematic classes.

2.2.6 connectivity from schematics

Just as the source code for a program must be parsed before it can be compiled or interpreted, so must a schematic drawing be processed to extract the connectivity information that is needed by most CAD tools. Because GPIO's schematic level data structures are all derived from its connectivity level structures, a GPIO-based application could just read in a **Schematic** object and ignore the derived class information that is not needed. There are two related problems with this: first, the schematic files are much larger than the connectivity files and thus take longer to read; and second, the unneeded information in the graphical structures can require an unacceptable amount of space in some applications.

GPIO addresses this problem by providing **Schematic** with a method called `WriteConn()` that writes out the **Schematic** as if it were a **Conn**. This is where the parallel `IO()`

and `GIO()` **Object** and **Collection** methods come in, as well as the optional argument to `NamedObject::Read/Write()`. Since `GIO()` routines are always implemented as calls to `IO()` routines plus I/O of the additional graphical data, the result of performing non-graphical I/O on all the schematic level objects is identical to the result of performing it on the corresponding connectivity level objects. A description of exactly how this is implemented is too involved for this paper, but it involves manipulation of the quasi-global method variable that controls which **Collection** I/O method objects use on their member collections.

`WriteConn()` is used by `Schematic::Write(...)` to produce a separate file containing only connectivity information. Applications that only want connectivity for a design can then create a **Conn** object for that design and `Read()` it in. Because it was created from the schematic data at the same time that the schematic data was written, it is guaranteed to be synchronized with it.

3 Extensibility

One of GPIO's goals was to allow application programs to extend its data structures to suit their needs without having to change GPIO itself. Since GPIO data structures are defined in terms of objects (aggregates) and collections, these are the things that an application can extend. Both of these entities are represented in GPIO by C++ classes, so the natural mechanism for extending them is C++ class derivation, or inheritance.

3.1 Application-derived classes

Since the original purpose of GPIO was to ensure agreement between different applications as to the form and content of data files, we decided early on that the 'default' GPIO objects would completely define the information content of the files. The intention of extensibility was not to allow applications to add *information* to the objects, at least not information that would be stored in the files. Instead, the intention was to allow *data* (i.e., redundant information) and application-specific information to be added to the objects for *run time use* only.

The implementation of this capability has taken several forms, but all of them were trying to simulate 'virtual' constructors. The current scheme uses class **Class** to encode knowledge of how to construct an object of each of the application-redefinable classes. When an application derives from a GPIO class, say from **SymInst** to **MySymInst**, it also derives from the corresponding descendent of class **Class**, **SymInstClass** to **MySymInstClass**, and creates a single static object of this new meta-class. As part of the derivation process, the application will reimplement `newObject()` to return a new instance of its derived class, **MySymInst**. The constructor for the default GPIO meta-classes is designed so that the global pointer-to-meta-class-object that GPIO knows about, `SymInstCls` in this case, will end up pointing to this instance of the application-derived meta-class, **MySymInstClass**. Internally, GPIO never `new`s application redefinable classes; instead it invokes the `NewObject` method of the appropriate meta-class object:

```
...
SymInst *si = SymInstCls->NewObject(); // when we know the type
...
```

```
...
Class *cls = ...;
Object *p = cls->newObject(); // when we don't [see Collection::_IO()]
...
```

3.2 Application-derived collections

Collections are the one area where GPIO is purely object oriented; class **Collection** contains nothing but methods, many of them virtual. GPIO defers implementation decisions about collections to applications in two ways. First, it allows applications to redefine the default collection implementation for each application-redefinable class, for instance, all collections of **PortInsts**. Second, it allows applications to control what type of collection is used for a given class for each case where such a collection occurs as a member of another object, for instance, the **PortInsts** collection in a **SymInst** can be different than the **PortInsts** collection in a **Net**. The purpose of this generality is to give applications as much control as possible over the time-space tradeoffs. In a space-critical application the less frequently used collections could be reimplemented to use minimal space (or non at all, theoretically).

The implementation issues for application-redefinable collections were very similar to those for redefinable objects. GPIO always invokes methods of the meta-class objects when creating collections. Recall that class **Class** has a method `newCollection()` and a method `newCollection(Class *ofClass)`. The first of these allocates the 'default' collection of the class described by the meta-class, e.g. `SymInstClass::newCollection()` creates a new instance of the default **SymInsts** collection. The second of these allocates a collection of the class described *by its argument*, `ofClass`, suitable for use as a member of the class described by the meta-class, e.g.

```
SymInst::SymInst(...) {
    ...
    portInsts = SymInstCls->NewCollection(PortInstCls);
    ...
}
```

The default implementation of `newCollection(Class *ofClass)` simply returns `ofClass->newCollection()`, so that all member collections of the same type of object are the same.

All of this deriving, reimplementing, and declaring of objects can get rather tedious, especially since within GPIO, and for most applications, the process is very mechanical. In an attempt to alleviate these problems, GPIO provides several horrendous token-splicing preprocessor macros for generating both declarations and implementations of derived classes.

4 Experience

GPIO has so far been used for four different sorts of applications. It was used to replace the database portion of our existing CAD system, while leaving the remaining portions largely intact. This was an interesting exercise, because most of the system was written in PL/1.

C++ actually turned out to be more helpful than not in the integration task. Because the goal of this project was not to build new applications based on GPIO's objects, but rather to build existing PL/1 data structures from GPIO's objects, this was less than an ideal showcase for GPIO's capabilities. It did, however, demonstrate that GPIO worked and that it was efficient enough in execution speed, memory usage, and data file sizes, to be practical even when used in this rather twisted way.

Some of the tools in the system *were* rewritten from scratch in light of GPIO. One of these is a browser/librarian program that does not actually operate on the contents of the design data files to any appreciable extent. Consequently, it was able to use GPIO 'as is' and it did not stress any of GPIO's more sophisticated extensibility-related features. One 'substantial' tool that was rewritten was our schematic verifier—a program that is something like a 'lint' for schematics. It performs a fairly detailed semantic analysis of a drawing and reports a number of common errors. In order to do the degree of analysis it does it needs more powerful objects than the ones provided by default in GPIO, so it was a reasonably good test case. It derives off of most of the 'G' versions of the connectivity objects; in fact, it turns out that if you want type-secure interfaces and you don't want to end up casting everywhere you are forced to derive from all members of a 'related' set of classes if you derive from any of them.

The fourth use to which GPIO was put was to build a brand new utility that would allow textual descriptions of connectivity (e.g., SPICE net lists) to be stored in our CAD database. This application was not particularly demanding, but it was the first program to build GPIO structures 'from scratch'. The developers of this program were not affiliated with the developers of GPIO at all, so they faced a steep learning curve.

So far we have not had any applications reimplement the default collection for any GPIO object type, let alone exploit the capability of using different collection types in different contexts. There are several possible explanations for this, but it is quite possible that some of these features are excessively general.

5 Problems Encountered

By far the most difficult problem we faced was getting used to C++ and object-oriented design. Early versions of GPIO were little more than glorified C `structs` with trivial `Set` and `Get` methods for each member. The present library is the result of a year of nearly constant refinement and redesigning. In the search for solutions to our problems, we left no stone unturned in C++. This was at once an interesting and frustrating mode to operate in, since C++, being a new language, had (and still has) numerous little-explored areas.

Perhaps the biggest impediment to a clean conceptual design is the problem of creating type-secure interfaces to implementations of abstract classes. We would very much like the hierarchy of our collection classes, at some level, to mirror the hierarchy of the classes of the objects in those collections. Unfortunately, in order to do this today and still retain independent control over how each collection is implemented we would be forced to replicate the implementation for every type of collection. We are not certain, but we suspect that one of the 'parameterized class' schemes being bandied about might solve this problem.

An implementation problem that we faced was how to construct objects whose members require different initialization in derived classes than in base classes. The most common

case of this in GPIO is the connectivity-level classes that contain collection members. These members are not actually collections, but pointers to collection objects. The schematic-level (G) objects derived from these use the same pointers to point to different collection objects, but we would like to only initialize those pointers once. Our initial 'solution' was to use virtual functions to initialize these pointers, but this does not work because during the execution of a base class constructor the virtual type of the object is the base class, not the derived class (although the analogous situation is *not* true for destructors). As things stand, there appears to be no good way to solve this problem. For now we are allocating the collections in the base class constructor, then `delete`ing them in the derived class constructor and allocating the derived class's collections.

Another recurring problem has been the lack of m.i. Its absence has forced us to define common methods at lower levels in the class hierarchy than really makes sense in order to have them available in all of the right derived classes. We have done some exploratory work with an experimental m.i. version of C++. Our initial impression is that m.i. can provide a very high degree of conceptual elegance, but it can also be very expensive in terms of object size when used extensively.

It might be useful to have meta-classes built into C++ so that such things as the name, size, constructors, and destructors for a class would be explicitly available as 'first class' entities. This would also open up some interesting opportunities for programs to control the 'dynamic' type of objects themselves.

It should be mentioned that C++ significantly overtaxes the functionality of most linkers that we have used. Features such as static object construction/destruction and sharing of virtual function tables which depend on linker support are very difficult to use. Our suspicion is that before too long C++ will needs its own compilation and linking paradigm not built out of the simple tools used for C programs. Once this program building environment is in place C++ will be free to clean up some of the current problem areas.

References

[1] Stroustrup, Bjarne: *The C++ Programming Language.* Addison-Wesley, 1986.

[2] Stroupstrup, Bjarne: *What is Object-Oriented Programming?* Proc. 1st European Conference on Object-Oriented Programming. Paris, 1987.

[3] Cox, Bradley: *Object-oriented Programming: An Evolutionary Approach*, Addison Wesley, 1986

[4] Goldberg, A., and Robson, D.: *Smalltalk-80 The Language and its Implementation*, Addison Wesley, 1983.

[5] Gorlen, Keith: *An Object-Oriented Class Library for C++ Programs*, Software—Practice and Experience, December, 1987.

C++: From Research to Practice

S. B. Lippman
B. E. Moo

AT&T Bell Laboratories
Warren, New Jersey 07060

1. Introduction

Research into language design has continued apace essentially since the invention of high level languages in the fifties. Each year sees yet another new language vie for inclusion into the much more restricted set of languages used for serious development. Few succeed.

At one level, of course, C++ is being used in practice because it is a very pleasant language in which to program. But, other new languages exist that are pleasant to program in and other languages have facilities that mesh cleanly and support programming at higher levels of abstraction. Yet we are here at a C++ conference and few other languages have generated interest sufficient to justify such a gathering. Why C++?

For the past two years, we have been involved in the effort to support C++: initially in the development organization that supports and distributes the AT&T C++ Translator, more recently focusing on spreading the use of C++ on major development projects within AT&T Bell Laboratories. Through this experience we have come to believe that much of the success of C++ is due to what we call the "meta-lingual" aspects of the language. By this we mean those aspects of C++ that transcend the language itself: the portability of its initial implementation, its compatibility with C and its pragmatic evolution. Interestingly, these very aspects of C++ that seem to have contributed to its ready acceptance have also contributed greatly to the complications of providing and supporting C++.

2. C++ Acceptance

C++ in its present form was first made available to internal AT&T projects and to universities in early 1985. This initial implementation of the language was Stroustrup's cfront Release E which compiled C++ source into C. Early use inside AT&T, as well as in the universities, was mostly in small, experimental projects. With some minor enhancements to the language and many bug fixes, that implementation of C++ was released by AT&T as the AT&T C++ Translator in the fall of 1985. Until earlier this year, this implementation formed the basis for all C++ implementations of which we are aware.

The formal release of the AT&T C++ product paralleled the first uses of C++ in actual development projects. The first products written in C++ at AT&T were initiated at about the time of the 1.0 Release of the C++ Translator. With each subsequent release of the C++ Translator the number of users, both inside and outside of AT&T, and the size and complexity of products being written in C++ have grown.

Today, C++ is rapidly becoming the de facto standard language within AT&T Bell Laboratories for development of new features and products. The primary use of C++ has been the development of new, mid-size (500,000 lines of code or less)

systems to support specialized network applications. Typically, new users of C++ have been programming in C and often the initial use of C++ is to provide some distinct new functionality to an existing system written in C. Users migrating to C++ are concerned about the expected things: support, training, documentation, ease of use for C programmers and ease of integration with existing C code. Experience on these projects has demonstrated the productivity gains expected from strong type checking and data abstraction.

A few projects have been in the field long enough to include maintenance releases produced by developers other than the original implementors. Experience on these projects has demonstrated the promise of data abstraction and object-oriented design in allowing modifications and additions to the system without requiring complete understanding of the whole application.[1]

On the whole, our experience in using C++ has been very positive. For our user community, the ease of migration from C to C++ has been especially important. Its availability on a wide variety of systems has also been key.

3. Portability

The greatest reason for the success of C++ is its availability on essentially any system. Most new languages must go through a relatively long period during which compilers for the language are available only on a restricted set of machines. Languages then must go through an uncomfortable "chicken and egg" period where potential new users, who might like to try the language, will only do so if a compiler exists for their system, but software vendors can only justify the relatively high costs of making a compiler available on those systems for which demand already exists.

This ready availability of C++ comes directly from the deliberate design decision in Stroustrup's initial implementation to generate C rather than object or assembly code. This has meant that C++ can be made available reasonably easily for any machine for which a C compiler exists. In essence, by generating C, the problem of porting the compiler is reduced from the tricky and expensive one of providing a code generator for each supported machine to the relatively straightforward problem of porting a roughly 20,000 line program. In practice, this has allowed people interested in C++ to use the language at a heretofore unprecedented low cost for a new language.

Likewise, once commercially available, C++ could be ported and made available more quickly than has been true of other languages. For example, within 6 months of its first commercial release from AT&T, implementations of C++ based on the AT&T Translator were available from other vendors for the PC market and several major mini-computers. It is today commercially available on over 24 different systems. Interestingly, it is only within the last several months that machine specific compilers, generating object code directly, are being made available. Surely the large and growing number of users of the C++ Translator helped these compiler writers justify the risk of introducing a new language into their product lines. It is interesting to speculate whether these products would ever have been written if C++ had been available only on the small set of machines that would have been possible had the initial implementation not generated C. Would

sufficient users of C++ have existed to justify such a great investment? Would sufficient users of C++ have existed to justify last year's USENIX C++ Workshop or this USENIX C++ Conference?

While this ready availability of C++ means that one can decide to use C++ on its own merits, without migrating to a new system or getting locked into a particular configuration, it does not come without cost. Distributing and supporting any large program on a wide variety of systems is a non-trivial task. As a general rule of thumb, any assumption one makes about the operating environment will be violated by at least one system.

Portability of the C++ Translator has been a consistent and explicit design goal. At times this has required sacrificing the best solution for a particular system to a general solution that, with possibly minor modifications, is applicable to all systems. In other cases, the goal of portability for the implementation has simply meant dealing, somehow, with the numerous system idiosyncrasies that invariably complicate programs intended to port to a wide variety of machines and operating systems.

The handling of static constructors presents a good example of the first problem: the tradeoff between implementing a feature with the most general approach for all machines rather than the best approach for a given machine. A static constructor is a constructor that must be called prior to the start-up of the program. Failure to do so will likely result in a run-time core dump of the program. For example,

```
#include <iostream.h>

class Buf {
public:
        Buf( int len = BUFSIZ );
        // ...
};

Buf inBuf( 4096 ); // requires a static constructor

main() {
        // inBuf must be initialized at this point
        // otherwise, this will write into hyperspace!!
        while ( cin >> inBuf )
        {
                // ...
                cout << inBuf;
        }
}
```

The problem is as follows. All C and C++ programs start execution in main(). However, if the first statement in main() is executed prior to the constructor call for inBuf, the program will fail. The program is composed of separately compiled modules and a set of libraries. The compiler has no way of knowing what constructors the executable requires. Generally available link-editor technology does not help. Any solution is going to be a hack.

One aspect of a solution is when to invoke the constructors. Here are two reasonable alternatives:

a. have crt0.s, the start-up routine, execute the constructors. ctr0.s, however, is written in assembly. This strategy requires a non-trivial amount of expertise from an individual wishing to port the Translator.

b. insert a function, _main(), into main() as the first executable statement. Then create a _main() function to execute the static constructors. An individual porting the Translator need not even be aware of _main()'s existence.

A second aspect of a solution is how to collect the set of static constructors that need to be called. The simplest approach would be to gather together the static constructors into a canonical function for each object module within the executable. These functions then would need to be invoked from _main(). Again, there are two possible alternatives:

a. Patch the executable directly. Read and modify the a.out directly, threading a list of initialization functions through which _main() could then iterate. This strategy requires intimate knowledge of the system's object file format. The individual porting the Translator would have to provide a new instance of patch.

b. Munch the symbol table. Dump the symbol table names, keeping track of all initialization functions. Build a table of function pointers and link with the executable. _main() can then walk through the table. This strategy requires some tool which can print out the symbol table strings. Under the UNIX® Operating System, the nm command is sufficient. An individual porting to any UNIX System need not even be aware of the existence of munch.

In each solution set, choice (b) clearly provides maximum system independence, while choice (a) is cleaner and likely to be faster. The ease of implementing case (b) so dominates the knowledge that would be needed to implement (a) on a new machine that the munch version was developed and distributed. To gain the speed advantages for a large class of machines, a patch version of the Translator for System V machines is also distributed with the product. Most ports to non-System V machines have, however, initially been done using the munch approach. munch versions of the C++ Translator have been ported from mainframes and super computers to PCs. Once a user community for a particular system develops, an optimized solution for that system can be implemented.

Our experience indicates that the C++ Translator is surprisingly easy to port to a new system. Design choices such as generating C and taking implementation tradeoffs that favor system independence have helped to restrict the porting effort to changes to header files, and establishing the target machine's size and alignment requirements. However, there remains a second, nittier class of portability decisions driven by the idiosyncrasies of this or that particular machine. Here too, our experience and the experience of our users in earlier porting efforts have helped shape the current implementation of the Translator. For example,

- Trouble-shooting the first port of Release 1.1 to a Control Data Cyber-160 turned into a transcontinental process of problem isolation between one

system which did not exhibit the problem but which had debugging facilities, and a second system which exhibited the problem and as a result did not have a working system capable of helping in the debugging process.

As it happens, the high bits of a pointer were being used to set protection rings. Internally, the Translator would cast integer constants into pointers for storage then cast them back into integers for later processing. That proved rather disastrous on the Cyber in question. The constant 8, for example, suddenly measured the distance between Earth and the nearest galaxy.

In Release 1.2, the internal data structure had a new derived class for storing integer constant values; explicit integer/pointer casts were removed. This also helped simplify porting to the Intel 80286 processor.

- A second example of an unexpected system idiosyncrasy occurred during a port of Release 1.2 to a Hewlett Packard 9000 series machine. In this case, everything worked fine, except that static destructors for class objects were not being invoked. Moreover, the same code executed correctly on an AT&T 3B20, Sun 3/60, VAX 8550 and Amdahl.

  Static destructors pose the same class of problem as static constructors. They must be invoked following completion of the program, but before the _cleanup() library call. The Translator's solution is to provide its own version of exit(). This version in libC.a executes the table of static destructors built up by munch, then in turn calls _cleanup() and _exit(). The only way not to invoke the static destructors is by not invoking exit() (for example, a direct call of abort()). Or, as it happens, by invoking the libc.a instance of exit().

  The Translator presumes that an unresolved symbol will bind to the first instance found in an archive. Therefore, to replace the libc.a's exit() with that defined in libC.a, the command line is fed the libraries in the order libC.a libc.a.

  As it happens, unless an explicit call to exit() is made within the program code, the link-editor on our HP9000 bound exit() to the libc.a instance regardless of the archive's placement on the command line. The simple fix of having the Translator insert a call to exit() within main() solves the problem. However, programs that correctly return from main() are likely to generate a spurious *statement not reached* warning message. A clean system level solution to this problem has as yet not been determined.

4. C Compatibility

Another important factor in the rapid spread of C++ has been the ease of using C++ with C. There are two aspects to this compatibility: the ability to use C++ with existing C based systems and the ease with which C programmers can learn and make effective use of C++.

In all cases, our user community is migrating to C++ from C. Some of what we have learned from their experience in melding C and C++ includes:

- Keep the interface between the C and C++ parts clear and clean.
- "If it ain't broke, don't fix it"[2]: that is, there isn't likely to be sufficient payoff to justify converting an existing system unless you are also doing some substantial new development at the same time.
- Use the new features of C++ gradually rather than all at once. You can get spaghetti classes just as you can have spaghetti code.
- Effective use of C++ comes with better design. You will get big payoffs from having a couple of the more experienced people spend the time to design a few classes that are fundamental to your application.
- Most users can start with a minimal knowledge of C++ and yet make effective use of general purpose or application specific class libraries.

While the close relationship C++ maintains with C is one of its greatest strengths, it can also be a source of tension in the language. The C declarator syntax is a case in point. Here maintaining compatibility with C results in some unavoidable ambiguity in the language which can only be resolved by explicitly defining the ambiguity resolution within the language. Maintaining the old-style C syntax is likely to be the design choice for which Stroustrup's name will be most taken in vain by compiler writers to come!

Explicit conversions in C++ may take the form of either C-style casts, such as

```
(X) i;
```

or function-style casts, such as

```
X (i);
```

Function style casts are necessary to provide support for casts (constructors) of user defined types which require more than a single argument, for example:

```
z = complex(x,y);
```

Using C style casting is not possible since

```
z = (complex)(x,y);
```

would be interpreted as casting the result of the comma operation (x,y)! However, at local scope, the function style cast syntax introduces parsing ambiguities. The conflict occurs when a statement begins with a type name. Lookahead can usually resolve the ambiguity. For example,

```
typedef int (*PFI)();
class X {
public:
        PFI f;
        // ...
};
```

```
typedef X *PX;
void **p, **q, **r;

f() {
        PX( *p )->f = 0;
}
```

Until the member selection operator (->) is seen, it is equally possible for PX(*p) to be evaluated as

- the declaration of a local instance of p.
  p would be of type *pointer to a pointer of* X. This is possible because C permits extraneous parentheses in declarations! The following two declarations are equivalent.

```
// equivalent declarations of p
PX *p;
PX (*p);
```

- an expression involving the global instance of p.
  p is first dereferenced. Its void* value is then cast to a pointer to X.

The member selection operator disambiguates the statement. Global p is dereferenced and cast to a pointer to X.

Here is a second statement requiring lookahead:

```
ff() {
        PFI( *q )( int );
}
```

Until the closing right parenthesis following the keyword int is seen, it is equally possible for PFI(*q)(int... to be evaluated as

- an expression involving the global instance of q.
  q is first dereferenced. Its void* value is then cast to a pointer to a function returning an int. That function is then invoked through q. q is passed a single argument which begins with the letters int.
- the declaration of a local instance of q.
  q would be a pointer to a function with a return type PFI.

The closing parenthesis disambiguates the statement. int evaluates as a type, not identifier. A local instance of q is being declared.

These are instances in which the ambiguity can be resolved by lookahead. This is not always the case. For example,

```
f3() {
        PFI( *r )();
}
```

It is equally possible for the parser to interpret this as

- a declaration statement of a local instance of r.
  r is a pointer to a function taking no arguments and returning a PFI.
- an expression statement involving the global instance of r.
  r is first dereferenced. Its void* value is then cast to a pointer of type PFI, which is then invoked.

The resolution is a meta-rule. Whenever a declaration and an expression are equally possible, the statement is taken to be a declaration.

C++ is not a formal superset of C, and there do exist one or two fundamental differences between the two languages. These differences are generally a result of the greater functionality of C++ and in our experience, have not caused difficulties for our user community.

The C language maintains separate name spaces for user-defined tag names and identifiers. This permits the same name to be used both as a tag name and an identifier at the same scope. This has given rise to C code such as the following:

```
struct stat { /* ... */ };
struct stat stat;

struct mallinfo { /* ... */ };
struct mallinfo mallinfo();
```

The tag name instances are indicated by prefixing the tag name with the struct, enum or union keyword.

C++, on the other hand, maintains a single name space for both user-defined tag names and identifiers. Were C++ not to have a single name space, programming with classes would need to look something like the following:

```
// a hypothetical syntax were C++ to maintain two name spaces
class B { /* ... */ };
class X : public class B {
        class B b;
        operator class B *();
        class X( const class X& );
        class X( int i = 0, int j = 0 );
        class X& operator=( const class X& );
        class X& operator+( class X& );
        // ...
};
```

```
class X::class X( const class X& x ) { /* ... */ }
class X& class X::operator=( const class X& ) { /* ... */ }

class X *xp = new class X();

// an explicit invocation of conversion operator
class B *bp = xp->operator class B *();

// an explicit constructor is required
class X x = *xp + class X (1,2);
```

In a sense, by breaking name-space compatibility, C++ maintains C's legacy of lexical elegance.

Certain small incompatibilities are the result of deliberate, often difficult, decisions. They are not made lightly, and in general reflect a trade-off between breaking compatibility with the older language or losing valuable functionality. A case in point is the scope of enumerations declared within a class type.

In C++, enumerations are local to the class in which they are declared. Private and protected enumerations are encapsulated as are the other private and protected members of a class. For example,

```
class ZooAnimal {
friend feedingHours( ZooAnimal& );
protected:
        enum Status { ONLOAN, ONDISPLAY };
        Status status;
        // ...
};

class Bear : private ZooAnimal {
protected:
        ZooAnimal::status;
public:
        isOnDisplay();
        // ...
};

// ok: these are permitted access to ZooAnimal::ONDISPLAY
Bear::isOnDisplay() { return( status == ONDISPLAY ); }

feedingHours( ZooAnimal& z ) {
        if ( z.status == ZooAnimal::ONDISPLAY )
        // ...
}
```

```
// error: these are not permitted access to ZooAnimal::ONDISPLAY
class TeddyBear : private Bear {
friend playingHours( TeddyBear& ):
public:
        isOnDisplay() { return( status == ONDISPLAY ); } // error
        // ...
}

playingHours( TeddyBear& t ) {
        if ( t.status == ZooAnimal::ONDISPLAY ) // error
        // ...
}
```

In C++, each class maintains an associated scope. Members within that scope may be referenced directly using the class scope operator. ZooAnimal::ONDISPLAY accesses the element ONDISPLAY within the scope of ZooAnimal. The global name space is not cluttered with element names only of interest to ZooAnimal and its derivations. The possibility that including a new class containing an enumeration will cause name collisions which break existing code is eliminated.

There is nothing analogous with regard to the C struct. There is no permissible syntax to allow access of elements within a struct, if structs in fact maintained their own scope, which they do not. Of course, in the draft proposed ANSI C standard[3], enumerations declared within a struct assume the same scope as that enclosing the struct. What other meaningful choice is there in C?

5. A New Language and A New Way to Program

Throughout its evolution, users of C++ have influenced its definition. Various features have been added to the language as a direct result of user feedback. Obviously for many of its early users, this ability to help move the language in a direction that made solving their problems easier was a real boon. Others, who've never suggested a change, benefit more indirectly as the language evolved to better meet specific needs that have occurred in real development.

The rule of thumb to date has been that requests for extensions to the language must result from a genuine need encountered by two unrelated users in trying to implement solutions to real problems. This has meant that changes in the language have tended to be designed to provide a better way of doing something people really have wanted to do. Once a feature or ability is requested, a process is undertaken to understand how the change fits with the existing language, to apply theory and engineering approaches to determine a clean solution and finally to provide a prototypical implementation to test the applicability of the solution to the original problem. Some enhancements to the language have resulted from requests for feature additions which would make particular classes of problems easier to solve. Others came from what were initially reported as bugs in the implementation. A few others resulted from natural extensions of existing features in the language. Here are some examples:

- Prior to Release 2.0, the base part of a derived class could only be initialized with the specified arguments of its constructors. An attempt to initialize it

with another base class object resulted in a compile-time type violation. For example,

```
class Base {
public:
        Base( int );
        Base( char * );
        // ...
};

class Derived : public Base {
public:
        // ok
        Derived( int i ) : (i) {};
        Derived( char *s ) : (s) {};

        // not ok prior to Release 2.0
        Derived( Base& b ) : Base(b) {};
        Derived( Derived *d ) : Base(*d) {};

        // ...
};
```

One source of this change came from an active user of C++. The argument was that object with object initialization was permitted for non-derived and member class objects. In these cases, a default bitwise copy was applied. It was only in this case that bitwise copy was not being applied. The argument was not theoretical, he claimed; his application needed this.

A second source of this change came from problems with default bitwise copy itself. (Bitwise copy has been replaced with memberwise initialization and assignment. A discussion of this can be found in [4].)

- Prior to Release 2.0, the order of initialization of member and base classes was undefined. This order is now fixed. (A discussion of this can be found in [4].)

  One impetus for this change was another active user of C++. He was speaking about extending his current work to incorporate multiple inheritance during an internal AT&T C++ user group conference. There was, he said, one obstacle he saw no solution to; that is, the undefined initialization order. His application allows for arbitrarily complex user-defined class types to be written out and read from disk. To insure the integrity of this process, the order of initialization must be guaranteed.

  A second impetus for this change comes from the proliferation of C++ compilers. Without a specified initialization order, uniformity across implementations cannot be guaranteed.

A particularly interesting example of the pragmatic basis for the evolution of C++ is the design of type safe linkage.[5] We had received several related concerns about the existing mechanism for overloading functions. The solution which emerged came to provide more than a simple fix to these problems. As alternatives to the existing mechanism were discussed and dismissed, the ultimate solution

evolved in such a way as to provide a natural and useful inter-module extension of the type checking features of C++. Beta users of this new type safe implementation report finding latent bugs in existing software. Incorrect function references between files which had previously gone unnoticed were now being caught.

Again, these advantages come at some cost. Here the issues are not for the implementation, but for our users. Because C++ is a new language, the implementation tends to be less stable than compilers for older languages. Documentation tends to be scarce and often incomplete or out of date. Because C++ provides support for new approaches to programming, new ways of organizing developments and designing programs are evolving. This results in users getting started and moving forward while a culture and common wisdom is still being developed.

Conferences such as this one, and publications of user experiences have begun the process of creating a C++ style. Tutorials and papers at this conference have ranged from introductions to C++ to techniques for advanced uses of the language. With time, these ideas will gel into a set of conventional approaches. Until then users are left more on their own than in other, more established languages. Informal mechanisms for spreading these notions have been surprisingly effective. The C++ netnews group (comp.lang.c++) has turned out to be a simple mechanism for quickly disseminating statements about what the language says and for discussions of what techniques are useful. More formal mechanisms are also beginning to appear. Many C++ books are either now hitting the book stores or will appear within the next six months. A C++ Newsletter is rumored to be in the planning stages and formal training courses are now available from a variety of vendors.

Internally, we have initiated a C++ user group which acts as a focal point for information about C++. We publish a more or less bimonthly newsletter which in addition to details about availability of software, includes a column focused on C++ technique. Additionally, we have put in place a set of people to help our user community. We have a hotline for questions about C++ as well as C++ consultants who work closely with specific projects participating in design discussions, providing suggestions for better ways of exploiting C++, and providing a general C++ resource to project personnel. As we develop new class libraries, in addition to the traditional UNIX System manual pages and tutorial material, we go out to the projects to present the new classes and discuss ways of using them within the user's application. Each of these activities is helping us understand useful ways of applying C++ to real world applications and is allowing us to help shape and subsequently document the ways in which C++ is being used in practice.

Perhaps most intriguing about the use of C++ in substantial development projects is the potential for the evolution of new styles of programming organization. In the simplest case, our experience already has been that C++ really does encourage the hitherto elusive goal of concentrating on upfront design as opposed to leaping directly into implementation. Several projects have reported that they have found it natural in C++ to start with the fundamental classes for the application, thus setting the overall design in place early. Related to this, has been a shift in organizing the staff on these projects. Several of our projects have been able to assign relatively experienced staff to design and implementation of the key classes for the application and been able to use more inexperienced people to develop

application code based on these libraries. This is giving us the ability to bring people on board more quickly, but also giving them the opportunity to concentrate on understanding the application before having to deal with the intricacies of the implementation.

As we look further to the future, we see even more pervasive changes in development organizations. As C++ libraries proliferate, we expect to see dramatic reductions in the amount of new code required to solve a particular problem. For simple, but uncontrived, test cases, we have seen code reductions of as much as four to one to implement the same functionality in C++ rather than in C.[6] If this scales and holds even to a two to one reduction, we should be able to cut development staffs dramatically for a given application size. Additionally, we expect the modularity which C++ supports to allow us to subdivide problems more cleanly so that the overhead of communications can be reduced via use of clean well thought out interfaces. Both these forces should allow smaller application teams to be formed. Smaller teams in turn will lead to more productive software development. It is well known that current software development cycles are greatly complicated by project size. Intuitively as software developers we know it from our own experience: the larger the project, the more time is spent on project communication and ensuring consistency among the parts of the application. More formally, even early studies[7] have borne out this increase in overhead with size of project. If we can cut development project size, we can cut this overhead as well.

6. *Conclusion*

There are many reasons why any system and especially why any programming language gets used. Our experience in supporting C++ for use on real development projects leads us to value certain aspects of the implementation that go beyond the actual features of the language. In the case of C++, we believe that its portability, its compatibility with C and its pragmatic evolution have been fundamental in its rapid and widespread user acceptance.

7. *Acknowledgement*

Laura Eaves implemented the lookahead component of the Translator's parser. Much of the discussion of parsing ambiguity comes from discussions with her. Comments on the initial abstract for this paper helped shape our treatment of many of the details presented.

NAPS - A C++ Project Case Study

C. Berman
R. Gur

AT&T Bell Laboratories
Middletown, New Jersey 07748

1. INTRODUCTION

Object-oriented Programming (OOP) promises to be one of the major advances in software methodology in the next decade. C++ is one of the first languages that offers a cost-effective execution environment for OOP. Yet the decision to use C++ in a production environment should be made on the basis of business rather than on "philosophical" considerations. The costs, and therefore risks, incurred in software development are too high to use a new language without an analysis of its costs and benefits.

Network Application Programming System (NAPS) is an application programming environment written in C++ using OOP, and supporting forty programmers writing large applications. End user products written using NAPS as a foundation will be coming to market in the near term. In the proposal and request-for-funding stage for NAPS, C++ and OOP were cited as means of implementing increased complexity with less code and greater reliability, and therefore with lower cost and greater quality. When asked if we have achieved these goals the real answer is, of course, both yes and no. This paper describes our experiences building NAPS and its applications as a case study in the use of C++ and object-oriented design and programming in a production environment.

2. NAPS DESCRIPTION

NAPS is a system written in C++, using OOP, supporting network management applications on a multi-processor networked environment. NAPS uses a transaction model for programming and provides a graphical user interface. Using UNIX* System V Release 3, X Windows V11 and INFORMIX** as its current software base, NAPS provides a stable software interface to application programmers during a period of expected changes in the base software.

A single object hierarchy provides base objects for inheritance in applications. Part of this hierarchy is shown below.

3. PROJECT HISTORY

NAPS was prototyped in several phases using small groups of designers and programmers over a period of a year before it was fleshed out into a production environment. This prototyping had several goals. First to experiment with the technologies of OOP, windowing systems, and networking, and to research existing designs in those areas. The second was to translate the former into a design that could be easily learned and utilized by a larger group of programmers. The common part of the system to be used by all applications was called the platform. Finally the facilities of the platform were compared with application feature requirements to ensure that all features could be implemented.

Over a shorter subsequent period of six to eight months, a second phase evolved NAPS from a prototype to a production programming environment. The prototype objects were formalized into an architecture description and object descriptions. In parallel with this

---

* UNIX is a trademark of AT&T

** INFORMIX is a trademark of INFORMIX Software, Inc.

Figure 1. NAPS Class Hierarchy

```
Object
        DatabaseView
        DisplayObject
        IpcAccount
        Message
        GraphicalView
                DialogItem
                        Field
                        Form
                Frame
                        Window
        Server
                NetworkServer
                TransactionServer
        Service
                TransactionService
                DatabaseService
        Transaction
                DatabaseTransaction
                NetworkTransaction
```

activity was the setting up of a development environment providing the necessary compilers, debuggers, source code control, and problem tracking facilities. The prototype was then substantially reimplemented to meet the formal specifications. A subset of applications was implemented simultaneously with the platform to provide a sanity check from the application point of view. A skeletal "NAPS Developer's Guide" showed how to write sample applications using the NAP objects. At the conclusion of this phase, a subset of the NAPS platform, along with the development environment and how-to documents were made available to the development population at-large.

In the following eight months to the present time the cycle of applications development and testing has been in full swing. New platform features are being developed in parallel.

4. PROTOTYPING

Prototyping was an invaluable tool for testing out concepts and technologies in a small and risk-free environment before committing to them in a larger and therefore less flexible setting. During the course of the prototyping period several object-oriented systems both in C++ and in other languages were investigated.

4.1 The First C++ Program

The first small prototypes involved three programmers initially testing C++ itself, and loosely translating object-oriented designs from Smalltalk-80[1] to C++ and SunView[2] on Sun 3 workstations. A month of coding produced about 5000 source lines resulting in a single process showing a graphical representation of a network and visually simulating network control. The participants were new to C++, but agreed that the learning experience was relatively painless coming from a C language background, and that this new language added great power to C while seeming to add little run-time penalty. A second observation was that the C++ language/UNIX environment had very fundamental incompatibilities with the Smalltalk-80 environment: Smalltalk-80 is interpreted, not compiled; it runs in a single address space, not in the multiple address spaces of UNIX processes; it is dynamically extensible, not statically linked. Herein lies one of the challenges of designing UNIX-based, object-oriented systems in C++: tapping the power and elegance of the interpreted object-

oriented systems in an efficient implementation that runs under the UNIX Operating System.

4.2 The OOPS Based Prototype

Before the next iteration of prototyping, we acquired the Object-Oriented Programming Support (OOPS) package[3]. OOPS provides an implementation of a portion of the Smalltalk-80 object hierarchy in C++. For the prototype team this was like getting the answers to the test questions at the end of the book! In addition to providing the `Collection` hierarchy and `Class` system, OOPS was like a textbook in C++ techniques for operator overloading, static initializations of lists. It also had useful documentation including UNIX-style manual pages for many of the objects.

The next user-interface prototype was based on the OOPS object hierarchy. It made use of its class identity facilities, such as `Object::isA()` and `Object::isKindOf()`, which respectively identify the `Class` corresponding to an object instance, or whether an object is a instance of a derived class. Other features used from OOPS were the `ClassDictionary`, storing and reading in of objects, and the rich set of `Collection` classes. The time span to complete this prototype was also approximately a month, and resulted in about 8000 lines of source code.

4.3 The Client/Server/Transaction Prototype

The last prototype was greatly increased in scope. It was to encompass a multi-process software architecture; perform real, not simulated, network control; and support discrete, packageable user commands called "transactions". A team of 10 programmers worked two and a half months to design and implement a system that would approximate the performance of the target end-user system.

In moving out of the user interface arena and into a distributed client/server process structure, however, there were fewer sources of object-oriented design to draw from. Some designs changed the semantics of the language, such as a C++ member function call to sending an Inter-Process Communication (IPC) message[4], or defined new languages to specify message interfaces[5]. With the C++ version in use, however, this involved changes to the translator itself or using an unsupported language, and were rejected as strategies for NAPS.

Another question that came into play when dealing with multiple processes was whether objects should be passed in descriptive form from one process to another. Like Smalltalk, OOPS provided through its `Object::readFrom()` and `Object::writeTo()` functions and `ClassDictionary` object the ability to read or write object descriptions to or from a file stream. The file stream could easily have been extended to a network device or IPC mechanism. However, the sending of objects between processes for the prototype was rejected both for performance reasons: it seemed unsuitable for real-time message passing, and because it required a run-time class system which was not planned for NAPS. Instead, only a `Message` object could be passed between processes. For this concept, of course, many non object-oriented designs were available to draw from. One source, that was available was the Sun RPC/XDR package[6] for server applications and processor-independent data formatting. The `Message` object could encapsulate the machine independent data formatting as well as its contents. These `Message` objects must be compiled into both sender and receiver processes.

The application abstraction was a `Transaction` object that resided in a `TransactionServer`, and that was input-driven. Invoked as a result of a user command, it would make a series of low-level requests to various other servers. While waiting for the result of these requests it would return control so the server could respond to new

commands. A state variable allowed the `Transaction` to continue when it was called with the result of the pending request.

The user interface, derived from the first two prototypes was the end client for the `TransactionServer`. The graphical and user interface objects were ported from SunView[2] to the X-ray Toolkit[7] running on X Windows Version 10.4.

4.4 Prototype Results

The final prototype was important not only for the design validation, but for the testing of the interaction among a group of ten people programming inter-dependent objects.

In most ways, the programming environment was like a traditional C language environment. A central node contained the most recent version of the headers and libraries containing the class definitions. It became obvious immediately that the dependence on header files was much greater than in C. This has been noted in single person efforts[8]. More of the structure and design of a program was contained in header files in C++ than in C. With software in flux, programmers were more likely to conflict with each other through the headers. It seemed that two or three was the maximum number of people that could work together at one time in a single directory tree.

In the review of the application development environment, it came out that the transaction programmers were unhappy with the idea of event-driven programming. Existing subroutine libraries had to be taken apart to handle the state machine. The more conventional sequential programming model was more understandable and adaptable to existing software.

5. METHODOLOGY

After a working prototype was constructed the next task was to try and adapt it to a larger community of programmers. Since the introduction of a new language was a major risk, we looked for a conservative approach to the design and coding standards. With an eye towards the warnings strewn through the C++ literature about how dangerous C programmers are when they are let loose in C++ we stayed with more "C-Like" syntax, for example, staying away from operator overloading, and using pointers instead of references.

The platform was to be implemented first. In addition to providing functionality, the platform code would be a coding model for the application designers and programmers who were new to the language. The applications that followed would be extensions of the platform code.

5.1 C++ Programming Style

Style considerations in C++ were important to the success of the project. Strict discipline, with respect to style, needs to be used in order to make efficient use of the C++ programming language, especially for new C++ programmers.

5.1.1 Resource Allocation in Constructors Constructors in C++ can execute code. Constructors cannot return error codes. This is a problem however when errors are encountered in a constructor that does non-trivial initializations. How do errors propagate back to the caller?

In NAPS we mandated that constructors could do things that had fatal errors, i.e. errors that would cause the termination of the process. These include errors such as memory allocation, window allocation or other allocation that would cause the system to fail. However, non-fatal errors could not be produced in constructors. This allowed programmers to call new of an

object and if the call returned then the program would be guaranteed the object was allocated without checking for an error. This meant that constructors could not do initializations that would result in non-fatal errors such as opening a file. These were done in member fuctions that returned error codes.

5.1.2 Recursive Headers The use of recursive headers makes programming in C++ easier. Using recursive header files programmers just need to know the name of the header file containing an object. Without recursive headers all objects and recursively all sub-objects must be included.

There are usually two ways to implement recursive headers. By using a conventional system of preprocessor ifdefs and defines such as the following:

```
#ifndef MYOBJECT_H
#define MYOBJECT_H

      class header definition

#endif
```

The second method, used by NAPS, is to use the `nmake`[9] preprocessor that recognizes multiple includes and only uses the first include. Recursive headers were mandated in NAPS and proved to make the programming task much easier.

5.2 Documentation

Documentation of objects was a key aspect of the project. The interfaces between objects is defined in NAPS by an interface document, similar to a UNIX style man page. This allowed easy update and maintenance of object interfaces. It is important to note that the initial strategy was to design the objects and write the interface documents first and then code the objects. Then if any interfaces changed as a result of the coding the plan was to fold those changes back into the interface documents.

This initial plan evolved into: define the headers for objects, review the headers, code the objects and then write the interface document. This strategy saves time and allows room for reviews. An example NAPS interface document for the `SysLog` object is presented at the end of this paper.

6. NAPS PROGRAMMING MODEL

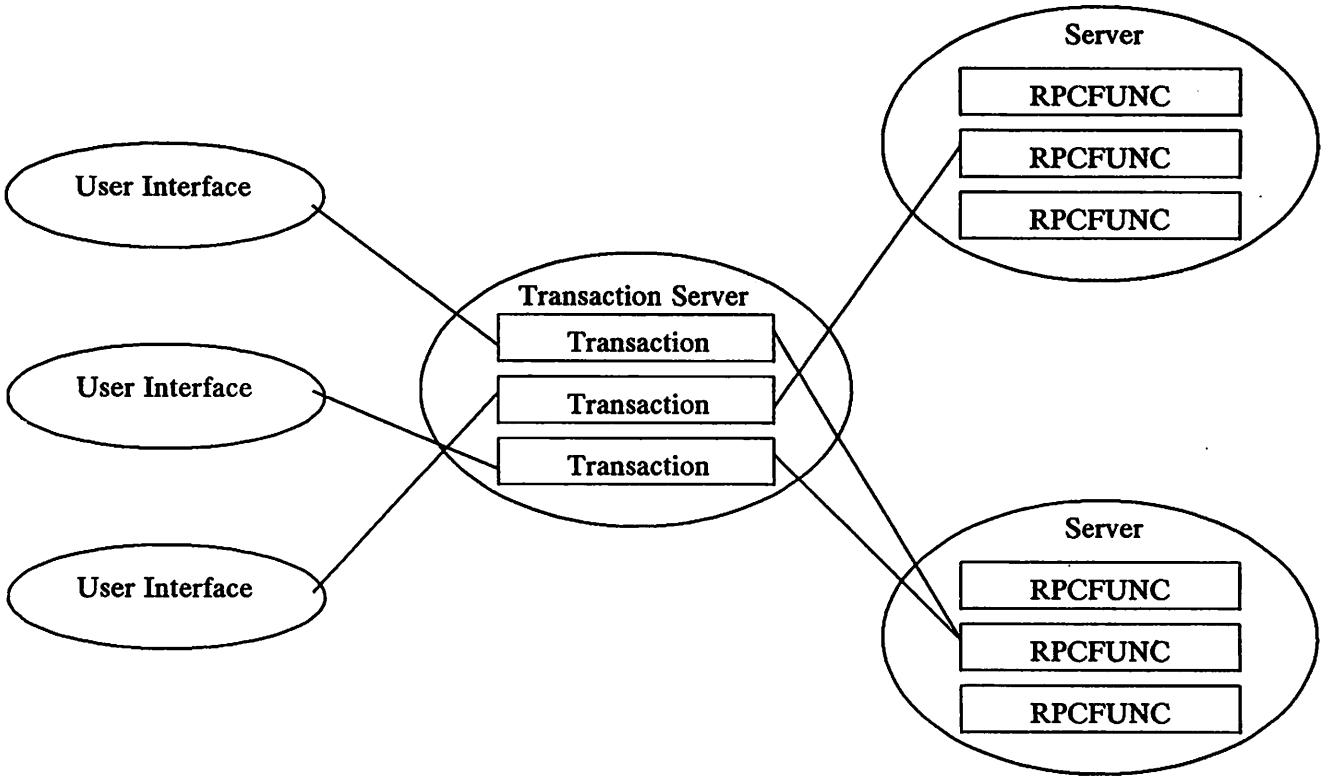
NAPS provides these basic units for application development: services, transactions, and user interfaces.

6.1 Services

Services provide the basic functions required for servicing end-user requests such as database records, network control, or real-time input using a client-server model. Services are implemented in server processes, and are accessed by clients using `Service` objects, which implement message protocol or IPC internally. Servers also allow event-driven and timer activated actions.

A service programmer creates an object derived from `Server` whose pseudo-code is as follows:

Figure 2. NAPS Programming Model




```
class Server : public Object {
      // private stuff
protected:
      // stores connections to clients
      IpcAccount* clientList[NUMBER_OF_CLIENTS];
public:
      // advertises a server name on the network
      int advertise(char* advertiseName);

      // associates request code with a message and function
      int registerService(int code, Message* msg, RPC_FUNCTION func);

      // waits for inputs and dispatches request messages
      int waitForInput(int timeout =-1);
};
```

as well as an object derived from `Service` object:

```
class Service : public IpcAccount {
      // private stuff
protected:
      int connectTo(char* serverName);
      int sendMessage(Message* msg);
      Message* receiveMessage();
public:
      Service(int fileDescriptor);
};
```

However the `Service` provided to the client shows only application-specific requests. No

message protocol is revealed to the client.

```
class DataService : public Service {
      // private stuff
public:
      status closeCircuit(char* circuitName);
      status openCircuit(char* circuitName);
      status rerouteCircuit(char* circuitName, int oldRouteId, int newRouteId);
      NetworkService(char* networkName);
};
```

6.2 Transactions

A transaction groups together low-level `Service` requests into an end user level command similar one that would appear in the product user manual. Transactions lack the full flexibility of a server in control, but provide a sequential programming model that a Server lacks. All NAPS transactions are derived from a base `Transaction` object:

```
class Transaction : public Object {
      // private stuff
public:
      int run(Collection* argumentValues);
      int exit();
      int deferSignal();
      int restoreSignal();

      void outputRecord(Collection *outputValues);
      void setOutputFormat(char *formName);
};
```

Each `Transaction` is like an autonomous program having its own input arguments, start, exit, and software interrupt handling routines. Its output is displayed record-by-record using a form name. Forms are also created by the transaction writer using a form language. Many transactions are grouped together into Transaction Server processes.

6.3 User Interfaces

User interface processes invoke transactions on behalf of users. They interpret forms, send transaction requests to transaction servers and display transaction output. All current user interfaces are part of the platform. In the future applications may develop their own user interfaces using the display library objects.

7. DEVELOPMENT ENVIRONMENT AND TOOLS

The C++ development environment is an incremental, not sudden, change from a C language development environment. Other than `cfront`, the C++ translator, no other new tools are strictly necessary. The NAPS development environment is transitional in that it is has a thin, growing layer of tools for C++.

7.1 nmake

As stated earlier, C++ forces greater reliance on header files than C. The UNIX tool `make` does not provide implicit header dependency rules. Additionally, to preserve the data encapsulation of class definitions NAPS headers contain nested includes of other header files. Nmake and its associated preprocessor simplify the bookkeeping of these nested headers both by automatically eliminating duplicate includes, and by searching through header files itself to

determine dependencies. An `nmake` Makefile for a C++ file is far easier to read than a corresponding `make` Makefile, and more reliable than some of the other methods such as `Imake` for maintaining up to date object files.

7.2 Debugging

Source level debugging is possible either with `dbx` in a BSD environment or `sdb`, but requires the user to be familiar with the C++ to C name translations. `Sdb++`, a C++ enhanced version of the System V `sdb`, is a welcome convenience since it knows some of the C++ syntax and naming conventions.

7.3 Needed Tools

NAPS programmers work without a browser. No tool shows the inheritance hierarchy of the system, or all the members of a certain class including those of the parent classes. This has the psychological effect of limiting the use of inheritance because it can't be seen easily on a terminal or printout when looking at class definitions. The programming environment should encourage the definition of deep inheritance trees with the maximum reuse of member functions of parent classes. Browsers and browser-like tools would make inheritance more accessible and easy to use.

7.4 Alternate Run-Time Strategies

The price of the greater efficiency of a compiled language C++ when compared with Smalltalk is a longer compile/test/debug cycle. The compile-time behavior of C++ is helpful during the initial coding phase of an object. It forces type-checking and catches many errors before the program is ever run. During debugging however, each small incremental change requires a long re-link, and tweaking the base objects can force long re-compiles and re-links. This problem is certainly not peculiar to NAPS, or C++ itself,[10] and is due both to the potentially large size of each object file and of each executable. Below are some areas for investigation in creating a better environment for C++ programming.

7.4.1 Shared Libraries UNIX System V Release 3 supports shared libraries of C code. Shared libraries are useful in decreasing object code size, and in allowing wholesale updating of system software libraries for applications without re-linking. The NAPS project has yet to devote the effort to making a C++ shared library, but it is not considered to be a difficult technical problem. However shared libraries, while they reduce the run-time size of a process, are restrictive in a debugging environment. In System V breakpoints cannot be set in a shared library, and stack traces don't work. Shared libraries in their current form are probably most useful for end-user software distribution rather than for software development.

7.4.2 Dynamic Loading of Objects Dynamic loading of objects allows objects to be compiled incrementally and brought into a running process when it is referenced. The Andrew System[11] has implemented this for several processors including VAX*, Sun, and IBM RT. In addition to providing a possible solution to the compile time problem it is potentially powerful from an application design point of view as well.

8. TRAINING

8.1 Learning C++

An implicit expectation of the C++ language is to have an easy transition from C both in software reuse and "personnel reuse". In the near future, however, any project using C++ has to deal with the language learning curve. For example, on our project out of forty programmers not one had prior experience with C++. Newcomers to NAPS were handed the

* VAX is a trademark of Digital Equipment

C++ text book[12] and sent to a one week course. In retrospect, the language itself was not as much of a problem as we might have feared, because a lot of the NAPS code "looked like" C because of the style guidelines.

8.2 Learning Object-Oriented Design

While the concept of data-hiding is very strongly enforced in the corporate R&D environment, other object-oriented concepts such as inheritance are much harder to grasp. Most application programmers did not define their own class sub-hierarchies but preferred to program in straight-line code and use predefined classes. As stated above, some of this is to be blamed on a primitive development environment that makes it difficult to see the inheritance hierarchy. More importantly, however, is the lack of object-oriented design material currently available for C++. This situation will undoubtedly improve over the coming years as more literature in C++ is published.

9. LANGUAGE ISSUES

9.1 Virtual Destructors

In the NAPS transaction system and user interface library, heterogeneous lists of related objects are maintained and deallocated using pointers to a common base class. Graphics objects such as in the following hierarchy are a common example of this construct.

```
Object
      DisplayObject
            DisplayBitmap
            DisplayShape
                  DisplayLine
                  DisplayPolygon
```

The classes derived from `DisplayObject` allocate additional resources. However, when the following code is executed:

```
DisplayObject* op = new DisplayBitmap;
delete op;
```

only the base `DisplayObject` destructor is called, leaving dangling resources if any calls to `delete` are required in the derived destructor. The most glaring example of this behavior is a "memory leak" caused by not freeing some memory as a result of the correct destructor not being called.

A virtual destructor allows the correct derived class destructor to be determined and called at run-time. This works the same way as other virtual functions in C++. The example below shows two related classes with a virtual destructor.

```
class DisplayObject : public Object {
public:
      // the destructor is declared virtual
      virtual ~DisplayObject();
      .
      .
};

class DisplayBitmap : public DisplayObject {
public:
      // all derived classes have virtual destructor
      ~DisplayBitmap();
      .
      .
};
```

There are several strategies for using virtual destructors. The one we opted for in NAPS was to declare the destructor for the base class Object to be virtual.

9.2 Task Library

The task library supplied with C++ is used in NAPS as a configuration option for Transaction Servers. If transactions are to be invoked often, and require relatively few database requests, there could be a performance improvement in not having to fork and possibly exec a UNIX process. Nevertheless, the tasking system is not a reliable environment to debug programs from scratch. NAPS Transaction Servers can be derived from a base class that runs in a single-threaded mode, or forks and execs for debugging, but derived from a different base class for production that uses the C++ task library. This is made possible by restrictions on transaction coding to not use UNIX system calls or signals, and not use static data. The member functions of the `Transaction` and `Service` class can be implemented either with streams or task queues, signals or task messages.

Unfortunately, the switch to the task library often brings up problems that would not occur in a fork/exec situation such as stack overflow, which does not occur on a single UNIX process stack, and "memory leaks" on task termination which would be cleaned up automatically on UNIX process exit. These problems are handled in NAPS, but nevertheless, switching from a process based transaction model to a task based one is time consuming to debug for the application programmer.

9.3 Exception Handling

One opportunity for reduction of code si[illegible] that NAPS has not attempted is an exception handling package. Although NAPS has [illegible]jects to report error conditions, it does so by reporting the line number and a text strin[illegible]escribing the problem. Much of the transaction code reads like the following:

```
if ((result = deviceVie[illegible]query(name)) == ERROR)
      // report error
      return ERROR;
```

The above method of coding is tedious and error prone. Most production code has close to half of the non-commented source lines devoted to catching these error conditions. An elegant solution to this problem is to provide member functions in the task objects that are called for a given exception. This would mean that most application level programmers would never have to check for error returns from functions. Instead they would inherit an exception handler from the base `Transaction` class to do application-specific cleanup.

9.4 Problems with Software Reuse

Although software reuse is a commonly cited advantage of object-oriented programming, the use of a single object namespace has negative influence on the use of libraries of objects in C++ from outside sources. The object namespace problem stems from the early work in object-oriented systems. where all programming was done within a single namespace. Libraries of objects cannot be used in a process unless the names of the objects in the library do not conflict with any object names within the process namespace.

One price projects using C++ will pay until the language matures is the lack of available sample source code. One of the best features of UNIX has always been the ability to browse through the source code at will to look for examples. C++ just does not have that body of code yet. In fact, C++ makes it harder to have a body of available code to look at because of the various coupling mechanisms that inheritance forces. However, several publicly distributed source packages such as OOPS[3], Andrew[11] and Interviews[13] have been valuable sources of OOP code. None of these packages has provided objects that would fit into our class hierarchy without modification. C++ comes standard now with `String` and `task` classes. These packages do not integrate into our system because they make their own assumptions. For example the standard task library prints errors to standard out, but in NAPS there is no standard out and the only way to provide error output is to use the NAPS `SysLog` object.

9.5 Field Maintenance

Since private object data is defined in the header files, and determines an object instance's size, binary compatibility between different versions of NAPS will be more difficult to maintain in C++ than in a C language product. A one byte change in a base class of NAPS will render the entire release incompatible with an earlier version.

9.6 Callbacks vs Command Objects

The first version of the User Interface class hierarchy was a set of classes that interfaced with the X Window Xt toolkit. These classes implemented an event driven control mechanism for these classes. The control portion was implemented via callback routines (i.e. routines that are called when a specific event occurs) that were pointers to class member functions. These were error-prone to program because the syntax was difficult, and it required disabling type-checking. Also virtual functions, whose addresses cannot be taken, could not be specified as callback functions.

The new version of the User Interface class hierarchy uses a `Command` class for handling user driven actions[14]. The callback mechanism is handled instead by a virtual `Command::doIt()` member function. Each derived `doit()` makes use of the instance data in the derived object rather than using arguments.

9.7 Multiple Inheritance

The most recent version of C++ has implemented multiple inheritance. Multiple inheritance was not considered when NAPS was being prototyped and implemented, since C++ did not support it at the time. In looking back over the the NAPS Class hierarchy, there are a few key places where it would have been easier to make use of multiple inheritance rather than single inheritance.

10. METRICS

In this section we present some metrics that are valid for the current state of NAPS. Since NAPS is not completed yet these are preliminary figures and must be taken in the context of the comments in this paper.

The total number of classes in NAPS is currently about 350 and growing. The number of

classes per process ranges from a low of 22 to a high of 135. On a 3B2 an average Transaction Server contains 74 classes and has a static size of 190 Kbytes. An average Server contains 38 classes with a size of 125 Kbytes. Both of these sizes are about 25% greater than what they should be because NAPS is still compiled using C++ 1.2 options which makes all class virtual tables static in each file. Version 2.0 should bring an automatic reduction in process size. Below are some non-commented source line counts for NAPS sub-systems.

Figure 3. Non-Commented Source Lines in NAPS

Sub-system	Headers	Code
Platform	4593	61235
Database Applications	1426	8808
Network Application 1	548	19744
Network Application 2	368	4743
Total	6935	94530

This translates into an average of about 300 lines per class throughout the system.

11. EFFECTIVENESS OF OBJECT-ORIENTED PROGRAMMING IN C++

11.1 Graphical User-Interfaces

Among the different software applications, graphical user interfaces have perhaps the most examples of OOP in both academic or commercially available systems. Some of these include Smalltalk-80[1], MacApp[14], Andrew[11], InterMedia[10], and InterViews[13]. With this depth of sources, it is easy to model the problem ourselves using OOP, or stated another way, more difficult to design without OOP.

11.2 Working in a Non Object-Oriented World

The strength of the C++ language is its ability to implement object-oriented facilities on conventional software and hardware architectures. Application designs such as NAPS have the same requirement. The lower levels on which NAPS is built such as the operating system, relational database, windowing system, and perhaps previously written applications code are not object-oriented. One of the first decisions in coming up with a design was to decide which entities should not be modeled as objects. NAPS has non-object representations of many entities. For example, fundamental types such as `int`, `long`, and `double` are not defined as instances of classes. The database contains tables and records rather than objects. Tables and records are represented using a data abstraction model, in which load and store operations are `virtual` and redefined for each table type. Existing application code also made it difficult to use a Network Device object from which applications would inherit specific models.

11.3 OOP as an Implementation Tool

Although NAPS has not been modeled entirely with objects, data encapsulation and inheritance are powerful implementation techniques even when used in a portion of the system. The `Server` and `Transaction` systems make heavy use of inheritance, and guarantee that the rest of the system, which is largely derived from them, will follow the same low-level protocols.

12. CONCLUSIONS

Unfortunately we do not have the "magic metrics" in hand to show that the job has been done better in half the lines of code. NAPS still suffers from many of the common problems associated with other efforts its size. However, NAPS is a more modular and extensible

system under C++ than it would have been using C. The language requires more human discipline at the design level and less at the programming level. This benefit is seen especially in a larger setting. On the negative side NAPS cannot be shown to be a significantly smaller system than its predecessor. Like several other implementors in C++ we feel that it is the "better" way to do things while we lack hard measurements to back up our case. We have found, however, that C++ is a reasonable platform for programming of real, production software in a medium size environment.

13. ACKNOWLEDGEMENTS

Thanks to Jay Armstrong who participated in the initial prototyping work, Barry Books and Mark Tuomenoksa for their work on the Client/Server system, Amnon Janiv for work on the transaction system, and Brad Nohejl for his design and implementation of the database system.

REFERENCES

1. A. Goldberg and D. Robson, *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983
2. "SunView Programmer's Guide", Sun Microsystems, Inc., 1986
3. K. E. Gorlen, "An Object-Oriented Class Library for C++ Programs", Software-Practice and Experience, Vol. 17(12), December 1987
4. L. A. Call, D. L. Cohrs, B. P. Miller, "CLAM - an Open System for Graphical User Interfaces", OOPSLA '87 Proceedings
5. M. B. Jones, R. F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems", OOPSLA '86 Proceedings
6. "External Data Representation Protocol Specification", "Remote Procedure Call Protocol Specification", Sun Microsystems, Inc., 1986
7. "Programming With the X Window System", Hewlett-Packard, 1986
8. T. A. Cargill, "Pi: A Case Study in Object-Oriented Programming", OOPSLA '86 Proceedings
9. Glenn S. Fowler, "The Fourth Generation Make", Proceedings of the Summer USENIX Conference, 1985
10. N. Meyrowitz, "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework", OOPSLA '86 Proceedings
11. The Andrew System Programmer's Guide to the Andrew Toolkit, Information Technology Center, Carnegy Mellon University, January 1988
12. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986
13. "InterViews Reference Manual Version 2.1", Computer Systems Laboratory, Stanford University, 1987
14. Kurt J. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden, Hasbrouck Heights, N.J., 1986

NAME

SysLog - System Log

SYNOPSIS

```
#include        "SysLog.h"

typedef long lMask;

extern SysLog*          Sy;

class SysLog : public Object {

public:

        void    setMask(lMask mask);
        void    print(lMask mask, const char * moduleName, char* format ...);

        SysLog();
        SysLog(lMask mask);
        ~SysLog();
};
```

DESCRIPTION

SysLog is an object that takes "printf(3)" style format specifications and arguments, composes the resulting text and stores the result into a log file. A mask is compared against a global mask to determine candidacy for logging and to determine which types of messages to be logged.

There is usually only one SysLog per process. The global SysLog pointer, *Sy, is used to access the global SysLog. All NAPS processes have a preallocated SysLog pointed at by Sy.

PUBLIC MEMBER FUNCTIONS

void setMask(lMask mask)

Set the mask for this SysLog. There is normally only one SysLog object per process. This routine sets the mask for this process. The list of possible values for mask and their usage is:

SY_ENTER	entering a function or process
SY_EXIT	exiting a function or process
SY_NEW	allocating memory
SY_DELETE	deallocating memory
SY_INFO	general information
SY_ERROR	error

The default setting for the mask is all possible values enabled.

SetMask should only be used for debugging a process. In a running system the mask will be set by an external process. The mask can be changed dynamically in a running process.

void print(lMask mask, const char * moduleName, char* format ...)

Outputs the given arguments into the default system log. The mask is and'ed with the global mask and if the result is non-zero the message is output. If the result is zero this functions returns without outputing the message.

The define FILEID should be used for the moduleName argument. This define expands to the current filename and current source line number.

If Sy is a pointer to a SysLog object and i is an integer and str is a string then the following is legal:

```
Sy->print(SY_DEBUG, FILEID,
"This is a variable %10x with a good string %s", i, str);
```

CONSTRUCTORS/DESTRUCTORS

SysLog()

Initialize a SysLog object with default values. There is a constructor of this type called at the beginning of any NAPS process to initialize the global Sy pointer.

SysLog(lMask mask)

Initialize a SysLog with the given mask.

~SysLog()

Delete a SysLog.

USAGE

include file	SysLog.h
library	SysLog.a
module	platform/SysLog

SEE ALSO

NAPS: System Logging Manual

WARNINGS

Syslog cannot be called in object constructors that may be static.

Data Level Parallel Programming in C++

Thomas M. Breuel *

Abstract

We describe the design and implementation of a programming system for data level parallel programming in C++.

C++ is particularly suited to implementing data level parallel programs because it allows us to extend the syntax of the language to new data types, because it provides operator overloading, and because it allows us to use a reference counting memory allocation scheme. Furthermore, C++ encourages the use of abstract data types, which makes it easy to provide efficient implementations of the programming system with identical software interfaces on both serial and parallel machines. To allow us to formulate algorithms largely independent of the specific implementation of the communication primitives for specific parallel machines, we take advantage of C++ function and operator overloading.

We present examples of data level parallel programs written in our embedded language to demonstrate the expressive power and elegance of using C++ for data level parallel programming.

A serial implementation of our language exists. We compare it to other data level parallel languages such as APL, *Lisp and C*. We will also discuss a number of idioms and techniques that we used, and point out issues in C++ language design that came up during the implementation.

We see our work as an interesting case study in using C++ as a tool for building embedded programming languages, and as providing a useful and practical tool for data level parallel programming. We are currently working on improving the serial implementation and writing a parallel version of the embedded language for the Thinking Machine Connection Machine.

1 Introduction

With the advent of fine grained SIMD ("Single Instruction, Multiple Data") machines has come a renewed interest in a form of programming that has come to be called "data level parallel programming", but whose origins can be traced back to languages such as APL [Ive62]. In particular, for the Connection Machine, a SIMD machine with a hypercube interconnection network and up to 2^{16} one bit processors, several libraries and programming language extensions exist to allow a programmer to make use of the hardware.

From our point of view, these existing programming systems have several major disadvantages. Low-level libraries such as the Paris library[Thi86a] for CommonLisp or C are too inconvenient to use for the application programmer. Language extensions for CommonLisp and C, called "*Lisp" and "C*"[Thi86c][Thi86b], respectively, are non-standard and rely to a significant extent on vendor-specific hardware features.

*Author's address: MIT Artificial Intelligence Laboratory, Room 711, 545 Technology Square, Cambridge, MA 02139, USA. The author was supported by a fellowship from the Fairchild foundation.

Programming languages such as CommonLisp[Ste84] and C[KR78] require language extensions to accomodate data types and operations for data level parallel programming because they either do not give the programmer sufficient control over storage allocation and de-allocation to make efficient use of the limited per-processor resources on highly parallel machines, or because their syntax is so inflexible that specifying parallel operations is cumbersome and counterintuitive.

For example, if we wanted to introduce a quaternions data type (a kind of number that can be represented by four real numbers) into CommonLisp, we could extend the syntax of the language to make arithmetic operations on quaternions appear syntactically identical to operations on, say, integers[1]. However, we could not avoid the run-time overhead of type dispatching and of storage allocation and de-allocation every time we carry out arithmetic with quaternions. Conversely, in C, we would have no trouble making sure that not a single byte of storage gets lost; however, we would have to use function call syntax to express arithmetic operations on our quaternions and handle memory allocation/deallocation explicitly everywhere.

The programming language C++[Str86], a derivative of C, remedies these problems. It allows the programmer to extend the existing syntax of the language to new data types, and gives him complete control over all aspects of storage allocation and de-allocation. It is therefore ideally suited to implementing a data level parallel programming library.

Our primary goal has been to develop a useful tool for data level parallel programming and to provide a system that encourages experimentation with new data level parallel primitives. Our approach has been influenced significantly by the design of the Connection Machine. This is not only because we will be using our programming system to program the Connection Machine, but also because the Connection Machine is the prototypical data level parallel machine.

2 What is Data Level Parallel Programming?

Data level parallelism is a restricted form of parallelism in which (conceptually) different processors carry out identical, independent operations on the elements of large, uniform data structures(see also[Thi86b][HS86]). An example of a data level parallel algorithm is the component-wise addition of two vectors.

Data level parallel algorithms are easier to design than general parallel algorithms because they do not require any explicit synchronization. Rather, synchronization is implicit at the completion of each primitive operation. The existence of a good data level parallel algorithm to solve a particular problem is a problem intrinsic property; data level parallelism is not a panacea for parallel programming. It is, however, one of a number of useful conceptual tools in the design of parallel algorithms[2].

A machine designed specifically for executing data level parallel programs is the Connection Machine[Hil85][Thi87]. The Connection Machine is a fine grained SIMD machine, meaning that a single instruction stream is broadcast to all the processors in the machine and operates in parallel on data in each of the processors' memories. In the CM-2, the latest model of the Connection Machine, each processor is a bit-serial one-bit processor. In

[1] With some difficulties. We find that the Common Lisp language is significantly flawed in that it discourages extending the existing syntax and functions to new data types.

[2] Other useful concepts are those of pipelining and independent branches of computations. These tend to come into play at the level of larger functional units of algorithms than data level parallelism.

addition, groups of 32 processors share a single 32bit floating point co-processor.

However, while a fine grained SIMD machine is perhaps the most obvious and straightforward design for a machine built to execute data level parallel programs, it is not necessarily the most economical. In particular, when the data types to be used in data level parallel algorithms are larger than the word size of the individual processors, it is probably more efficient to use a smaller number of more complex processors and simulate several "virtual processors" with each physical processor, in order to give the programmer a programming model of a large uniform machine. Once virtual processors are used, an implementation of a data level parallel algorithm can benefit further from a MIMD architecture because operations in individual simulated processors may take different amounts of time, and a MIMD architecture executing a data level parallel algorithm can defer synchronization until the end of each data level primitive operation.

Data level parallel programming can also have significant benefits when used on vector architectures like the Cray series of computers, or, due to instruction caching and data prefetch effects, even on conventional serial microprocessors. An extreme example of a serial machine specifically designed to execute data level parallel programs efficiently is the cellular automata machine (CAM) designed by Toffoli[Tof84][3].

But apart from issues of efficiency, data level parallel programming also enforces certain kinds of abstractions that are analogous to the use of operator notation in favor of index notation in mathematics. Rather than operating on large collections of data by iterating over index sets, data level parallel programming requires the use of operators that "operate on all the data at once"[4].

3 The Library

Regardless of whether a data level parallel algorithm runs on a fine grained machine with one physical processor per data element, on a coarse grained machine with many data elements per physical processor, or on a serial machine, we will use the following terminology. The basic data structure of data level parallel programming is a "pvar" [5]. A pvar is a data structure very much like a vector, i.e. a collection of items of identical type indexed by a subset $\{0, 1, \ldots, n\}$ of the integers. A "processor" is an element of the index set.

Each pvar has associated with it a length which specifies the range of elements of elements that take part in operations involving this pvar. Operations involving several pvars check whether the pvars participating in the operation have compatible lengths.

Pvars come in several types that correspond to C++ data types. The library defines the types `pbool`, `pbyte`, `pshort`, `pint`, and `pfloat`. It is guaranteed that each of the data types `pbool`, `pbyte`, `pshort`, and `pint` is at least as large as the preceeding one in the list. Furthermore, a `pbyte` is at least 8 signed bits large, a `pshort` is at least 16 signed bits large, and a `pint` is at least 32 signed bits large. An element of a `pint` is guaranteed to be able

[3] The next generation of CAM[TM85] will use 256 processors in parallel and is expected to execute data level parallel programs that operate on individual bits with grid communication at speeds comparable to the CM-2, at a fraction of the cost.

[4] This abstraction is certainly not a benefit that is exclusive to the use of data level parallel programming tools. Index free computation over arrays has been the main attraction of the APL programming language. Modern introductions to programming such as Abelson and Sussman[AS85] and modern approaches to numerical methods as presented in Halfant and Sussman[HS87] encourage similar abstractions.

[5] Our terminology is similar to that used with the *Lisp language and APL programming languages.

to hold the index of an element of a pvar. Corresponding C data types are provided under the names `Bool`, `Byte`, `Short`, `Int`, and `Float`.

Pvars are declared and initialized as follows:

```
// pvar of length 0
pint v1;
// pvar of length 8
pint v2(8);
```

pvars can also be assigned and converted easily as follows:

```
// set every element in v1 to 3
v1=3;
// convert apfloat to a pint
v1=pint(apfloat);
// set every element in v1 to
// its processor number
v2=address(8);
```

The contents of a pvar can be examined and changed using the `set`, `ref`, and `print` member functions. (These member functions can even be used interactively in a debugger such as gdb+).

```
x.print();
    <pint: 3 1 9 2>
x.set(2,77);
x.print();
    <pint: 3 1 77 2>
printf("%d\n",x(0));
    3
```

Pvars can be used as components of structures, can be assigned to, and can be subscripted like arrays. All the arithmetic and logical operations of the C++ programming language have counterparts for pvars that operate componentwise on the elements of the pvars. Pvars and scalars can be combined freely in expressions. A number of communication primitives to move data between processors are also supported. These are described in more detail in Sections 5.3 and 7.2. Here are some simple examples:

```
struct pcomplex {
    pfloat real,imag;
};

pcomplex operator*(pcomplex x,pcomplex y) {
    pcomplex r;
    r.real=x.real*y.real-x.imag*y.imag;
    r.imag=x.real*y.imag+x.imag*y.real;
}

pfloat scale(pfloat x) {
    float s=(x*x).reduce(op_sum);
    return x/s;
```

```
}

pfloat fiota(int n) {
    pfloat x(n);
    for(int i=0;i<n;i++) x[i]=(float)i;
    return x;
}
```

Two control structures are provided, "`where(`*pbool*`)` *statement*" and "`all` *statement*". These restrict operations on pvars occurring dynamically withing the *statement* to those processors in which the *pbool* is `TRUE`, or all processors in the case of `all`. While these constructs are useful on occasion, we encourage the use of compression operations (see section 5.3) and the `operator?:` to replace them whenever possible.

4 The Implementation

The data level parallel programming system described here is implemented in GNU C++ [Tie88] using the M4 macro preprocessor[KR77]. The output of the M4 preprocessor is a set of C++ source files and header files. Programs using the data level parallel library do not need to use the M4 preprocessor; they can include the header files generated by the M4 preprocessor using the standard C++ preprocessor.

All pvar data types (`pbool`, `pbyte`, `pshort`, `pint`, `pfloat`) are derived from a base class `pvar` that implements allocation and deallocation and declares a number of virtual member functions. An object of type `pvar` is actually just a structure containing a pointer to the data and reference count associated with the pvar[6]. We will discuss this in more detail in Sections 5.1 and 6.2.

We used the M4 macro preprocessor to write code templates for classes of operations such as "binary arithmetic operators" and instantiated these code templates for specific data types and operators. This use of the M4 macro preprocessor reduced the amount of work involved in recreating for pvars all the arithmetic and logical operators of C++ that work on scalars significantly. See Appendix B for an example.

The `where` and `all` control structures are implemented as C++ preprocessor macros. Essentially, they have to modify a variable, the `CSS` ("Currently Selected Set"), before and after executing a statement. They expand into the head of a complicated `for` statement. This trick allows us to use the same syntax for these new control structures as for built-in control structures like `if`.

5 Where C++ Helped

5.1 Per-processor Memory is Limited

Per-processor memory tends to be limited in data parallel programming. Efficient and immediate reclamation of unreferenced data structures ("garbage") is therefore very important. Indeed, one of our main motivations for writing a data level parallel programming language in C++ was that storage management in *Lisp, the data level parallel programming language most commonly used at MIT, is very cumbersome. *Lisp cannot use the

[6] This implementation of reference counting is similar to the one described in Section 6.9 of [Str86].

CommonLisp garbage collection facilities to reclaim storage because garbage collections would occur very frequently (due to the limited amount of per-processor memory on the Connection Machine). The overhead of a invoking full Lisp garbage collection to reclaim data structures residing in the Connection Machine's memories is unacceptably high.

*Lisp attempts to solve this problem by providing a number of macros to be used instead of standard Common Lisp special forms in an attempt to implement a form of mark and release type memory allocation. This solution is, however, very unsatisfactory. Common Lisp does not really support the notion of providing replacements for standard operations like copy-on-return, assignment, or initialization, and the *Lisp user is forced to live with two different sets of language primitives that interact in non-obvious ways. For example, to define functions that use pvars as arguments or return a pvar, the `*defun` macro must be used, otherwise the Common Lisp `defun` special form is permissible. To introduce variable names that will hold pvars, the macro `*let` must be used, whereas for non-pvar bindings, the `let` special form is permissible.

*Lisp violates several of the principles of the Lisp programming language. It introduces the notion that types are lexically associated with variable names[7], it uses objects with dynamic lifetime, and it requires the programmer to use explicit calls to memory management functions for data structures that cannot be allocated on the stack.

In our implementation, we use reference counting for management of the per-processor memory. A reference counting memory management strategy can be implemented transparently in C++, without introducing new constructs or primitives foreign to the language. As in the *Lisp language, the programmer is required to allocate and deallocate pvars explicitly when they are used in contexts where the compiler cannot determine their lifetime easily. However, as opposed to *Lisp, this is entirely analogous to the allocation and deallocation of any other complex data structure in C or C++ and therefore does not break with the conventions of the language.

5.2 Assignment by Copying is Expensive

Reference counting also allows us to implement shallow assignment. Shallow assignment improves performance significantly on a serial implementation. However, one might object that shallow assignment is a break with the conventions of the C++ programming language and may lead to un-obvious behavior from the programmer's point of view. We have found so far that in most cases shallow assignment can replace assignment by copying without changes to the meaning of the program, since most operations on pvars will produce new pvars anyway.

In principle it is possible to implement a "copy-when-needed" scheme that provides the semantics of copy-on-assignment but avoids unnecessary copy operations. So far, we have not experienced the need for this, and we fear that the overhead might be noticeable.

5.3 Machine Dependence of Communication Primitives

Data level parallel programming relies on a number of communication primitives[LLM+88] [Thi86b]. Our library provides primitives for moving data around between different processors, implemented by the `operator[]`, and for scanning[8] and reduction oprations, implemented by the member functions `scan` and `reduce`. The `operator[]` takes either a pbool

[7] Note that *optional* declarations *are* lexically associated with identifiers even in Common Lisp.

[8] Also referred to as "parallel prefix". Our terminology agrees largely with APL's.

or a pint as an argument. When a pvar is subscripted by a pbool, a pvar of the same type is constructed consisting only of those items stored in processors where the pbool is TRUE. We refer to this operation as "compression". When a pvar is subscripted by a pint, a pvar of the same type is constructed whose length is the same as the subscripting pint, and in which each element is selected from the processor specified by each element of the pint. We refer to this operation as "indexing".

Here are some straightforward examples of the use of compression and indexing:

```
// permute the variable x
// with a fixed, given permutation

try(pfloat x) {
    pint perm(3);
    int p1[]={3,2,1};
    perm=p1;
    return x[perm];
}

// determine how many applications
// of the permutation perm it takes
// to get back the original variable ind

order(pint ind,pint perm) {
    int n=0;
    pint last_ind;
    do { last_ind=ind; ind=ind[perm]; n++; } while(ind!=last_ind);
    return n;
}

// sum the values of the 4nhd

pint sumnhd(pint v) {
    // north &c. are either special address classes or
    // global constants of type pint
    return v[north]+v[south]+v[east]+v[west]+v;
}
```

On many parallel machines, some communication patterns are more efficient than others. For example, on the Connection Machine, communication that takes place on a two dimensional grid is significantly faster than general permutations. Often, it is also advantageous to pre-compute some additional information about a specific communication pattern and associate this additional information with the communication pattern itself; for example, it is frequently desirable to replace a particular communication step by two steps with randomly chosen intermediates in order to obtain good expected performance of an algorithm.

To simplify experimentation with different kinds of communication primitives in a data level parallel program, the operator[] is overloaded for different kinds of arguments that represent different communication patterns. The algorithm itself, formulated in terms of the operator[], remains unchanged then when the types of the arguments that describe

the communication operation are changed. Thus, to change an algorithm to use a user-defined `distributor` class instead of the built-in send primitive, it is sufficient to change the definition of the pvar used for the distribution, not the algorithm itself:

```
oldAlgorithm(pint x) {
    pint distribution=makeDistribution(x);
    // some algorithm involving distribution follows
    ...
}

newAlgorithm(pint x) {
    distributor distribution=makeDistribution(x);
    // some algorithm involving distribution follows
    ...
}
```

Overloading of `operator[]` could also be used to eliminate the explicit definition of a scan member function from the library.

5.4 Assigning to Subscripted Pvars

It is convenient to treat pvars syntactically like arrays. Unfortunately, it is difficult on some parallel machines to treat references to per-processor memory as pointers; i.e. the per-processor memories are not mapped into the host machine's address space. We provide some syntactic sugar that lets us write expressions such as:

```
pint temp(10);
temp=0;
temp[3]=1;
temp[9]=1;
```

This is implemented by having the `operator[](int)` return an object of type `lvalue_pvar` and overloading the assignment operator.

A generalization of this method allows assignment to pvars with indexing[9]. For example:

```
pint rp=random_permutation();
y[rp.inverse()]=x;
assert(y==x[rp]);
```

I.e. if `rp` is a permutation of the processors, the expressions "`y[rp.inverse()]=x`" and "`y=x[rp]`" are entirely equivalent[10]. However, the assignment form allows collisions. How these are handled depends on the type of the argument to `operator[]`. By default, the behavior is unspecified. The implementation of permuted pvars on the left-hand side of assignment operators is entirely equivalent to the implementation of subscripted pvars on the left-hand side of assignment operators.

[9] This operation is equivalent to the Paris send primitive

[10] The member function inverse returns the inverse of a permutation.

6 Where C++ Didn't Help

6.1 Compiler Optimization

Perhaps the conceptually most displeasing disadvantage of our approach is that because operations are implemented as library subroutines (regardless of the syntactic sugar provided for them), the C++ compiler cannot perform optimizations that a dedicated compiler could. A C++ compiler can perform constant folding, move expressions from loops, or eliminate dead expressions, to name only a few optimizations, for arithmetic expressions involving, say, the data type "int", because it knows a great deal about the properties of 32 bit, two's complement numbers. It has no such knowledge about user-defined arithmetic data types. A dedicated compiler for a data level parallel language could know about such properties and take advantage of them.

Practically, this objection is of little significance, since most special purpose languages, such as C*, tend to lack good optimiziers anyway, simply because not as many resources go into their development. Nevertheless, it would be desirable if the programmer could declare to the compiler that a user defined data type behaves "just like a built-in type", in our case, that the arithmetic properties of, say, pint's are the same as the properties of ints. Less promiscuous declarations would state that certain functions or operators are side-effect free, that arguments are not modified or are "constant references", or that certain algebraic properties such as commutativity or associativity hold for a given operator.

In passing, we would like to mention that two particular kinds of optimization would probably improve performance of the serial implementation of our library significantly: elimination of index variables and loop merging. While the former is straightforward to implement (and will be included in upcoming versions of the GNU C++ compiler), the latter requires considerably more sophistication and global optimization strategies.

6.2 Storage Allocation

Currently, the storage for pvars is allocated using the `operator new` in the serial implementation, and a general, `malloc`-like storage allocator in the parallel implementation[11]. In principle, the overhead of calling a general storage allocator could be avoided in many cases and a stack allocation scheme be used instead. A stack allocation strategy would also reduce memory fragmentation, which is particularly important in the parallel implementation where per-processor memory is limited. Unfortunately, the C++ language definition does not specify how a constructor can determine when stack allocation can be used for additional storage associated with a class object.

We also feel that the use of reference counting to achieve immediate de-allocation of unreferenced per-processor memory and the use of double indirection to achieve reference counting are not the best possible strategies for memory management. Experience with implementations of Smalltalk has shown that using direct pointers and generation scavenging storage reclamation are more efficient and can also be used to reclaim memory almost as soon as it has become garbage[UP84][Ung87].

[11] See also the `string` class defined in [Str86], Section 6.9.

7 Some Unresolved Issues in Languages for Data Level Parallel Programming

7.1 Multidimensional Pvars

We would like to provide the same kind of general treatment for multidimensional pvars in our library that multidimensional arrays receive in APL. Unfortunately, the ease with which multidimensional arrays are manipulated in APL derives largely from its specialized syntax. C++ does not even provide multidimensional forms of the subscripting operator[12].

More importantly, in APL, most operations on multidimensional arrays have a straightforward and obvious implementation. Arrays can simply be represented as vectors with some associated information about their shape; reshaping a 3×5 array into a 5×3 array does not require any actual data to be moved. However, the same approach does not work on parallel machines. If we assign an n-tuple of integers as a multidimensional address to each processor, reshaping an array actually involves inter-processor communication in order to assure that data items with the same multidimensional addresses reside in the same processor.

We are not sure yet how to support multidimensional pvars cleanly. Support in the *Lisp language is rather *ad hoc* and machine specific. By making multidimensional pvars derived classes of pvars, and using the methods described in Section 5.3, the same kind of functionality can be provided easily and more cleanly as an extension to our library.

7.2 Efficient Communications

A great variety of interconnection schemes for the processors in parallel and massively parallel computers has been proposed. Complete networks, hypercube interconnection networks, shuffle-exchange networks, fat-trees, two- and three-dimensional grids, and multigrids are just a few other favorites.

Once a particular data level parallel algorithm has been chosen to solve a given problem, the programmer's task is to implement the communication patterns required by the algorithm as efficiently as possible in terms of the underlying hardware. As in the case of serial programming languages, he can do best by programming the data level parallel computer in "machine language". However, as in the case of serial programming languages, this is undesirable because it requires considerable effort and is highly machine specific.

The solution is, of course, to provide a small set of abstract operations that can be implemented reasonably efficiently on a variety of different machines [13]. By including more specialized abstract operations, an implementer can provide more efficient implementations for specific cases, at the cost of increasing the complexity of the language.

We do not know yet what a good and reasonably complete set of abstract communication primitives for data level parallel programming is. The classes of communication patterns that we have encountered in practice are: arbitrary communication patterns with collisions, arbitrary communication patterns without collisions, and low-dimensional communication patterns without collisions (e.g. communication on a two-dimensional grid).

[12] We would like to see multidimensional subscripting added to the C++ language, i.e. to allow the programmer to overload operator[] with multiple arguments. This would require the programmer to parenthesize expressions containing the "," operator inside the subscripting operator, an incompatible but minor change in the syntax of the language.

[13] In many cases, the programmer can also give optional declarations that don't change the meaning of the program, but allow the implementer to use shortcuts to improve performance on specific machines.

Two basic classes of operations that occur frequently for all communication patterns are send operations (that simply move data around), and scan operations (that combine data with binary associative operators as it is moved; see in particular[Ble86] for a discussion of the use of the scanning primitive in data level parallel programming). Finally, for each operation and class of communication pattern, both uncompiled and compiled forms are useful; the uncompiled form is used for communication patterns that change often, whereas communication patterns that are used for many communication operations benefit from compilation.

But as important as a set of primitives is the expectations of the programmer of how "fast" or "slow" each primitive will be. Primitives for the high-dimensional uncompiled communication patterns and scanning primitives cannot be expected to execute quickly in general on data level parallel hardware that has only low-dimensional communication networks. Programming experience will show whether there are useful abstractions of classes of communication patterns intermediate in power between the high and low dimensional patterns.

These issues can be addressed and studied in our embedded data level parallel programming library because new communication primitives are easily added and easily tried with existing algorithms. More complex are questions of compile-time optimization of communication patterns. If some communication patterns are known at compile time, a compiler for a data level parallel programming language could in principle allocate elements of pvars to processors in such a way as to optimize the performance of the communication primitives. In cases such as the following, the compiler might, for example, decide to "renumber" the elements of pvar b globally to minimize the amount of inter-processor communication required during the execution of the program:

```
pint x=address(10),y=10-address(10);
pfloat a,b;

pfloat fun() {
    return a[x]+b[y]*b[y];
}
```

However, at the very least, optimizations like these require the compiler to evaluate constant expressions involving pvars.

8 Conclusions

C++ has proven to be a very useful tool for designing and implementing a programming system that lets us experiment with data level parallel programming. The ability to overload functions and operators allowed us to implement a reference counting memory allocation scheme, to specify data level parallel algorithms independently of what communication primitives are actually used, and to write expressions involving pvars and/or scalars concisely and in standard arithmetic notation. Virtual functions permit us to use communication primitives independent of the type of arguments they operate on.

The advantages to our approach are quite clear. Because our data level programming system is implemented as a library in a widely-used programming language, a programmer or user is already familiar with the syntax and semantics of the language. Both serial and parallel versions of the library can be implemented quickly and ported to a variety of

machines easily (since only library source code and not translator source code, as in the case of language extensions, needs to be transported).

There are also certain disadvantages to our approach. In principle, a dedicated compiler can perform better optimizations than the C++ compiler can, since a dedicated compiler "knows" more about the properties of specific language primitives. In practice, this is not a significant argument, since dedicated compilers tend to have poorer optimizers than compilers for general purpose languages, since significantly more development effort has been expended on the latter. We also hope that including new optional declarations in the C++ language will allow the C++ compiler to be able to perform optimizations that up to now only a dedicated compiler could perform. A dedicated compiler can also provide useful syntactic extensions to the base language. However, we feel that the C++ language syntax is powerful enough to let the programmer express his algorithms clearly and concisely.

A more serious practical argument against our approach is that by trying to be general and vendor-independent, a programmer may be forced to write less efficient code than if he were using a machine specific programming system. The answer is the same as in the case of the dualism of assembly language and high-level languages: use the latter for specifying and testing your algorithm, and if speed is of utmost importance, implement a few critical sections in assembly language. A Connection Machine implementation of our library still allows the programmer to use Paris instructions whenever speed is critical.

Altogether, we believe that our system is a useful and efficient alternative to existing programming systems for data level parallel programming. We expect that it will be of great utility in experimenting with different data level parallel programming primitives and in writing applications for artificial intelligence and vision research.

Acknowledgements

I would like to thank Jim Little, Robert S. Thau, and many others who have participated in discussions about data level parallel programming and/or made useful comments on the design of our system and the paper.

References

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[Ble86] Guy E. Blelloch. Parallel Prefix vs. Concurrent Memory Access. Technical report, Thinking Machines Corporation, October 1986.

[Hil85] W. Daniel Hillis. *The Connection Machine.* An ACM Distinguished Dissertation, 1985. MIT Press, 1985.

[HS86] W. Daniel Hillis and Guy L. Steele Jr. "Data Parallel Algorithms. *Communications of the ACM*, December 1986.

[HS87] Matthew Halfant and Gerald Jay Sussman. Abstraction in Numerical Methods. Technical Report 997, MIT Artificial Intelligence Laboratory, October 1987.

[ive62] K. E. Iverson. *A Programming Language.* John Wiley & Sons, Inc, 1962.

[KR77] Brian W. Kernighan and Dennis M. Ritchie. *The M4 Macro Processor*, July 1977.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 1978.

[LLM+88] Tom Leighton, Charles E. Leiserson, Bruce Maggs, Serge Plotkin, and Joel Wein. Theory of Parallel and VLSI Computation. Technical Report 1, MIT Laboratory for Computer Science, March 1988.

[Ste84] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1986.

[Thi86a] Thinking Machines Corporation. *Connection Machine Parallel Instruction Set (PARIS), Release 2 Revision 7*, July 1986.

[Thi86b] Thinking Machines Corporation. *Introduction to Data Level Parallelism*, April 1986.

[Thi86c] Thinking Machines Corporation. *The Essential *Lisp Manual*, July 1986.

[Thi87] Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary*, 1987.

[Tie88] Michael D. Tiemann. *User's Guide to GNU C++*, May 1988.

[TM85] Tommaso Toffoli and Norman Margolus. The CAM-7 Multiprocessor: A Cellular Automata Machine. Technical Report LCS-TM-289, MIT Laboratory for Computer Science, 1985.

[Tof84] Tommaso Toffoli. CAM: A high-performance cellular-automaton machine. *Physica*, 10D:195–204, 1984.

[Ung87] David Michael Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* An ACM Distinguished Dissertation. MIT Press, 1987.

[UP84] David M. Ungar and David A. Patterson. Berkeley Smalltalk: Who Knows Where the Time Goes? In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, 1984.

A Examples

The sieve is a nice example of how the C++ operator notation lets us express a data level parallel algorithm concisely. The idea is that the variable **sieve** represents those numbers whose primality has not beed decided yet, whereas **primes** holds numbers that are definitely prime. At each step the lowest number in **sieve** is obtained, which must be a prime, it is added to **primes**, and it and its multiples are removed from **sieve**. After $\sqrt{n}$ iterations, **sieve** contains all primes greater than **n**, and **primes** all primes less than **n**. The union of both variables is the answer.

```
//
// Compute the n lowest primes using the sieve method.
//
#include "pvar.hh"
void puts(char*);
double ceil(double);
double sqrt(double);

const n=20000;

int lowest(pbool& b) {
    pint adr=address(n);
    return adr[b][0];
}

main() {
    pint adr=address(n);
    pbool sieve(n),primes(n);
    int sqrtn=ceil(sqrt(n)),j;

    sieve=1;
    where(adr<2) sieve=0;
    primes=0;
    while((j=lowest(sieve))<sqrtn) {
        sieve&=((adr%j)!=0);
        primes[j]=1;
    }
    adr[primes|sieve].print();
}
```

B Code Templates

The following example illustrates the use of code templates in M4 to generate a complete set of operators corresponding to the operators that exist for standard scalar data types in C++:

```
// OP(returnType1,classType2,argumentType3,name4,op5)
define(OP,
p$1 p$2::$4(p$3& other) {
        int n=min(box->size,other.box->size);
        p$1 result(n);
        if(!selection) {
                for(int i=0;i<n;i++)
                        result.box->contents$1[i]=
                                (box->contents$2[i] $5 other.box->contents$3[i]);
        } else {
                bool *psel=selection->box->contentsbool;
                for(int i=0;i<n;i++)
```

```
                        if(psel[i])
                                result.box->contents$1[i]=
                                        (box->contents$2[i] $5 other.box->contents$3[i]);
        }
        shallowret(result);
})
```

We can now define groups of operators and use these to generate all operators for a particular data type as follows:

```
define(ARITH,
        // arith $1
        UNARY($1,$1,operator++,++)
        UNARY($1,$1,operator--,--)
        OP($1,$1,$1,operator+,+)
        OP($1,$1,$1,operator-,-)
        OP($1,$1,$1,operator*,*)
        OP($1,$1,$1,operator/,/)
        ...)

define(BOOLARITH,
        // boolarith $1
        OP($1,$1,$1,operator&,&&)
        OP($1,$1,$1,operator|,||)
        OP($1,$1,$1,operator^,!=)
        COP($1,$1,$1,operator&,&&)
        ...)
```

A Multiprocessor Operating System Simulator*

Gary M. Johnston
Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Avenue, Urbana, IL 61801-2987

Abstract

This paper describes a multiprocessor operating system simulator that was developed by the authors in the Fall semester of 1987. The simulator was built in response to the need to provide students with an environment in which to build and test operating system concepts as part of the coursework of a third-year undergraduate operating systems course.

Written in C++ [1], the simulator uses the co-routine style *task* package [2] that is distributed with the AT&T C++ Translator to provide a hierarchy of classes that represents a broad range of operating system software and hardware components. The class hierarchy closely follows that of the *Choices* [3] family of operating systems for loosely- and tightly-coupled multiprocessors. During an operating system course, these classes are refined and specialized by students in homework assignments to facilitate experimentation with different aspects of operating system design and policy decisions.

The current implementation runs on the IBM RT PC[1] under 4.3bsd UNIX.[2]

1 Introduction

The principles of low-level operating system design have implications that are difficult to appreciate without the practical experience gained by programming such systems. Multiprocessor systems present even more problems. However, it is difficult to provide university students with such a learning experience. Hardware resources are too expensive to allow each student single user access to a multiprocessor workstation. Low level parallel processing systems software, as an instructional resource, is usually poorly organized and difficult to understand. In addition, there is little support for the debugging and testing of low-level systems programs on multiprocessors. This paper describes a multiprocessor operating system simulator we have constructed in C++ to overcome these problems. The current implementation is used in the department's instructional laboratory and runs on 30 IBM RT PCs which were donated to the university by the IBM Corporation.

The simulator is modeled on the Choices multiprocessor operating system family [3] [4] [5]. It includes classes to model both the processes, schedulers, and exception handling

*This work was supported in part by NSF grant CISE-1-5-30035, by NASA grant NSG1471, and by AT&T ISEP.

[1]RT PC is a trademark of IBM.

[2]UNIX is a trademark of AT&T.

mechanisms of Choices and the processors, I/O devices, traps, interrupts, timers, and other hardware components of a typical multiprocessor like the Encore Multimax.[3]

The simulator was designed for a third- or fourth-year undergraduate course on operating systems that is taught in the department. The goal of the course is to introduce students to the principles of operating systems and to reinforce those principles with practical experiments and projects involving the design of operating system mechanisms and policies.

Using the simulator, experimentation is conducted within the framework of the class hierarchy and object-oriented programming mechanisms afforded by C++. Many of the practical design exercises involve specializing an abstract class into a concrete class that implements a particular policy or mechanism. Policy exercises include process scheduling, real memory management, page replacement, and disk scheduling. Mechanism exercises include synchronization primitives, I/O queues, paging mechanisms, exception handling schemes, and message passing primitives.

The operating system course benefitted from the use of C++ in several ways. The language allows an efficient simulation of the operating system while providing a level of type checking that aids debugging of student programs. Debugging and tracing aids are built into the base classes of the simulator and help the students implement their designs. The class hierarchy organizes the components of the simulation into similar algorithms and data structures. This organization is a useful aid to the student that is learning the system. The class hierarchy enables fairly large simulations of an operating system to be built incrementally by the students.

The remainder of this paper consists of four major sections. Section 2 describes the model of the Choices operating system and class hierarchy supported by the simulator. Section 3 discusses the design and implementation of the simulator. Section 4 describes how the simulator was used, including descriptions of some of the projects. Finally, we summarize our experience with the simulator in section 5.

2 Choices Overview

Choices is a family of operating systems built using a class hierarchical object-oriented approach to systems design and programming. A Choices operating system has been implemented on an Encore Multimax and is being ported to an Intel iPSC/2[4] hypercube [6]. It demonstrates that object-oriented design techniques are both appropriate and beneficial for writing complete operating systems for multiprocessors and networks of multiprocessors.

In Choices, a class hierarchy represents the major components of a family of operating system designs. Classes represent the interfaces and implementations of processes, virtual memory, context switching, exception handling, scheduling, and synchronization. They are also used to provide a hardware/software interface by encapsulating machine dependent algorithms and data structures for the hardware entities such as the CPUs, MMUs, interval timers, disks, and networks.

The goal of the system is to allow an operating system designer to select, refine, and combine classes from the class hierarchy to build a custom operating system for a particular hardware environment or a particular application. The resulting operating system is also more easily modified or extended than one based on more "traditional" approaches [7].

[3] Multimax is a trademark of Encore Computer Corporation

[4] iPSC is a trademark of Intel Corporation.

Choices Simulator Classes				
Class	Methods			
Object	–	–	–	–
↑Process	–	–	–	–
↑↑IdleProcess	–	–	–	–
↑ProcessContainer	**add**	**remove**	–	–
↑↑CPU	*add*	*remove*	**interrupt**	**trap**
↑↑FIFOQueue	*add*	*remove*	–	–
↑↑ProcessQueue	*add*	*remove*	–	–
↑Exception	**raise**	**await**	**handle**	–
↑↑InterruptException	↑	↑	*handle*	–
↑↑↑ResetException	↑	↑	*handle*	–
↑↑↑TimerException	↑	↑	*handle*	–
↑↑SoftwareException	*raise*	↑	*handle*	–
↑↑↑IdleException	↑	↑	*handle*	–
↑↑↑TerminateException	↑	↑	*handle*	–
↑↑↑SemaphoreException	↑	↑	*handle*	–
↑Semaphore	**P**	**V**	–	–

Legend	
Symbol	Meaning
method	Definition of method.
method	Redefinition of method.
↑	Subclass or inherited method.
–	Undefined method.

Table 1: Choices Simulator Classes.

The ease of module substitution greatly facilitates prototyping, a great benefit to practical operating systems research and experimentation.

This section presents a brief overview of the Choices project and of the Choices class hierarchy as implemented by the simulator. For more detail, see [3] [4] [5].

2.1 The Choices Class Hierarchy

The major classes of Choices as modeled by the simulator are shown in table 1. Class *Object* is the root of the hierarchy. Subclasses are used to provide abstract interfaces and concrete implementations for operating system mechanisms. They are used to encapsulate data, policies, and alternative implementations or versions. Subclasses of Object define the basic entities within an operating system. Further, subclasses of these classes add and/or redefine methods in order to augment, specialize, or provide concrete implementations of these classes.

2.1.1 Processes and ProcessContainers

Class *Process* provides the basic unit of execution within Choices. Process management in the operating system is achieved by moving Processes between *ProcessContainers*. Subclasses of ProcessContainers represent processors and schedulers.

An *IdleProcess* is associated with each simulated processor. It is executed only when there are no other runnable Processes available. Each IdleProcess periodically checks the scheduler and signals the processor when it detects that there is a Process which could be executed.

Other Processes represent "user-level" processes. The behavior is redefined by the simulation designer as necessary. Usually user-level Processes are designed to simulate a "job load" for the simulation.

The ProcessContainer class defines methods to *add()* and *remove()* Processes. These methods are specialized by the subclasses *CPU*, *FIFOQueue*, and *ProcessQueue*.

The CPU subclass of ProcessContainer represents processors. Adding a Process to a CPU specifies that it should be executed by a particular processor of the multiprocessor system; that is, the Process is dispatched on the CPU. Removing a Process from a CPU idles the processor, which represents preemption of the Process. Multiple instances of class CPU represent multiple processors in a multiprocessor.

Facilities for scheduling and blocking Processes are provided by classes FIFOQueue and ProcessQueue. A FIFOQueue acts as a simple "first-in-first-out" queue of Processes, while a ProcessQueue is associated with a timeslice quantum. When a Process is removed from a ProcessQueue, the timeslice quantum field of the Process is set to the quantum associated with the ProcessQueue. This field is used by the CPU to determine the maximum amount of time the Process should be allowed to execute before being preempted. The quantum associated with a ProcessQueue may be any value desired. The default quantum is a value which means "run-to-completion." These classes may be refined by other subclasses in order to implement a wide range of policies. FIFOQueues can act as queues of blocked Processes. Other subclasses of ProcessContainer can be defined and substituted to provide whatever sorts of scheduling disciplines the system designer desires.

2.1.2 Exceptions

In Choices, most movement of Processes between ProcessContainers is by *Exception* handlers. In addition, the only way in which an executing Process can relinquish its CPU is by the raising of an Exception. Relinquishing the CPU may be a voluntary, synchronous action performed by the Process (i.e., a "trap") or an involuntary, asynchronous action caused by an external event (i.e., an "interrupt").

Class Exception itself is an *abstract* class.[5] An Exception provides the methods *handle()*, *raise()*, and *await()*. The raising of an Exception causes its handler to be invoked (with the possible side-effect of unblocking one or more Processes awaiting the Exception).

There are two abstract subclasses of Exception: *InterruptException* and *SoftwareException*, each of which is further subclassed. An InterruptException is associated with an interrupt vector which, when delivered to a CPU, causes the associated InterruptException to be raised. Thus, InterruptExceptions occur asynchronously with the execution of Processes. A SoftwareException is *not* associated with an interrupt vector. Instead, it is raised directly by an executing Process and acts like a "trap."

InterruptException subclasses include *ResetException* and *TimerException*. Each CPU is associated with an instance of each of these. A ResetException provides the actions to be taken when the CPU is "reset". The TimerException handles the expiration of the per-CPU interval timer.

[5] By *abstract*, we mean that no instances of the class ever exist. Rather, it is used as a base class from which subclasses are derived in order to provide specialized behavior.

Other SoftwareException subclasses include *IdleException*, *SemaphoreException*, and *TerminateException*. An IdleException is a software event that signifies that Processes are available to the CPU for execution. An IdleException is raised by a CPU's IdleProcess when it detects that the CPU's scheduler is non-empty. A TerminateException is raised to remove the Process from the CPU and delete it. A SemaphoreException is raised when a Process attempts to acquire a semaphore which is unavailable. The SemaphoreException removes the Process from the CPU and adds it to the queue of Processes waiting for the semaphore.

2.1.3 Semaphores

A *Semaphore* is the basic synchronization primitive within the simulator. It defines the familiar *P()* and *V()* operations [8] for acquiring and releasing the Semaphore.

3 Implementation

The simulator provides a class hierarchy from which simulated multiprocessor operating systems can be designed and studied, following the Choices model as closely as possible. This section discusses the implementation of the simulator.

3.1 Microscheduling

Like Choices itself, the simulator is written in C++. In order to provide the required simulated concurrency, the simulator uses the "coroutine-style task package" which accompanies the AT&T C++ Translator [2].

The task package provides user-level coroutine-style tasks, but does not provide for non-voluntary relinquishing of the virtual processor. That is, an executing task does not block unless it explicitly calls a task package procedure (for example, *delay()* or *sleep()*). While this is very useful for system simulation, it is inadequate to emulate a multiprocessor programming environment realistically. A simulated user-level task executing an "infinite loop" will prevent all the other simulated tasks from proceeding. This simple implementation of tasks is inadequate to emulate interrupts or preemptive scheduling policies such as round-robin time-slicing, multi-level feedback queues, or "shortest job first."

In addition, we wanted to simulate the nondeterminancy that must be dealt with by programs using or implementing synchronization primitives and executing within a multiprocessor environment. Therefore, the basic task package was augmented with a "microscheduling" sub-system that time-slices between executable tasks preemptively. Note that this involved only *additions* to the task package. The task package itself was *not* modified.

The microscheduling mechanism implements a time-sliced round-robin mechanism underneath the basic task package. This mechanism gives each executable (i.e., non-blocked) task a "microquantum" equal to one virtual clock tick. At the end of the microquantum, the task is delayed for one clock tick, and the next executable task is dispatched. In this manner, executable tasks are preemptively time-multiplexed on the underlying UNIX process.

The 4.3bsd UNIX interval timer and signal mechanisms were used to implement the actual preemption of tasks. At simulator initialization time, an interval timer is armed to deliver a signal to the underlying UNIX process each time the timer expires. When the signal is received, the signal handler executes in the context of the current task. The signal

handler executes a call to the task package to delay the current task by one virtual clock tick, thus relinquishing the underlying UNIX process to execute another runnable task. If there are no more immediately runnable tasks, the virtual clock is incremented (by the task system), allowing tasks which had delayed themselves during the previous clock tick to become "ready" again. When a task that had previously delayed itself via the signal handler becomes ready again, its invocation of the signal handler returns, thus restoring that task's context to that which was in effect when the signal was received. The task then continues execution at the point where it was preempted. Thus, the microscheduling effectively implements a round-robin scheduling policy underneath the existing task package.

The basic task package requires no explicit shared resource access control internally because there is no preemption. Provided that critical sections do not delay, they do not need synchronization because, without preemption, races cannot occur. Once microscheduling has been added, however, this is no longer the case. Within the Choices classes, mutual exclusion primitives are used in order to ensure that critical sections are protected. In order to support these primitives in the simulator, instances of two low-level task classes are distinguished by the microscheduling mechanism and are *not* preempted.[6] Therefore, these classes' methods do not need to use explicit mutual exclusion primitives.

3.2 Class Hierarchies and Layering

The simulator is organized into two major class hierarchies: the augmented task package class hierarchy (including the microscheduling mechanism) and the Choices class hierarchy itself.

The basic task package provides the abstraction of a *task*, which is the primitive unit of execution within a task package application. This hierarchy has been augmented by creating subclasses of the task class in order to provide more specialized behavior as needed by the rest of the simulator. These classes are *CPUManager*, *CPUTimer*, and *ProcessTask*. A CPUManager and a CPUTimer are associated with each simulated CPU. The CPUManager simulates the activity of the CPU. This includes interrupt vector processing, trap processing, and exception handling actions. The CPUTimer simulates a per-CPU interval timer to provide support for preemptive time-slicing of simulated Processes. A ProcessTask is associated with each simulated Process. The CPUManager associated with a CPU allows the ProcessTask to execute (on behalf of the simulated Process) when the Process is dispatched on that CPU.

The Choices simulator class hierarchy provides the classes that form the basis for operating system simulations: Process, ProcessContainer, CPU, Exception (and its subclasses), etc. Table 1 shows this hierarchy. Figure 1 shows the arrangement in terms of layers.

3.3 Class CPU

A CPU contains a number of objects in addition to its CPUManager and CPUTimer. Each CPU has a current Process and an IdleProcess. The current Process is the Process currently being executed by that CPU. Since a CPU is a ProcessContainer, the current Process of a CPU references a Process which has been added to the CPU. The IdleProcess is executed only when the CPU is otherwise idle (e.g., when there are fewer Processes in the "system" than there are CPUs[7]).

[6] These classes are *CPUManager* and *CPUTimer*, discussed below.

[7] not including IdleProcesses, of course.

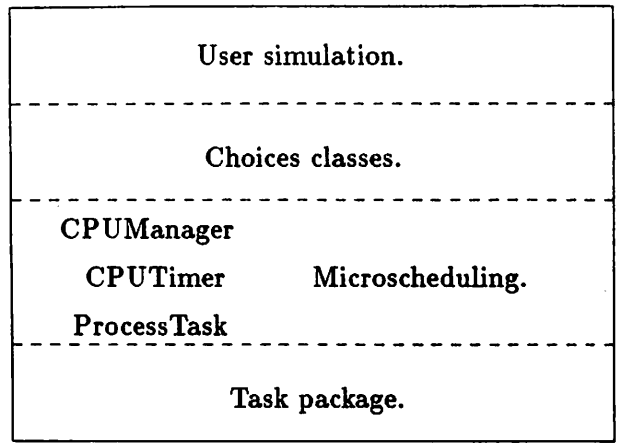


Figure 1: Conceptual Layering in the Choices Simulator.

Next, a CPU contains a queue of pending interrupt vectors and a table that maps interrupt vectors to InterruptExceptions. Incoming interrupt vectors and SoftwareExceptions are detected by the CPUManager which then executes the InterruptException handlers.

Each CPU references a ProcessContainer that operates as the "ready queue" or scheduler. When an executable Process is removed from the CPU, it is added to this scheduler. Also, a Process is removed from this scheduler when the CPU requires one. For example, when an executing Process' timeslice expires, it is removed from the CPU and added to this scheduler. Then, another Process is removed from the scheduler and added to the CPU.

In this way, several CPUs may be associated with a particular scheduler. There is no reason why there can't be more than one scheduler in the system, each associated with its own set of CPUs. The simulation designer can change this association dynamically at any time.

There are two groups of operations on a CPU: "private" routines intended for use by "friends" (essentially CPUManagers and Exception handlers) and "public" routines intended for use by the simulation writer.

The private routines include *add()*, and *remove()*, which are redefinitions of the superclass ProcessContainer methods for adding/removing Processes to/from a ProcessContainer. Adding a Process to a CPU is effectively a "dispatch" of the Process, while removing a Process from a CPU corresponds to a "preemption" of the Process.

Two other important private routines are *removeVector()* and *getException()*. These are used by the CPUManager to remove an interrupt vector from the incoming vector queue, and to map a vector to an InterruptException, respectively.

The public operations include the constructor and destructor, routines to get and set the CPU's scheduler ProcessContainer, the *interrupt()* routine which is used to send an interrupt vector to a CPU, the *trap()* method which is used when a SoftwareException is raised, and the *setException()* routine which is used to associate an interrupt vector with an InterruptException.

When a CPU is created it is empty, i.e., it contains no Process. The Exception table (which maps interrupt vectors to InterruptExceptions) contains two default mappings: a ResetException is associated with the ResetVector, and a TimerException is associated with the TimerVector.

In the implementation, the CPU itself is passive; it is the CPUManager and the CPUTimer which are the active entities, controlling the activities of the CPU. These are discussed

next.

3.4 CPUTimer

A CPU's CPUTimer implements timed preemption of Processes. A CPUTimer is a task that sends the TimerVector to the CPU when the time interval expires. If the CPUTimer is stopped before it expires, then the residual time can be retrieved.

In general, when a Process that specifies a timeslice quantum is dispatched, the CPUManager sets the CPUTimer to expire at the appropriate time. If the CPUTimer expires, the TimerVector interrupt triggers the execution of the associated InterruptException's handler (usually a TimerException). If the Process is preempted for some reason other than CPUTimer expiration, the CPUTimer is stopped and the residual is read and stored in a field of the Process for possible use by the scheduler.

3.5 CPUManager Duties

The CPUManager handles asynchronous events in the system like interrupts, as well as synchronous events such as traps, and invokes the Exception handlers associated with them. The CPUManager is initially "asleep," and the arrival of an interrupt or trap "wakes up" the CPUManager. When a CPU's *interrupt()* method is called, the vector is enqueued on the CPU and its CPUManager is awakened. When a CPU's *trap()* method is called, the SoftwareException is saved on the CPU, the invoking Process is stopped, and the CPUManager is awakened. The general control loop of the CPUManager is shown in figure 2.

3.6 Processes and ProcessTasks

Each Process is implemented by a *ProcessTask* which executes when the Process is dispatched on a CPU. Each Process contains a timeslice quantum and a residual, which is used for preemptive timeslicing. The residual field is set by the CPU when the Process is preempted. This information is intended for use by schedulers. In addition, each Process keeps run-time statistics.

The ProcessTask associated with a Process is the entity which is actually executed. It is ProcessTasks that are multiplexed on the underlying UNIX process by the microscheduling mechanism. The task methods are used by a CPUManager to start and stop the execution of a Process' ProcessTask. In order to provide low-level critical section protection, methods are provided to disable and re-enable the preemption of a ProcessTask by the microscheduling mechanism.

IdleProcess is the subclass of Process that is executed by a CPU when there are no other Processes for it to run. There is one IdleProcess associated with each CPU. The IdleProcess continually checks the scheduler ProcessContainer of its CPU. When it detects that this scheduler is not empty, it raises an IdleException which causes a Process to be removed from the scheduler and added to the CPU, suspending the IdleProcess until such time as the CPU becomes idle again.

3.7 Exceptions

The Exception subclasses are the major means by which Processes are moved between ProcessContainers. Each Exception subclass provides specialized handling. There are two sub-

```
// A CPUManager's work is never done...
for (;;) {
    // Wait for an interrupt.
    sleep();

    // Stop and delete the CPUTimer, if there is one, saving the residual.
    int residual = 0;
    if ( cpu->timer != NULL ) {
        residual = cpu->timer->stop();
        delete cpu->timer;
        cpu->timer = NULL;
    }

    // Handle and reset the pending trap (SoftwareException), if there is one.
    // Otherwise, stop the current Process, if there is one.
    Process * currentProcess = cpu->currentProcess;
    if ( cpu->trap != NULL ) {
        SoftwareException * trap = cpu->trap;
        cpu->trap = NULL;
        trap->handle( cpu );
    } else if ( currentProcess != NULL ) {
            currentProcess->stop();
    }

    // Handle any pending interrupts (InterruptExceptions).
    while ( ( int vector = cpu->removeVector() ) != NoVector ) {
        // Get the corresponding Exception.
        // Call the Exception handler.
        InterruptException * interrupt = cpu->getException( vector );
        interrupt->handle( vector, cpu );
    }

    // Start the current Process, if there is one.
    // Note: The current Process we start here might very well not be
    //       the same one we stopped.
    if ( cpu->currentProcess != NULL ) {
        // Determine how much time the Process will get:
        //     If the current Process is the same as before,
        //     it gets the rest of its timeslice (i.e., the residual).
        //     Otherwise, it gets whatever its scheduler specified.
        int time = (cpu->currentProcess == currentProcess) ?
                    residual :
                    cpu->currentProcess->getQuantum();

        // Start the CPUTimer, unless the Process is marked "run to completion."
        if ( time != RunToCompletion )
            cpu->timer = new CPUTimer( cpu, time );
    }
}
```

Figure 2: Simplified CPUManager control loop.

classes of Exception, InterruptException, and SoftwareException. Instances of subclasses of InterruptException represent hardware interrupts. When an interrupt is delivered to a CPU, it is mapped by the CPUManager to an InterruptException whose handler is then called. Subclasses of InterruptException include:

ResetException: Associated with the ResetVector. It adds the CPU's IdleProcess to the CPU.

TimerException: Associated with the TimerVector which is sent when the CPU's CPU-Timer expires. It removes the current Process from the CPU and adds it to the scheduler ProcessContainer associated with the CPU. It then removes a Process from the scheduler and adds it to the CPU.

A SoftwareException is raised as a direct result of the execution of a Process. SoftwareExceptions are not associated with interrupt vectors; the raise method is invoked directly. SoftwareException subclasses include:

IdleException: Raised when a CPU's IdleProcess detects that the CPU's scheduler has become non-empty. Its handler removes the IdleProcess from the CPU, and then removes a Process from the scheduler and adds it to the CPU.

TerminateException: Raised when the current Process on the CPU is to be terminated. It removes and deletes the current Process from the CPU, and then removes a Process from the scheduler and adds it to the CPU.

SemaphoreException: Raised by a Semaphore when a *P()* operation detects that the requesting Process must block (i.e., the resource is not available). It removes the current Process from the CPU and adds it to the ProcessContainer associated with the Semaphore. It then removes a Process from the CPU's scheduler and adds it to the CPU.

3.8 Semaphores

Each Semaphore contains a count and a FIFOQueue ProcessContainer which holds Processes that have been blocked attempting to acquire the Semaphore. It also references a SemaphoreException that is raised when a Process must block.

The *P()* operation decrements the count. If the count then indicates that the Process must block, a SemaphoreException is raised. The SemaphoreException removes the Process from the CPU and adds it to the queue of blocked Processses.

The *V()* operation increments the count. If there are blocked Processes, one is removed from the queue and added to the scheduler.

4 Projects

The resulting simulator has proven to be very realistic. Several of the race conditions that occurred as bugs in the development of the real Choices operating system were also encountered by students as they developed their own operating system components within the simulator. During the course, the students developed semaphores, messages, supervisor requests, scheduling policies, real storage management, virtual storage management, disk

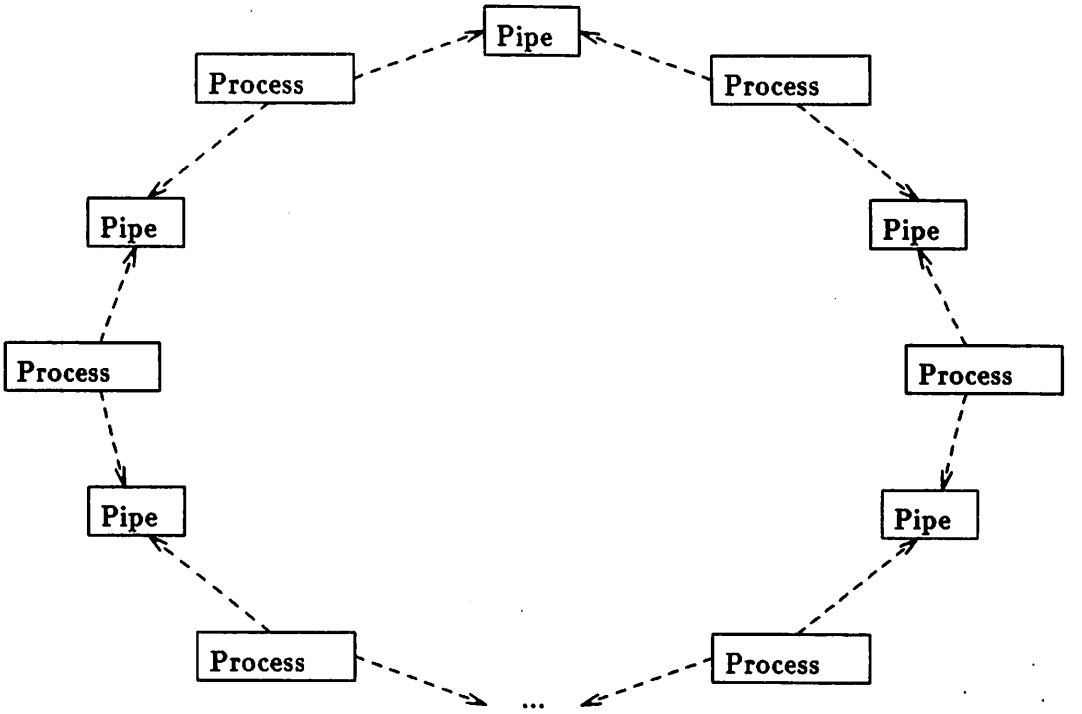


Figure 3: A Ring of Processes Connected by Pipes.

storage management and scheduling for the multiprocessor environment. This section discusses some of these projects and how they were implemented within the environment of the simulator.

4.1 Multiple Concurrent Producers and Consumers

The object of this exercise was to give students experience in designing systems involving producer/consumer relationships among Processes, including deadlock detection and recovery.

Initially, class *Pipe* had to be implemented to support a two-ended stream of *Messages*. Methods were required to perform blocking, non-blocking, and synchronous send operations (*send_block()*, *send()*, and *send_sync()*, respectively), as well as blocking and non-blocking receive operations (*receive_block()* and *receive()*, respectively). Each Message essentially consists of a string of data bytes and an identifier specifying the ultimate destination Process.

In this exercise, Processes are connected by Pipes in a ring, as shown in figure 3. Each Process executes a loop in which it repeatedly choses one of the send or receive operations at random, and then performs this operation on one of its two Pipes. For send operations, destinations are chosen at random. For receive operations, if a Message is received on a Pipe whose destination does not specify the receiving Process, it is forwarded on the other Pipe. Since the Processes are arranged in ring, all Messages eventually reach their destinations (unless they are lost or cancelled).

In this situation, deadlocks can and do occur. Students implemented a centralized deadlock detection and recovery mechanism that consisted of a central Pipe *Control* information object and an additional deadlock control Process that periodically examined the Control information, discovering and breaking deadlock situations. The Pipe class was modified to

support this. Each send and receive operation on a Pipe would report its updated state to the Control object, where it could then be used by the deadlock control Process.

4.2 Real Memory Management

This project involved the implementation of "Choices-like" real memory management. Two major classes were implemented: *RealMemoryObject* and *RealMemoryManager*.

A RealMemoryObject represents a "segment" or contiguous range of memory organized in fixed-size pages. The operations supported are *read()* and *write()*. Each operation specifies an *offset* into the RealMemoryObject at which the transfer is to begin, a *length* (in bytes), and a destination/source *buffer* address. Initial reads from unwritten RealMemoryObject locations return zeros. The RealMemoryObject maintains a "dirty bit" for each page which has been written. The constructor specifies the range of addresses which the RealMemoryObject will represent.

The other major class required for this project was a RealMemoryManager. A RealMemoryManager represents the physical memory of the simulated machine, so only one instance of this class is created. The RealMemoryManager allocates and deallocates RealMemoryObjects as requested by user Processes. Operations are *allocate()* and *deallocate()*.

The *allocate()* operation specifies a number of bytes, and returns a RealMemoryObject. The RealMemoryManager must find an unallocated range of memory that is at least as large as the request. It then creates a RealMemoryObject to manage the range and returns it.

The *deallocate()* operation specifies a RealMemoryObject to be deleted. The RealMemoryManager deletes the RealMemoryObject, thus freeing the range of memory for possible allocation in future *allocate()* requests.

RealMemoryObject and RealMemoryManager provide simulated system services, and are not supposed to be directly accessible by the user Processes (although the simulator cannot enforce this). Therefore, the students implemented a subclass of SoftwareException called *SVCException*. This class provides a user program interface to the system. Mechanisms for passing arguments into the "kernel" and for passing results back to the invoking Process were also implemented.

Simulated user Processes were created to randomly allocate and deallocate RealMemoryObjects, and to read and write them randomly. Statistics about memory usage, fragmentation, and allocation routine times, etc. were collected. The allocation algorithms commonly known as "first fit," "best fit," and "worst fit" were implemented and analyzed.

4.3 Virtual Memory Management

This project extended the ideas from the previous project in order to provide students with experience in the various aspects of virtual memory management.

The idea of a RealMemoryObject was expanded to represent a Process' virtual address space. This is encompassed by class *MemoryObjectCache*. A MemoryObjectCache maintains the state of each page in the virtual address space it represents. In addition to the "dirty bit" (which was maintained by the RealMemoryObject in the previous project), the MemoryObjectCache must maintain a "referenced bit" and a bit indicating whether or not the page is resident. If the page is non-resident, the location of the page in secondary storage must be stored. A MemoryObjectCache supports the same read and write operations as

described for a RealMemoryObject, except that pages may be moved to and from secondary storage.

When a MemoryObjectCache must read or write a page that is marked non-resident, that page must first be retrieved from secondary storage. To facilitate this, an instance of class *PageManager* manages the physical memory of the machine and is responsible for paging to and from secondary storage. The PageManager implements the *pageFault()* method, which is invoked by a MemoryObjectCache when a non-resident page needs to be brought in from secondary storage. The PageManager fetches the specified page from secondary storage and marks it as resident.

Secondary storage is implemented with an instance of class *DiskManager*. The DiskManager responds to the messages *readPage()* and *writePage()*.

Various page replacement algorithms were implemented and studied. These included "least recently used," "not recently used," "first in, first out," and "random." In addition, various disk scheduling strategies were used including "first come, first served," "linear (or unidirectional) scan," and "circular (bidirectional) scan." Finally, the page access patterns of the Processes were varied in order to simulate different degrees of temporal and spatial locality.

5 Conclusion

In this paper, we have described the use of C++ as a high-level language for describing the system data structures and algorithms introduced in a university course in operating systems. The students used a simulator programmed in C++ that emulated a system based on Choices, an experimental multiprocessor operating system that we are building at the University of Illinois. Class projects and exercises were chosen to give students practice at systems design and programming. These projects and exercises were written in C++ and refined or replaced classes in the simulator.

Most of the students in the course had programmed in C in a previous course on systems programming and machine organization. The transition to C++ was orderly. The students found the additional type checking in C++ an aid; however, many of the diagnostic messages from the compiler required the students to seek help from their teaching assistants. The debugging and tracing aids built into the simulator were found to be very useful as the standard UNIX debugger cannot give accurate diagnostic messages in terms of the names used in C++ programs. This is because the current C++ compiler generates C code which is then compiled by the C compiler. A native C++ compiler would solve many of these problems.

C++ was proved to be an efficient programming language for the simulator. Quite large simulations (both in terms of size and length) could be done on a workstation during the period of time permitted each student in the laboratory.

The use of a class hierarchical object-oriented description of an operating system was instrumental in helping the students understand Choices. The class hierarchies organized the common algorithms and data structures of an operating system and allowed students to infer the properties of the simulator classes from the more abstract classes presented during lecture. Unlike previous operating system courses that we have taught, we were able to present multiprocessor operating system material couched in the general principles of operating system design. The more "traditional" single processor operating system algorithms and data structures could be presented as degenerate cases of the multiprocessor ones.

Currently, a simulator is the only practical approach to providing a large class of students (approximately sixty) with a hands-on environment for multiprocessor operating system design. Many of the problems that are encountered in multiprocessor operating system design — deadlocks, races, unnecessary mutual exclusion and interrupt disabling, etc. — were pointed out in lecture and successfully diagnosed by students during their exercises on the simulator. In this and many other respects, the simulator provided a remarkably accurate emulation of real multiprocessor system software development. The accuracy of that emulation requires better diagnostic and tracing tools than we implemented in the simulator. We believe some form of graphical visualization of the system is needed in order to provide students with a better understanding of the utilization of resources, bottlenecks, and communication flows. However, we do not see this as a drawback to the approach. Rather, it points out a lack of necessary human interfaces and tools for designing complex software. Such software tools would not only be useful in education, but they would have application in the customization of Choices for particular applications and hardware. We plan to incorporate such tools in the future revisions of the simulator.

References

[1] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] Bjarne Stroustrup & Jonathan E. Shopiro, "A Set of C++ Classes for Co-Routine Style Programming," *Proceedings of the USENIX C++ Workshop* (1987).

[3] Roy Campbell, Vincent Russo & Gary Johnston, "The Design of a Multiprocessor Operating System," *Proceedings of the USENIX C++ Workshop* (1987).

[4] Vincent Russo, Gary Johnston & Roy Campbell, "Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques," *OOPSLA '88 Conference Proceedings* (forthcoming).

[5] Roy H. Campbell, Gary M. Johnston & Vincent F. Russo, "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)," *Operating Systems Review* 21 (July 1987), 9–17.

[6] Roy H. Campbell & Daniel A. Reed, "Tapestry: Unifying Shared and Distributed Memory Parallel Systems," Department of Computer Science, University of Illinois at Urbana-Champaign, Technical Report No. UIUCDCS-R-88-1449, Urbana, Illinois, 1988.

[7] Ralph Johnson & Brian Foote, "Designing Reusable Classes," *The Journal of Object-Oriented Programming*. 1:2.

[8] Edsger W. Dijkstra, "The Structure of the THE-Multiprogramming System," *Communications of the ACM* 11 (May 1968), 341–346.

Modelling of Control Systems with C++ and PHIGS

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
Box 118, S-221 00 Lund, SWEDEN

E-mail: dag@control.lth.se

Abstract

This paper describes an interactive tool for modelling of control systems. The focus is on practical experiences with C++ as a development tool, and the need for multiple inheritance, parameterized types, and exception handling, in this application. Experiences with a new graphics standard, PHIGS, using an object-oriented programming style, are briefly covered.

1. Introduction

Modelling has traditionally been one of the main topics in control engineering. Control systems are complex and require careful design and analysis, in particular, as errors in control system design can become expensive. There exists today a great need for computer aided design of control systems.

Our research is centered around tools for model development and simulation. The objective is to design the basic concepts needed for structuring models, and to design the internal computer representation of control system models. An experimental tool for modelling and simulation has been developed in KEE, an expert system shell.

The experimental tool will form the basis of an engineering tool for the designer of control systems. In such a product, flexible, efficient and affordable system software must be used. We have therefore evaluated C++ as the future implementation language, and PHIGS as the main graphics system. A simplified experimental tool has been implemented in C++. Whereas the KEE version supports all essential parts of an engineering tool, the C++ version only provides graphical interaction; the internal structure is quite similar, in order to meet future needs.

2. Modelling of control systems

The model of a control system can be regarded as a hierarchy of components. One of the fundamental ideas is to build libraries of component models, ranging from basic items (for example, a pump) to more complex objects (for example, a distillation column). The designer has the option of working bottom-up, putting predefined components together to form a new component, or top-down, decomposing a complex object into manageable pieces,

or most likely, a combination of bottom-up and top-down design [Nilsson, 1987]. The key word is reuseability — of earlier designs and of standard components.

A single component can be described in many ways: graphically, textually, using block diagrams (describing its structure), or mathematically (for example, in state-space or transfer-function form). It is also necessary to use models with different degrees of detail and complexity, for example, an efficient simulation model for normal operation, and an extended model for analyzing error conditions. All these models are needed in different stages of the design, and should be available in a model development tool. It should be noted that the common "machine" view may be replaced by a "materials" view. For example, a chemical compound may carry all knowledge in the model, while the stations in the refinery only signal changes of state.

With our set of basic concepts, a model has three properties: it has terminals which provide an interface to the outside world, parameters for adapting its behaviour, and at least one realization that defines its behaviour. Only data in the terminals are available to other components; there are no global data, except a time reference for simulation.

We currently support two types of realizations: primitive realizations using ordinary differential equations, and structured realizations using block diagrams. A structured realization consists of submodels and connections (between submodels, and between submodels and the terminals of the enclosing model). Interaction between components is defined only by connections.

Simulation is often used to analyze control systems, and the designer should be able to simulate his/her model using this tool. Simulation introduces a number of interesting mathematical problems, which will not be covered further in this paper [Mattsson, 1988b]. The connection concept also raises interesting questions: for example, what is a legal connection, and how do you define compatibility between terminals [Mattsson, 1988a].

According to current trends, it is also necessary to throw in an expert system and a couple of knowledge bases.

3. Direct model representation

Modelling of control systems maps nicely to the ideas in object-oriented programming. It is natural to represent a model with a class in the programming language used for implementing the design tool. It is then possible to develop new models using inheritance and specialization of classes.

Inheritance is not suitable for describing all kinds of relationships between models. Multiple representations of a single model (textual or mathematical), and specialization (a car is a special kind of vehicle), can be described with inheritance. Decomposition of a model into its components is different. For example, that a car has tyres does not mean that the car can be inflated, so inheritance is not the right mechanism; components are represented by class members (Listing 1).

The direct way of representing models with classes is used in the experimental tool developed in KEE. Instantiation is used, for example, to create objects that contain simulation data. A necessary key feature of KEE (and object-oriented systems like Loops) is the possibility to dynamically define new classes while the program is running.

```
class vehicle {
  char*  owner;
};

class car : public vehicle {
  tyre   fl, fr, rl, rr;
  engine e;
};
```

Listing 1. Direct representation of a car model, derived from vehicle.

4. Model representation in C++

If interactive model development is presumed, direct representation is not possible in C++, simply because classes cannot be defined at runtime. Consequently, components cannot be represented directly with class members, and inheritance cannot be used to derive new models. To be able to interactively create models, we must implement a dynamic framework for representing models, realizations, etc. This framework is similar to the class systems commonly based on Lisp, but the implementation task is simplified by the structure of control systems.

It should be noted that the engineer developing control systems will see an interactive modelling tool; C++ is used only to implement the dynamic framework, not as a control system description language. One can also say that the object-oriented aspects of model representation have been separated from the object-oriented aspects of C++. Still, object-oriented programming effectively supports the design and implementation of the framework.

Internal data structures

Now, let's plunge straight into the internal data structures of the C++ program. The code listed below is slightly simplified; constructors and destructors are not listed, and most general purpose routines have been omitted. An example will be given below.

All objects are components; they have a name, and they can be inserted into lists (Listing 2).

```
class component {
  char* name;
  link  next;

public:
  virtual void menuaction();
  virtual void redraw();
};
```

Listing 2. Definition of the basic component class.

Method `redraw` is a schoolbook virtual function in C++: every component has a graphical representation, so all components must implement `redraw` in some way. Graphics will be described further in Section 5.

When the user points at a component and presses a mouse button, some components (e. g., models and realizations) will respond by displaying a menu. Other components (e. g., terminals and connections) are not associated with a menu. In C++, which in its present shape only supports single inheritance, method `menuaction` must be declared as a

virtual function in the base class, component. When multiple inheritance becomes available in C++, menuaction would more naturally be the property of a class associated purely with the user interface; models and realizations would be derived from this class, but not terminals and connections [Stroustrup, 1987a].

Generally speaking, multiple inheritance enables us to separate the user interface and the modelling structure more effectively. There will be one "thread" of inheritance for the user interface (drawing block diagrams, and menu actions when applicable), and one thread of inheritance for the modelling of control systems (components, models, etc.). The development of class libraries, in particular, will benefit from multiple inheritance. For example, functions provided by the operating system and the window manager, will be easier to describe and use in an object-oriented fashion with multiple inheritance.

The model contains terminals and realizations, in C++ represented with linked lists (Listing 3). General purpose lists of components are used, which effectively corrupts the type security in C++. In addition, the programmer must bother about explicit type conversions. Alternatively, generic lists could be faked with macros. Future versions of C++ may incorporate true generics, also called parameterized types [Stroustrup, 1987b]. The need is evident, even in this small example.

```
class model : public component {
  list terminals;
  list realizations;

  void new_terminal();
  void new_realization();

public:
  void menuaction();
  void redraw();
};
```

Listing 3. Definition of the model class.

There are two different kinds of model realizations: primitive realizations based on equations, and structured realizations based on hierarchical block diagrams (Listing 5). There is no "one-of" concept (for example, allowing a pointer to a set of classes) in C++, so an additional class realization is needed (Listing 4). In this case, there are no real problems; in other cases, an awkward data structure might be forced upon the programmer. The one-of concept is available with full type checking in KEE, and has reduced the need for common base classes.

```
class realization : public component {
};
```

Listing 4. The common part of all realizations.

A submodel establishes a relation between two models, one fully enclosed in the other (Listing 6). With a structured realization, a model is described by the behaviour of its submodels and by its connections. The submodel also has a graphical meaning. When a model is simulated, the submodel must be "instantiated" by the model representation framework. Although many submodels may refer to a single model (e. g., a pump), every submodel requires a private data area to hold simulation variables.

```
class eqn_realization : public realization {
  list equations;

  void new_equation();

public:
  void menuaction();
  void redraw();
};

class struct_realization : public realization {
  list  submodels;
  list  connections;

  void new_submodel();
  void new_connection();

public:
  void menuaction();
  void redraw();
};
```

Listing 5. Primitive and structured model realizations.

```
class submodel : public component {
  point  position, size;
  model* parent;
  model* sub;
  void*  data;

public:
  void move();
  void scale();
  void instantiate();
  void redraw();
};
```

Listing 6. Definition of the submodel class.

An example

A small example will demonstrate the data structures above: a servo built from a regulator and a motor. On the screen, the engineer will see a block diagram as in Figure 1. Input to the servo is the reference value, also called the setpoint. Output from the servo is the actual position of the actuator. The regulator controls the motor, but the common feedback loop has been left out to simplify the example.

The textual representation in Figure 2 reveals the most important C++ objects needed for the servo. The servo object has two terminals and a realization (terminals and connections will not be described in more detail). The realization is of course structured, and contains two submodels. It also contains three connections: the reference value imported to the regulator, the control signal from regulator to motor (shown in Figure 2), and the exported actuator position.

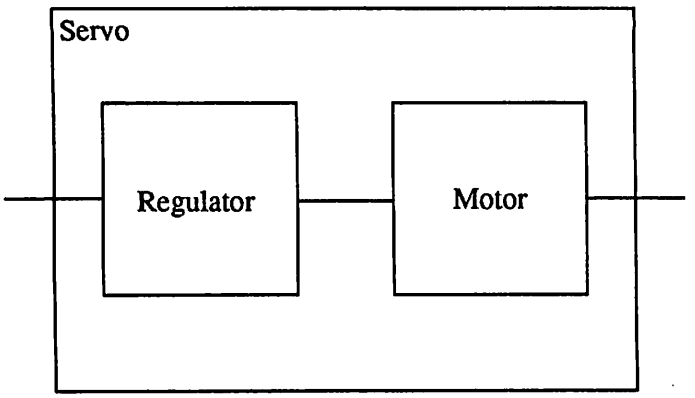


Figure 1. A servo with two submodels.

The submodel objects (for example, **MotorSub**) serve two purposes in this example. Firstly, the graphical appearance of a structured realization is determined mainly by the position and size of the submodels. This information cannot be stored in the model object; a certain kind of motor can be used as a submodel in many different models. Secondly, the submodels establish a relationship between the enclosing model (the servo), and the model objects used as components (e.g., the motor). The two pointers in the submodel object are used, for example, when defining connections. The references between models, realizations and submodels are shown graphically in Figure 3. The role of the submodel when simulating the control system is not discussed here.

The C++ objects used for representing the regulator and the motor are similar to the servo objects. The main difference is that the regulator and the motor have primitive realizations, probably expressed with differential equations.

Exception handling

Handling of exceptions (errors and similar uncommon events) is a problem in all software systems. Ordinary programming techniques, using status flags and if-statements, lead either to bad program structure and cluttered code, or to programs that take proper behaviour for granted. A well designed exception handling mechanism (as in Ada), is an invaluable asset in practical software development. Exceptions increase the readability of the program and indicates the programmer's assumptions about expected and unexpected events [Ghezzi and Jazayeri, 1982, page 22].

The model development tool is quite complex, and many inconsistencies must be checked step-by-step, at different times. Exception handling is useful for restoring the internal data structures to a previous well-defined state. Storing as little redundant information as possible makes this task easier, but may increase complexity in other areas.

The absence of exception handling is a serious flaw of C++. Ada style exception handling, which is also available in C [Lee, 1983], is very effective, but a more flexible scheme may be called for in C++. Some people say that exception handling is needed for developing good class libraries.

Finally, it should be noted that friend functions have been used sparingly (for example, a connection needs free access to terminals and submodels), and proved to be extremely useful. By bending the rules a little, a natural data structure has been maintained; ever-expanding modules because of too strict encapsulation is often a problem with Modula-2 and Ada.

Model: **Servo**
  Terminals: **[Ref, Pos]**
  Realizations: **[ServoRealiz]**

Struct-realization: **ServoRealiz**
  Submodels: **[RegSub, MotorSub]**
  Connections: **[RegSub.u — MotorSub.u, ...]**

Submodel: **RegSub**
  Position: $(-0.6, 0)$
  Size: $(0.5, 0.5)$
  Parent: **→Servo**
  Sub: **→Regulator**

Submodel: **MotorSub**
  Position: $(0.6, 0)$
  Size: $(0.5, 0.5)$
  Parent: **→Servo**
  Sub: **→Motor**

Model: **Regulator**
  Terminals: **[Ref, u]**
  Realizations: **[RegRealiz]**

Eqn-realization: **RegRealiz**
  Equations: $[\cdots]$

Model: **Motor**
  Terminals: **[u, Pos]**
  Realizations: **[MotorRealiz]**

Eqn-realization: **MotorRealiz**
  Equations: $[\cdots]$

Figure 2. Textual representation of the servo; terminals, connections and equations are not shown. Square brackets denote a list, an arrow (→) a pointer reference.

5. Using PHIGS

PHIGS (Programmer's Hierarchical Interactive Graphics Standard) is a new 3D graphics standard, aimed at interactive CAE/CAD applications [Brown, 1985]. PHIGS should be regarded as an extension and a complement to the Graphical Kernel Standard [Hopgood et al., 1983], but not as a replacement.

The basic unit in PHIGS is the structure (cf. segment in GKS). A structure contains elements for drawing, graphical attributes, and transformations. It is possible to build hierarchies of structures (i. e., one structure may call another), and to edit the contents of a structure; this is not possible in GKS. Application data may also be stored in a structure, possibly a useful feature.

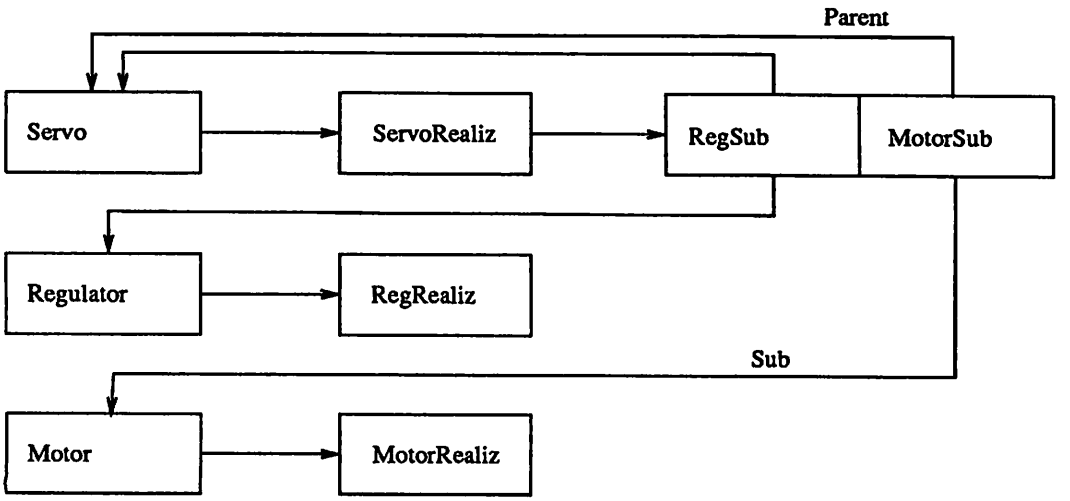


Figure 3. References between models, realizations and submodels of the servo. Terminals, connections and equations are not shown.

In order to take maximum advantage of the hierarchical structures in PHIGS, one structure is associated with every object in the C++ program. This one-to-one correspondence is very convenient; changes are normally localized to a single PHIGS structure, and complete regeneration of the graphics can be avoided. As a typical example, consider changing a pump model: the structure associated with the pump must be changed, but models using the pump as a submodel only refer to a structure identifier, and need no changes. The fine granularity of the graphics hierarchy causes an extra overhead at redraw, which is quite tolerable in this application, though. It can be noted that the model development tool is not a typical PHIGS application, in the sense that it uses the hierarchical features of PHIGS, but not the 3D capabilities.

The correspondence between the object hierarchy and the PHIGS structure hierarchy is shown in Figure 4. The object structure on the left is the same as in Figure 3, but the regulator objects are not shown. A PHIGS structure is associated with each object, as indicated by dashed arrows. The PHIGS structures on the right form a parallel hierarchy, logically connected with "execute structure" primitives. The graphical representation of a model is determined by the realization and its associated structure. The PHIGS structures are in reality more complex, for example, to control picking (see below).

The problem of associating a C++ object with a structure, was solved by some fancy programming. A C++ object can easily refer to a structure by storing the structure identifier, but a problem arises when control must go from a structure to the associated C++ object (for example, when the object's menu action should be invoked). The solution is to use the object's `this` pointer as pick identifier, after conversion to an integer. When the PHIGS system returns a pick identifier, the identifier is converted back to a "pointer to component." The exact nature of the object is not known, but all components implement method `menuaction` (Listing 2).

PHIGS can display graphics on multiple "workstations," which in a workstation environment corresponds to multiple windows. By using so called filters, different graphical representations can be displayed with a single structure hierarchy. Regrettably, multiple workstations are not yet supported by some PHIGS implementations. Event mode input and rubberband lines may also be missing in current implementations. Window management is not available in the PHIGS standard, and may therefore cause considerable practical problems.

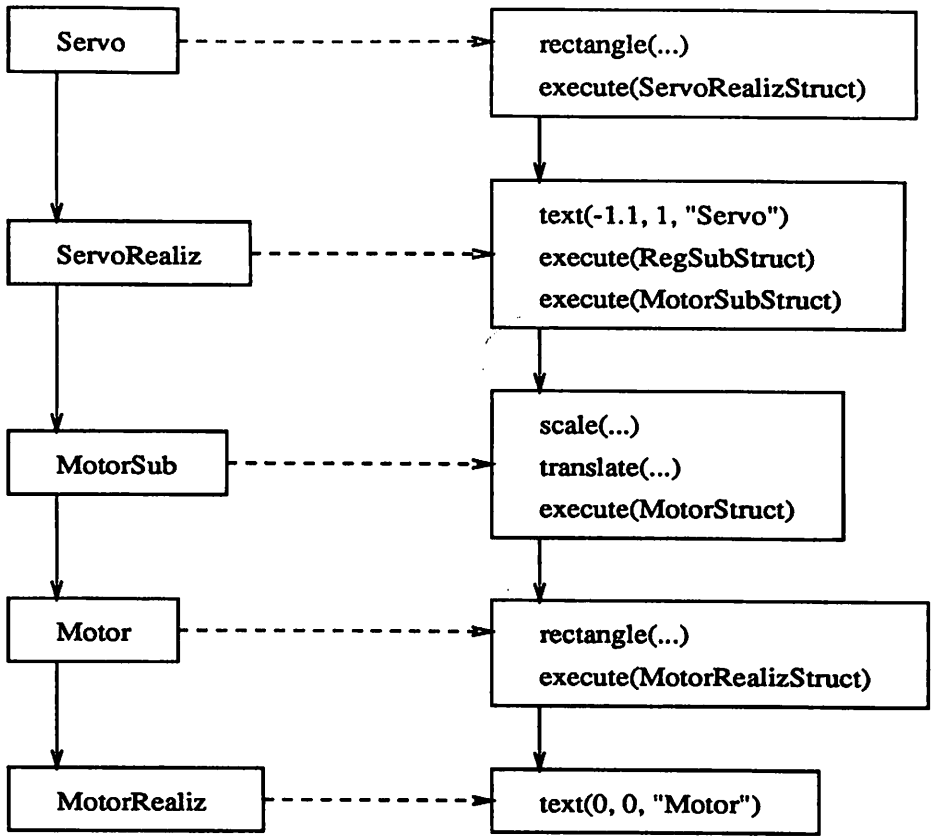


Figure 4. Parallel hierarchies of C++ objects (left) and PHIGS structures (right).

6. Conclusions

In our experience, a dynamic environment like KEE is the best choice for research and rapid prototyping. An engineering tool requires a less expensive and more efficient implementation tool that is available on many computers; in this case, C++ is superior. We have not made a detailed evaluation of KEE versus C++, but the current work shows that programs and data structures using the object-oriented parts of KEE can be implemented in C++ with reasonable effort.

The major difficulty is that C++ does not support dynamic creation of classes. For this reason, models of control systems cannot be directly expressed as classes in C++, so an object-oriented framework must be implemented. The data abstraction and object-oriented programming aspects of C++ provide good support for this framework, and a good programming environment in general. Multiple inheritance, parameterized types and exception handling are much needed extensions to C++.

PHIGS is a powerful new graphics standard, but current implementations need improvement. Window management remains a problem area.

Acknowledgements

I am grateful for many interesting discussions with Sven Erik Mattsson and Mats Andersson, and for comments on the manuscript by Mats Andersson, Ola Dahl and the reviewers. This work was supported by the Swedish National Board for Technical Development (STU).

References

BROWN, MAXINE D. (1985): *Understanding PHIGS*, Template Graphics, San Diego, CA, USA.

GHEZZI, CARLO and MEHDI JAZAYERI (1982): *Programming Language Concepts*, John Wiley & Sons.

HOPGOOD, F. R. A., D. A. DUCE, J. R. GALLOP and D. C. SUTCLIFFE (1983): *Introduction to the Graphical Kernel Standard (GKS)*, Academic Press.

LEE, P. A. (1983): "Exception Handling in C Programs," *Software — Practice and Experience*, **13**, 389–405, May 1983.

MATTSSON, SVEN ERIK (1988a): "On Model Structuring Concepts," *Proc. 4th IFAC Symposium on Computer-Aided Design in Control Systems*, Beijing, P. R. China.

MATTSSON, SVEN ERIK (1988b): "On Modelling and Differential/Algebraic Systems," *Simulation*, Accepted for publication.

NILSSON, BERNT (1987): "Experiences of Describing a Distillation Column in some Modelling Languages," CODEN: LUTFD2/TFRT-7362, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

STROUSTRUP, BJARNE (1987a): "The Evolution of C++: 1985 to 1987," *Proc. USENIX C++ Workshop*, Santa Fe, NM, USA.

STROUSTRUP, BJARNE (1987b): "Possible Directions for C++," *Proc. USENIX C++ Workshop*, Santa Fe, NM, USA.

Type-safe Linkage for C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the problems involved in generating names for overloaded functions in C++ and in linking to C programs. It also discusses how these problems relate to library building. It presents a solution that provides a degree of type-safe linkage. This eliminates several classes of errors from C++ and allows libraries to be composed more freely than has hitherto been possible. Finally the current encoding scheme for C++ names is presented.

1 Introduction

This paper describes the type-safe linkage scheme used by the 2.0 release of C++ and the mechanism provided to allow traditional (unsafe) linkage to non-C++ functions. It describes the problems with the scheme used by previous releases, the alternative solutions considered, and the practicalities involved in converting from the old linkage scheme to the new.

The new scheme makes the `overload` keyword redundant, simplifies the construction of tools operating on C++ object code, makes the composition of C++ libraries simpler and safer, and enables reliable detection of subtle program inconsistencies. The scheme does not involve any run-time costs and does not appear to add measurably to compile or link time.

The scheme is compatible with older C++ implementations for pure C++ programs but requires explicit specification of linkage requirements for linkage to non-C++ functions.

2 The Original Problem

C++ allows overloading of function names; that is, two functions may have the same name provided their argument types differ sufficiently for the compiler to tell them apart. For example,

```
double sqrt(double);
complex sqrt(complex);
```

Naturally, these functions must have different names in the object code produced from a C++ program. This is achieved by suffixing the name the user chose with an encoding of the argument types (the *signature* of the function). Thus the names of the two `sqrt()` functions become:

```
sqrt__Fd                // the sqrt that takes a double argument
sqrt__F7complex         // the sqrt that takes a complex argument
```

Some details of the encoding scheme are described in Appendix A.

When experiments along this line began five years ago it was immediately noticed that for many sets of overloaded functions there was exactly one function of that name in the standard C library. Since C does not provide function name overloading there could not be two. It was deemed essential for C++ to be able to use the C libraries without modification, recompilation, or indirection. Thus the problem became to design an overloading facility for C++ that allowed calls to C library functions such as `sqrt()` even when the name `sqrt` was overloaded in the C++ program.

3 The Original Solution

The solution, as used in all non-experimental C++ implementations up to now, was to let the name generated for a C++ function be the same as would be generated for a C function of the same name whenever possible. Thus `open()` gets the name `open` on systems where C doesn't modify its names on output, the name `_open` on systems where C prepends an underscore, etc.

This simple scheme clearly isn't sufficient to cope with overloaded functions. The keyword `overload` was introduced to distinguish the hard case from the easy one and also because function name overloading was considered a potentially dangerous feature that should not be accidentally or implicitly applied. In retrospect this was a mistake.

To allow linkage to C functions the rule was introduced that only the second and subsequent versions of an overloaded function had their names encoded. Thus the programmer would write

```
overload sqrt;
double sqrt(double);            // sqrt
complex sqrt(complex);          // sqrt__F7complex
```

and the effect would be that the C++ compiler generated code referring to `sqrt` and `sqrt__F7complex`. This enabled a C++ programmer to use the C libraries. This trick solves the problems of name encoding, linkage to C, and protection against accidental overloading, but it is clearly a hack. Fortunately, it was documented only in the BUGS section of the C++ manual page.

4 Problems with the Original Solution

There are at least three problems with this scheme:

- How to name overloaded functions so that one may be a C function.
- How to detect errors caused by inconsistent function declarations.
- How to specify libraries so that several libraries can be easily used together.

The `overload` Linkage Problem

Consider a program that uses an overloaded function `print()` to output `globs` and `widgets`. Naturally `globs` are defined in `glob.h` and `widgets` in `widget.h`. A user writes

```
// file1.c:
#include <glob.h>
#include <widget.h>
```

but this elicits an error message from the C++ compiler since `print()` is declared twice with different argument types. The user then modifies the program to read

```
// file1.c:
overload print;
#include <glob.h>
#include <widget.h>
```

and all is well until someone in some other part of the program writes

```
// file2.c:
overload print;
#include <widget.h>
#include <glob.h>
```

This fails to link since the object code file produced from `file1.c` refers to `print` (meaning `print(glob)`) and `print__F6widget`, whereas the output from `file2.c` refers to `print` (meaning `print(widget)`) and `print__F4glob`.

This is of course a nuisance, but at least the program fails to link and the programmer can – after some detective work based on relatively uninformative linker error messages – fix the problem. The nastier variation of this will happen to the conscientious programmer who knows that `print()` is overloaded and inserts the appropriate `overload` declarations, but happens to use only one variation of `print()` in each of two source files:

```
// file1.c:
overload print;
#include <glob.h>

// file2.c:
overload print;
#include <widget.h>
```

The output from `file1.c` and `file2.c` now both refer to `print`. Unfortunately, in the output from file1.c `print` means `print(glob)` whereas `print` refers to `print(widget)` in the output from file2.c. One might expect linkage to fail because `print()` has been defined twice. However, on most systems this is not what happens in the important case where the definitions of `print(glob)` and `print(widget)` are placed in libraries. Then, the linker simply picks the first definition of `print()` it encounters and ignores the second. The net effect is that calls (silently) go to the wrong version of `print()`. If we are lucky, the program will fail miserably (core dump); if not, we will simply get wrong results.

The requirement that the `overload` keyword must be used and the non-uniform treatment of overloaded functions ("the first overloaded function has C linkage") is a cause of complexity in C++ compilers and in other tools that deal with C++ program text or with object code generated by a C++ compiler.

The General Linkage Problem

This problem of inconsistent linkage is a variation of the general problem that C provides only the most rudimentary facilities for ensuring consistent linkage. For example, even in ANSI C and in C++ (until now) the following example will compile and link without warning:

```
#include <stdio.h>
extern int sqrt(int);

main()
{
        printf("sqrt(%d) == %d\n",2,sqrt(2));
}
```

and produce output like this

```
sqrt(2) == 0
```

because even though the user clearly specified that an integer `sqrt()` was to be used, the C compiler/linker uses the double precision floating point `sqrt()` from the standard library. This problem can be handled by consistent and comprehensive use of correct and complete header files. However, that is not an easy thing to achieve reliably and is not standard practice. The traditional C and C++ compiler/linker systems do not provide the programmer with any help in detecting errors, oversights, or dangerous practices.

These linkage problems are especially nasty because they increase disproportionally with the size of programs and with the amount of library use.

Combining Libraries

The standard header `complex.h` overloads `sqrt()`:

```
// complex.h:
overload sqrt;
#include <math.h>
complex sqrt(complex);
```

Some other header, `3d.h`, declares `sqrt()` without overloading it:

```
// 3d.h:
#include <math.h>
```

Now a user wants both the 3d and the complex number packages in a program:

```
#include <3d.h>
#include <complex.h>
```

Unfortunately this does not compile because of this sequence of operations:

```
double sqrt(double);     // from <math.h> via <3d.h>
overload sqrt;           // from <complex.h>
```

A function that is to be overloaded must be explicitly declared overloaded before its first declaration is processed. So the programmer, who really did not want to know about the internals of those headers, must reorder the #include directives to get the program to compile:

```
#include <complex.h>
#include <3d.h>
```

This will work unless `3d.h` overloads some function, say `atan()`, that `complex.h` does not. Even in that case the programmer can cope with the problem by adding sufficient `overload` declarations where `3d.h` and `complex.h` are included:

```
overload sqrt;
overload atan;
#include <3d.h>
#include <complex.h>
```

This reordering and/or adding of `overload` declarations is irrelevant to the job the programmer is trying to do. Worse, if the extra `overload` declarations were placed in a header file the programmer has now set the scene for the users of the new package to have exactly the same problems when they try combining this new library with other libraries. It becomes tempting to overload all functions or at least to provide header files that overload all interesting functions. This again defeats any real or imagined benefits of requiring explicit `overload` declarations.

5 A General Solution

The overloading scheme used for C++ (until now) interacts with traditional C linkage scheme in ways that bring out the worst in both. Overloading of function names, which was introduced to provide notational convenience for programmers, is becoming a noticeable source of extra work and complexity for builders and users of libraries. Either the idea of overloading is bad or else its implementation in C++ is deficient. The insecure C linkage scheme is a source of subtle and not-so-subtle errors. In summary:

[1] Lack of type checking in the linker causes problems.

[2] Use of the `overload` keyword causes problems.

[3] We must be able to link C++ and C program fragments.

A solution to 1 is to augment the name of *every* function with an encoding of its signature. A solution to 2 is to cease to require the use of `overload` (and eventually abolish it completely). A solution to 3 is to require a C++ programmer to state explicitly when a function is supposed to have C-style linkage.

The question is whether a solution based on these three premises can be implemented without noticeable overhead and with only minimal inconvenience to C++ programmers. The ideal solution would

- require no C++ language changes;
- provide type-safe linkage;
- allow for simple and convenient linkage to C;
- break no existing C++ code;
- allow use of (ANSI style) C headers;
- provide good error detection and error reporting;
- be a good tool for library building;
- impose no run-time overhead;
- impose no compile time overhead;
- impose no link time overhead.

We have not been able to devise a scheme that fulfills all of these criteria strictly, but the adopted scheme is a good approximation.

Type-safe C++ Linkage

First of all, every C++ function name is encoded by appending its signature. This ensures that a program will load only provided every function called has a definition and that the argument types specified in declarations used to compile calls are the same as the types specified in the function definition. For example, given:

```
f(int i) { ... }                // f__Fi
f(int i, char* j) { ... }       // f__FiPc
```

These examples will cause correct linkage:

```
extern f(int);                  // f__Fi   - links to f(int)
f(1);

extern f(int,char*);            // f__FiPc - links to f(int,char*)
f(1,"asdf");
```

These examples will cause linkage errors independent of where in the program they occur because no `f()` with a suitable signature has been defined:

```
// no declaration of f() in this file
// (this is legal only in C programs)
f(1);                           // f      - links to ???

extern f(char*);                // f__FPc - links to ???
f("asdf");

extern f(int ...);              // f__Fie - links to ???
f(1,"asdf");
```

One might consider extending this encoding scheme to include global variables, etc., but this does not appear to be a good idea since that would introduce at least as many problems as it would solve. For example:

```
// file1.c:
int aa = 1;
extern int bb;


//file2.c:
char* aa = "asdf";      // error: aa is declared int in file1.c
extern char* bb;        // error: bb is declared int in file1.c
```

Under the current C scheme, the double definition of `aa` will be caught and the inconsistent declarations of `bb` will not. Using an encoding scheme, the double definition of `aa` would not be caught since the difference in encoding would cause *two* differently named objects to be created – contrary to the rules of C and C++. The fact that the inconsistent declarations of `bb` would be caught by some linkers (not all) does not compensate for the incorrect linkage of `aa`. Consequently only functions are encoded using their signatures.

For a similar reason function argument types are *not* encoded (except for pointer to argument types):

```
// hypothetical encoding using return types:

// file1.c:
int f() { ... };        // f__Fv_i

//file2.c:
char* f();              // f__Fv_Pc
```

Here a linker would report `f()` undefined because of the name mismatch. This could be quite confusing.

The adopted linkage scheme is much safer than what is currently used for C, but it cannot detect all linkage problems. For example, if two libraries each provides a function `f(int)` as part of their public

interface there is no mechanism that allows the compiler to detect that there are supposed to be two different `f(int)`s. If the .o files are loaded together the linker will detect the error, but when a library search mechanism is employed the error may go undetected.

Note that this linking scheme simply enforces the C++ rules that every function must be declared before it is called and that every declaration of an external name in C++ must have exactly the same type.

In essence, we use the name encoding scheme to "trick" the linker into doing type checking of the separately compiled files. More comprehensive solutions can be achieved by modifying the linker to understand C++ types. For example, a linker could check the types of global data objects and the return types of functions. It might also provide features for ensuring the consistency of global constants and classes. However, getting an improved linker into use is typically a hard and slow process. The scheme presented here is portable across a great range of systems and can be used immediately.

Implicit Overloading

If a function is declared twice with different argument types it is overloaded. For example:

```
double sqrt(double);
complex sqrt(complex);
```

is accepted without any explicit `overload` declaration. Naturally, `overload` declarations will be accepted in the foreseeable future; they are simply not necessary any more.

Does this relaxation of the C++ rules cause new problems? It does not appear to. For example, originally I imagined that obvious mistakes such as

```
double sqrt(double);              // sqrt__Fd
double d = sqrt(2.3);

double sqrt(int d) { ... }        // sqrt__Fi
```

would cause hard-to-find errors. It certainly would with the traditional C linkage rules, but with type-safe linkage the program simply will not link because there is no function called `sqrt__Fd` defined anywhere. Even the standard library function will not be found because its name is as always "plain" `sqrt`.

Another imagined problem was that a call

```
f(x);
```

would suddenly change its meaning when a function became overloaded by the inclusion of a new header file containing the declaration of another function `f()`. The only case where `f(x)` can have its meaning changed by the introduction of a new declaration `f(T)` is where `T` is the type of `x`. In this case the meaning of `f(x)` ought to change. In all other cases, the C++ ambiguity rules ensure that the introduction of a new `f()` will either leave the meaning of `f(x)` unchanged (when the new `f()` is unrelated to the type of `x`) or will cause a compile time error (when an ambiguity is introduced).

C Linkage

This leaves the problem of how to call a C function or a C++ function "masquerading" as a C function. To do this a programmer must state that a function has C linkage. Otherwise, a function is assumed to be a C++ function and its name is encoded. To express this an extension of the "extern" declaration is introduced into C++:

```
extern "C" {
        double sqrt(double);      // sqrt(double) has C linkage
}
```

This linkage specification does not affect the semantics of the program using `sqrt()` but simply tells the compiler to use the C naming conventions for the name used for `sqrt()` in the object code. This means that the name of *this* `sqrt()` is `sqrt` or `_sqrt` or whatever is required by the C linkage conventions on a given system. One could even imagine a system where the C linkage rules were the type-safe C++ linkage rules as described above so that the name of `sqrt()` was `sqrt__Fd`.

Linkage specifications nest, so that if we had other linkage conventions, such as Pascal linkage, we could write:

```
                                        // default: C++ linkage here
extern "C" {
                                        // C linkage here
        extern "Pascal" {
                                        // Pascal linkage here
                extern "C++" {
                                        // C++ linkage here
                }
                                        // Pascal linkage here
        }
                                        // C linkage here
}
                                        // C++ linkage here
```

Such nestings will typically occur as the result of nested `#includes`.

The { } in a linkage specification does *not* introduce a new scope; the braces are simply used for grouping. This use of { } strongly resembles their use in enumerations.

The keyword `extern` was chosen because it is already used to specify linkage in C and C++. Strings (for example, `"C"` and `"C++"`) were chosen as linkage specifiers because identifiers (e.g. `C` and `Cplusplus`) would de facto introduce new keywords into the language and because a larger alphabet can be used in strings.

Naturally, only one of a set of overloaded functions can have C linkage, so the following causes a compile time error:

```
extern "C" {
        double sqrt(double);
        complex sqrt(complex);
}
```

Note that C linkage can be used for C++ functions intended to be called from C programs as well as for C functions. In particular, it is necessary to use C linkage for C++ functions written to implement standard C library functions for use by C programs. However, using the encoded C++ name from C preserves type-safety at link time. This technique can be valuable in other languages too. I have already seen an example of the C++ scheme applied to assembly code to prevent nasty link errors for low level routines. One might consider using this C++ linkage scheme for C also, but I suspect that the sloppy use of type information in many C programs would make that too painful.

In an "all C++" environment no linkage specifications would be needed. The linkage mechanism is intended to ease integration of C++ code into a multi-lingual system.

Caveat

One could extend this linkage specification mechanism to other languages such as Fortran, Lisp, Pascal, PL/1, etc. The way such an extension is done should be considered very carefully because one "obvious" way of doing it would be to build into a C++ compiler the full knowledge of the type structure and calling conventions of such "foreign" languages. For example, a C++ compiler *might* handle conversion of zero-terminated C++ strings into Pascal strings with a length prefix at the call point of a function with Pascal linkage and *might* use Fortran call by reference rules when calling a function with Fortran linkage, etc.

There are serious problems with this approach:

- The complexity and speed of a C++ compiler could be seriously affected by such extensions.
- Unless an extension is widely available and accepted programs using it will not be portable.
- Two implementations might "extend" C++ with a linkage specification to the same "foreign" language, say Fortran, in different ways so as to make identical C++ programs have subtly different effects on different implementations.

Naturally, these problems are not unique to linkage issues or to this approach to linkage specification.

I conjecture that in most cases linkage from C++ to another language is best done simply by using a

common and fairly simple convention such as "C linkage" plus some standard library routines and/or rules for argument passing, format conversion, etc., to avoid building knowledge of non-standard calling conventions into C++ compilers. This ought to be simpler from C++ than from most other languages. For example, reference type arguments can be used to handle Fortran argument passing conventions in many cases and a Pascal string type with a constructor taking a C style string can trivially be written. Where extension are unavoidable, however, C++ now provides a standard syntax for expressing them.

6 Experience

The natural first reaction to this scheme is to look for a way of handling linkage and overloading without requiring explicit linkage specifications. We have not been able to come up with a system that enabled C linkage to be implicit without serious side effects. I will summarize the advantages of the adopted scheme here and discuss several possible objections to it. Section 7 below describes alternative schemes that were considered and rejected.

Making Linkage Specifications Invisible

One obvious advantage of this scheme is that it allows a programmer to give a set of functions C linkage with a single linkage specification without modifying the individual function declarations. This is particularly useful when standard C headers are used. Given a C header (that is, an ANSI C header with function prototypes, etc.),

```
// C header:
// C declarations
```

one can trivially modify the header for use from C++:

```
// C++ header:

extern "C" {
        // C header:
        // C declarations
}
```

This creates a C++ header that cannot be shared with C.

Sharing with C can be achieved using `#ifdef`:

```
// C and C++  header:

#ifdef __cplusplus
extern "C" {
#endif
        // C header:
        // C declarations
#ifdef __cplusplus
}
#endif
```

where `__cplusplus` is defined by every C++ compiler.

In cases where one for some reason cannot or should not modify the header itself one can use an indirection:

```
// C++ header:

extern "C" {
#include "C_header"
}
```

Fortunately, such transformations can be done by trivial programs so that most of the effort in converting C headers need not be done by hand.

It was soon discovered that even though programmers tend to scatter function declarations throughout the C++ program text, most C functions actually come from well-defined C libraries for

which there are – or ought to be – standard header files.

Placing all of the necessary linkage specifications in standard header files means that they are not seen by most users most of the time. Except for programmers studying the details of C library interfaces, programmers installing headers for new C libraries for C++ users, and programmers providing C++ implementations for C interfaces, the linkage specifications are invisible.

Error Handling

The linker detects errors, but reports them using the names found in the object code. This can be compensated for by adding knowledge about the C++ naming conventions to the linker or (simpler) by providing a filter for processing linker error messages. This output was produced by such a filter:

```
C++ symbol mapping:

PathListHead::~PathListHead()          __dt__12PathListHeadFv
Path_list::sepWork()                   sepWork__9Path_listFv
Path::pathnorm()                       pathnorm__4PathFv
Path::operator&(Path&)                 __ad__4PathFR4Path
Path::first()                          first__4PathFv
Path::last()                           last__4PathFv
Path::rmfirst()                        rmfirst__4PathFv
Path::rmlast()                         rmlast__4PathFv
Path::rmdots()                         rmdots__4PathFv
Path::findpath(String&)                findpath__4PathFR6String
Path::fullpath()                       fullpath__4PathFv
```

Introducing this filter had the curious effect of replacing the usual complaint about "ugly C++ names" with complaints that the linker didn't provide enough information about C functions and global data objects.

The reason for presenting the encoded and unencoded names of undefined functions side by side is to help users who use tools, such as debuggers, that haven't yet been converted to understand C++ names.

A plain C debugger such as `sdb`, `dbx`, or `codeview` can be used for C++ and will correctly refer to the C++ source, but it will use the encoded names found in the object code. This can be avoided by employing a routine that "reverses" the encoding, that is, reads an encoded name and extracts information from it†. The encoding scheme is described in Appendix A. A C++ name decoder should be generally available for use by debugger writers and others who deal directly with object code. Until such decoders are in widespread use the programmer must have at least a minimal understanding of the encoding scheme.

Upgrading Existing C++ Programs

Decorating the standard header files with the appropriate linkage specifications had two effects. The first phenomenon observed was that most of the declarations scattered in the program text that were referring to C functions were either redundant (because the function had already been declared in a header) or at least potentially incorrect (because they differed from the declaration of that header file on some commonly used system). The second phenomenon observed was that every non-trivial program converted to the new linkage system contained inconsistent function declarations. A noticeable number of declarations found in the program text were plain wrong, that is, different from the ones used in the function definition. This was caused in part by sloppiness, for example, where a programmer had declared a function

```
char* f(int ...);
```

to suppress compiler warnings instead of looking up the type of the second argument. A more common problem was that the "standard" header files had changed since the function declaration was placed in

† Naturally, this would be the same function as was used to write the linker output filter. The examples here are based on the name decoding routine written by Steve Brandt and used to modify the UNIX System V C debugger `sdb` into `sdb++`.

the text so that the "local" declaration didn't match any more; this often happens when a file is transferred from one system to another, say from a BSD to a System V.

In summary, introducing the new linkage system involved adding linkage specifications. Typically, these linkage specifications were only needed in standard header files. The process of introducing linkage specifications invariably revealed errors in the programs – even in programs that had been considered correct for years. The process strongly resembles trying `lint` on an old C program.

As was expected, some programmers first tried to get around the requirements for explicit C linkage by enclosing their entire program in a linkage directive. This might have been considered a fine way of converting old C++ programs with minimum effort had it not had the effect of ensuring that every program that uses facilities provided by such a program would also have to use the unsafe C linkage. To achieve the benefits from the new linkage scheme most C++ programs must use it. The requirement that at most one of a set of overloaded functions can have C linkage defeats this way of converting programs. The slightly slower and more involved method of using standard header files (already containing the necessary linkage specifications) and adding a few extra linkage specifications in local headers where needed must be used. This also has the benefit of unearthing unexpected errors.

7 Details

The scope of C function declarations has always been a subject for debate. In the context of C++ with linkage specifications and overloaded functions it seems prudent to answer some variations of the standard questions.

Default Linkage

Consider:

```
extern "C" {
        int f(int);
}

int f(int);     // default (C++ linkage) overruled: f() has C linkage
```

Is it the same `f()` that was defined with C linkage above and does it have C or C++ linkage? It is the same `f()` and it does (still) have C linkage. The first linkage specification "wins" provided the second declaration has "only" default (that is, C++) linkage.

Where linkage is explicitly specified for a function, that specification must agree with any previous linkage. For example:

```
extern "C" {
        int f(int);     // f() has C linkage
}

int g();                // default: g() has C++ linkage
int f(int);             // fine: default overruled, f() has C linkage

extern "C" {
        int f(int);     // fine
        int g();        // error: inconsistent linkage specification
}
```

The reason to require agreement of explicit linkage specifications is to avoid unnecessary order dependencies. The reason to allow a second declaration with implicit C++ linkage to take on the linkage from a previous explicit linkage specification is to cope with the common case where a declaration occurs both in a `.c` file and in a standard header file.

Declarations in Different Scopes

Consider:

```
extern "C" {
        int f(int);
}

void g1()
{
        int f(int);
        f(1);
}
```

Is the `f()` declared local to `g1` the same as the global `f()` and does the function called in `g1()` have C linkage? It is the same `f()` and it does have C linkage.

Consider:

```
extern "C" {
        int f(int);
}

void g2()
{
        int f(char*);
        f(1);
        f("asdf");
}
```

Does the local declaration of `f()` overload the global `f()` or does it hide it? In other words, is the call `f(1)` legal? That call is an error because the local declaration introduces a new `f()` that hides the global `f()`. In the tradition of C, the declaration of `f(char*)` also draws an warning.

Consider:

```
void g3()
{
        int ff(int);
};

void g4()
{
        int ff(char*);
        ff("asdf");
        ff(1);
};
```

Does the second declaration of `ff()` overload the first? In other words, is the call `ff(1)` legal? The call is an error and a warning is issued about the two declarations of `ff()` because (as in the example above) overloading in different scopes is considered a likely mistake.

Local Linkage Specification

Linkage specifications are *not* allowed inside function definitions. For example:

```
void g5()
{
        extern "C" {    // error: linkage specification in function
                int h();
        }
}
```

The reason for this restriction is to discourage the use of local declarations of C functions and to simplify the language rules.

8 Alternative Solutions

So, the linkage specification scheme works, but isn't there a better way of achieving the benefits of that scheme? Several schemes were considered. This section presents the first two or three alternatives people usually come up with and explains why we rejected them. Naturally, we also considered more and weirder solutions, but all the plausible ones were variations of the ones presented here.

The Scope Trick

The first attempt to provide type-safe linkage involved the use of `overload` and the C++ scope rules. All overloaded function names were encoded, but non-overloaded function names were not. This scheme had the benefit that the linkage rules for most functions were the C linkage rules – and had the problem that those rules are unsafe. The most obvious problem was that at first glance there is no way of linking an overloaded function to a standard C library function. This problem was handled using a "scope trick":

```
overload sqrt;
complex sqrt(complex);
inline double sqrt(double d)
{
        extern double sqrt(double);     // A completely new sqrt()
                                        // not overloaded

        return sqrt(d);                 // not a recursive call
                                        // but a call of the C function
                                        // sqrt
}
```

In effect, we provided a C++ calling stub for the C function `sqrt()`. The snag is that having thus *defined* `sqrt(double)` in a standard header a user cannot provide an alternative to the standard version. The problems with library combination in the presence of `overload` are not addressed in this scheme, and are actually made worse by the proliferation of definitions of overloaded functions in header files. In particular, if two "standard" libraries each overload a function then these two libraries cannot be used together since that function will be defined twice: once in each of the two standard headers.

There is also a compile time overhead involved. In retrospect, I consider this scheme somewhat worse than the original "the first overloaded has C linkage" scheme.

C "storage class"

It is clear that the definitions providing a calling stub are redundant. We could simply provide a way of stating that a member of a set of overloaded functions should be a C function. For example:

```
complex sqrt(complex);
cdecl double sqrt(double);      // sqrt(double) has C linkage
```

This is equivalent to

```
complex sqrt(complex);
extern "C" {
        double sqrt(double);
}
```

but less ugly. However, it involves complicating the C++ language with yet another keyword. Functions from other languages will have to be called too and they each have separate requirements for linkage so the logical development of this idea would eventually make `ada`, `fortran`, `lisp`, `pascal`, etc., keywords. Using a keyword also requires modification of the declarations of the C functions and those are exactly the declarations we would want *not* to touch since they will typically live in header files shared with an ANSI C compiler. In some cases we would even like not to touch a file in which such declarations reside.

Overload "storage class"

The use of a keyword to indicate that a function is a C function is logically very similar to the linkage specification solution, though inferior in detail. An alternative is to have a keyword indicate that a function should have its signature added. The keyword `overload` might be used. For example:

```
overload complex sqrt(complex);          // use C++ linkage
double sqrt(double);                     // C linkage by default
```

This has the disadvantage that the programmer has to add information to gain type safety rather than having it as default and would de facto ensure that the C++ type-safe linkage rules would be used only for overloaded functions. Furthermore, this would mean that libraries could be combined only if the designers of these libraries had decorated all the relevant functions with `overload`. This scheme also invalidates all old C++ programs without providing significant benefits.

Calling Stubs

One way of dealing with C linkage would be *not* to provide any facilities for it in the C++ language, but to require every function called to be a C++ function. To achieve this one would simply re-compile all libraries and have one version for C and another for C++. This is a lot of work, a lot of waste, and not feasible in general. In the cases where recompilation of a C program as a C++ program is not a reasonable proposition (because you don't have the source, because you cannot get the program to compile, because you don't have the time, because you don't have the file space to hold the result, etc.) you can provide a small dummy C++ function to call the C function. Such a function would be written in C (for portability) or in assembler (for efficiency). For example:

```
double sqrt__Fd(d) double d; /* C calling stub for sqrt(double): */
{
        extern double sqrt();
        return sqrt(d);
}
```

A program can be provided to read the linker output and produce the required stubs.

This scheme has the advantage that the user works in what appears to be an "all C++" environment (but so does the adopted scheme once a few C libraries have been recompiled with C++ and/or a few header files have been decorated with linkage specifications). It does, however, also suffer from a few severe disadvantages. A "C calling stub maker" program cannot be written portably. Therefore, it would become a bottleneck for porting C++ implementations and C++ programs and thus a bottleneck for the use of C++. It is also not clear that this approach can be implemented everywhere without loss of efficiency since it requires large numbers of functions to have two names (a C name and a C++ name). This takes up code space and introduces large numbers of extra names that would slow down programs reading object files such as linkers, loaders, debuggers, etc. The C calling interfaces would also be ubiquitous and available for anyone to use by mistake, thus re-introducing the C linkage problems in a new guise.

Encode only C++ Functions

The fundamental problem with all but the last scheme outlined above is that they require the programmer to decorate the source code with directives to help the compiler determine which functions are C functions. Ideally, the compiler would simply look at the program and determine the linkage necessary for each individual function based on its type. Could the compiler be that smart? Unfortunately, no. There is no way for the compiler to know whether

```
extern double sqrt(double);
```

is written in C or C++. However, one might handle most cases by the heuristic that if a function is clearly a C++ function it gets C++ linkage and if it isn't it gets C linkage. For example:

```
complex sqrt(complex);  // clearly C++: sqrt__F7complex
double sqrt(double);    // could be C:  sqrt
```

Since `complex` is a class, `sqrt(complex)` is clearly a C++ function and it is encoded. The other `sqrt()` might be C so it isn't.

Applying this heuristic would mean that most functions would not have type-safe linkage – but we are used to that. It would also mean that overloading a function based on two C types would be impossible or require special syntax:

```
int max(int,int);
double max(double,double);
```

Such overloading *must* be possible because there are many such examples and several of those are important, especially when support for both single and double precision floating point arithmetic becomes widespread:

```
float sqrt(float);
double sqrt(double);
```

This implies that either `overload` or linkage specifications must be introduced to handle such cases. The heuristic nature of the specification of where these directives are needed will lead to confusion, overuse, and errors.

If `overload` is re-introduced, the cautious programmer will use it systematically wherever a relatively simple class is used (in case a revision of the system should turn it into a plain C struct), wherever an argument is typedef'd (because that `typedef` might some day refer to a plain C type), and wherever there is any doubt. This will lead to the now well known problems of combining libraries. Similarly, if linkage specifications are required anywhere, they will proliferate because of doubts about where they are needed.

It does not seem wise to refrain from checking linkage in a large number of cases and to introduce a rather arbitrary heuristic into the linking rules for C++ without being able to reduce the complexity of the language or to reduce the burden on the programmer somewhere.

Nothing

Naturally, while considering these alternative schemes the easy option of doing nothing was regularly re-considered. However, the original scheme still suffers from the problems described in section 4: insecure linkage, spurious `overload` declarations, overloading rules that complicate the life of library writers and library users, and unnecessary complexity for tools builders.

9 Syntax Alternatives

The scheme of giving all C++ functions type-safe linkage and providing a syntax for expressing that a given function is to have C linkage was thus chosen and tried. However, there were still several alternatives for expressing C linkage for this general scheme.

Why `extern`?

Instead of employing the existing keyword `extern` we might have introduced a new one such as `linkage` or `foreign`. The introduction of a new keyword always breaks some programs (though usually not in any serious way and for a well chosen new keyword not many programs) and `extern` already has the right meaning in C and C++. In almost all cases `extern` is redundant since external linkage is the default for global names and for locally declared functions. When used, `extern` simply emphasizes the fact that a name should have external linkage. The use of `extern` introduced here merely allows the programmer to tag an `extern` declaration with information of *how* that linkage is to be established.

Linkage for Individual Functions

One obvious alternative is to add the linkage specification to each individual function:

```
extern "C" double sqrt(double); // sqrt(double) has C linkage
```

The advantage of this scheme is that the linkage is obvious from looking at an individual function

declaration. The problem with this is that it does not serve the need to be able to give a set of C functions C linkage with one declaration and requires the declaration of every C function to be modified. In particular, it does not allow a C header (that is, an ANSI C header) to be used from a C++ program in such a way that all the functions declared in it get C linkage.

This notation for linkage specification of individual functions is not just an alternative to the linkage "block" adopted but also an obvious extension to the adopted syntax. After observing the use of linkage blocks for a while and listening to the comments from users this extension was adopted.

```
extern "C" double sqrt(double); // sqrt(double) has C linkage
```

is by definition equivalent to

```
extern "C" { double sqrt(double); }// sqrt(double) has C linkage
```

Naturally, a linkage specification applies to all members of a declaration list:

```
extern "C" double sin(double), cos(double);      // sin and cos have C linkage
```

Linkage Pragmas

The original implementation of the linkage specifications used a `#pragma` syntax:

```
#pragma linkage C
double sqrt(double); // sqrt(double) has C linkage
#pragma linkage
```

This was considered too ugly by many but did appear to have significant advantages. For example, it can be argued that linkage to "foreign languages" is not part of the language proper. Such linkage cannot be specified once and for all in a language manual since it involves the *implementations* of *two* languages on a given system. Such implementation specific concepts are exactly what pragmas were introduced into Ada and ANSI C to handle. The `#pragma` syntax was trivial to implement and easy to read. It was also ugly enough to discourage overuse and to encourage hiding of linkage specifications in header files.

There are problems with this view, though. For example, it is most often assumed that any `#pragma` can be ignored without affecting the meaning of a program. This would not be the case with linkage pragmas. Another problem is that for the moment many C implementations do not support a pragma mechanism and it is not certain that those that do can be relied upon to "do the right thing" for linkage pragmas used by a C generating C++ compiler.

Linkage to a particular foreign language does not belong in C++ because such linkage will in principle be local to a given system and non-portable. However, the fact that linkage to other languages occurs is a general concept that can and ought to be supported by a language intended to be used in multi-language environments. In practice, one can assume that at least C and Fortran will be available on most systems where C++ is used and that a large group of users will need to call functions written in these languages. Consequently, one would expect C++ implementations to support C and Fortran linkage.

The fact that C (like most other languages) does not provide a concept of linkage to program fragments written in other languages led to the absence of an explicit linkage mechanism in C++ and to the problems of link safety and overloading.

Special Linkage Blocks

Another approach would be to introduce a new keyword, say `linkage`, and use it to specify both the start and the end of a linkage block:

```
linkage("C");
double sqrt(double); // sqrt(double) has C linkage
linkage("");
```

This avoids introducing yet another meaning for `{}`, allows setting and restoring of linkage to be two separate operations, allows all linkage directives to be found by simple pattern matching in a line oriented editor, and allows all linkage directives to be suppressed by a single macro

```
#define linkage(a)
```

The problem with this seems to be that it tempts people to think of as linkage as a compiler "mode" that can be switched on and off at random times and doesn't obey block structure. For example:

```
linkage("C");

double sqrt(double); // sqrt(double) has C linkage

f() {
        extern g();     // g() has C linkage
linkage("");
        extern h();     // h() has C++ linkage
        ...
}
```

It also becomes hard to convince people that linkage specifications come in pairs and can be nested.

The same approach, with the same educational problems, can be tried without introducing a new keyword:

```
extern "C";
double sqrt(double); // sqrt(double) has C linkage
extern "";
```

Note that whatever syntax was chosen, linkage specifications were intended to obey block structure to fit cleanly into the language. In particular, if linkage "blocks" and ordinary blocks were not obliged to nest, the job of writers of tools manipulating C++ source text, such as a C++ incremental compilation environment, would be needlessly complicated.

10 Conclusions

The use of function name encodings involving type signatures provides a significant improvement in link safety compared to C and earlier C++ implementations. It enables the (eventual) abolition of the redundant keyword `overload` and allows libraries to be combined more freely than before. The use of linkage specifications enables relatively painless linkage to C and eventually to other language as well. The scheme described here appears to be better than any alternative we have been able to devise.

11 Acknowledgements

The new linkage and overloading scheme was essentially a joint effort of Andrew Koenig, Doug McIlroy, Jerry Schwarz, Jonathan Shopiro, and me. Brian Kernighan made many useful comments. The name encoding scheme is based on a proposal by Stan Lippman and Steve Dewhurst with input from Andrew Koenig and me. Steve Dewhurst, Margaret Ellis, Georges Gonthier, Bill Hopkins, Jim Howard, Mike Mowbray, Tim O'Konski, and Roger Scott also made valuable comments on earlier versions on this paper.

Appendix A: The Function Name Encoding Scheme

The (revised) C++ function name encoding scheme was originally designed primarily to allow the function and class names to be reliably extracted from encoded class member names. It was then modified for use for *all* C++ functions and to ensure that relatively short encodings (less than 31 characters) could be achieved reliably for systems with limitations on the length of identifiers seen by the linker. The description here is just intended to give an idea of the technique used, not as a guide for implementers.

The basic approach is to append a function's signature to the function name. The separator `__` is used so a decoder could be confused by a name that contained `__` except as an initial sequence, so don't use names such as `a__b__c` in a C++ program if you like your debugger and other tools to be able to decompose the generated names.

The encoding scheme is designed so that it it easy to determine

- if a name is an encoded name;

- what (unencoded) name the user wrote;
- what class (if any) the function is a member of;
- what are the types of the function arguments.

The basic types are encoded as

```
void          v
char          c
short         s
int           i
long          l
float         f
double        d
long double   r
...           e
```

A global function name is encoded by appending `__F` followed by the signature so that `f(int,char,double)` becomes `f__Ficd`. Since `f()` is equivalent to `f(void)` it becomes `f__Fv`.

Names of classes are encoded as the length of the name followed by the name itself to avoid terminators. For example, `x::f()` becomes `f__1xFv` and `rec::update(int)` becomes `update__3recFi`.

Type modifiers are encoded as

```
unsigned      U
const         C
volatile      V
signed        S
```

so `f(unsigned)` becomes `f__FUi`. If more than one modifier is used they will appear in alphabetical order so `f(const signed char)` becomes `f__FCSc`.

The standard modifiers are encoded as

```
pointer         *       P
reference       &       R
array           [10]    A10_
function        ()      F
ptr to member   S::*    M1S
```

So `f(char*)` becomes `f__FPc` and `printf(const char* ...)` becomes `printf__FPCce`.

Function return types are encoded for arguments of type *pointer to function*. The return type appears after the argument types preceded by a single underscore; for example, `f(int(*)(char*))` becomes `f__FPFPc_i`. The return type is not encoded except for pointer to argument types (see §5).

To shorten encodings repeated types in an argument list are not repeated in full; rather, a reference to the first occurrence of the type in the argument list is used. For example:

```
f(complex,complex);      // f__F7complexT1
        // the second argument is of the same type as argument 1

f(record,record,record,record);          // f__F6recordN31
        // the 3 arguments 2, 3, and 4 are of the same type as argument 1
```

A slightly different encoding is used on systems without case distinction in linker names. On systems where the linker imposes a restriction on the length of identifiers, the last two characters in the longest legal name are replaced with a hash code for the remaining characters. For example, if a 45 character name is generated on a system with a 31 character limit, the last 16 characters are replaced by a 2 character hash code yielding a 31 character name.

Naturally, the encoding of signatures into identifier of limited length cannot be perfect since information is destroyed. However, experience shows that even truncation at 31 characters for the old and less dense encoding was sufficient to generate distinct names in real programs. Furthermore, one can often rely on the linker to detect accidental name clashes caused by the hash coding. The chance of an undetected error is orders of magnitude less than the occurrence of known problems such as C

programmers accidentally choosing identical names for different objects in such a way that the problem isn't detected by the compiler or the linker.

Implementing A Logic-Based Executable Specification Language in C++

Peter A. Kirslis

AT&T Bell Laboratories
11900 North Pecos St.
Denver, Colorado 80234

Robert B. Terwilliger

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

ABSTRACT

PK/C++ is an object-oriented, logic-based, wide-spectrum executable specification language which supports software development by incremental refinement. PK/C++ is part of ENCOMPASS, an environment that supports software development using formal techniques similar to the Vienna Development Method (VDM). PK/C++ can be viewed as a combination of C++ and pure Prolog; it has the syntax of C++. Using PK/C++, software components are first specified using a combination of conventional programming languages and predicate logic. These abstract components are then incrementally refined into components in an implementation language. PK/C++ specifications may be used in proofs of correctness. They are also executable; therefore, initial specifications can be validated and refinements can be verified using testing-based techniques. We believe the use of PK/C++ will enhance the development process. The choice of C++ as the implementation language facilitated this work. Inheritance and virtual functions support code sharing between our data types, and operator overloading permits us to incorporate features of Prolog into our implementation; all without requiring extensions to the syntax of C++. In this paper, we describe the implementation of PK/C++ in reasonable detail and give examples of its use.

1. Introduction

The efficient production of software remains in most cases an elusive goal. One of the most important problems is *quality*; many of the systems produced do not satisfy their purchasers in either functionality, performance or reliability. Depending on the model of software development used, the software quality problem can be subdivided in a number of ways. Initially, a system exists only as an idea in the minds of its users, purchasers, or producers. In our model, the first step in the development process is the creation of a *specification* which precisely

This research was supported in part by NSF Grant CCR-8809418, and by a gift from AT&T.

describes the properties and qualities of the software to be constructed [4]. Unfortunately, current methods do not guarantee that the specification correctly or completely describes the customers' desires. A specification is *validated* when it is shown to correctly state the customers' requirements [4]. It has been suggested that *prototyping* and the use of *executable specification languages* can enhance the communication between customers and developers [15]; providing prototypes for experimentation and evaluation should enhance the validation process. In general, the specification need not be executable; it must be translatable into an implementation. Depending on the method used for translation, the exact relationship between the specification and implementation may be unknown. An implementation is *verified* when it is shown to satisfy its specification [4]. Many techniques can be used to certify this relationship including testing [8], technical review [3], and formal methods [7].

The ENCOMPASS environment supports incremental software development using executable specification languages [9, 11, 14]. In ENCOMPASS, the general approach is to address the validation problem using executable specifications and to address the verification problem using a combination of formal, testing, and peer review techniques. ENCOMPASS automates these techniques and integrates them smoothly into the traditional life-cycle. ENCOMPASS supports a *phased* or *waterfall* life-cycle [4], extended to allow the use of executable specifications and VDM: a separate phase is added for user validation, and the design and implementation processes are combined into a single refinement phase. In ENCOMPASS, a development progresses through the phases: planning, requirements definition, validation, refinement and system integration. ENCOMPASS provides support for all aspects of this development paradigm including simple tools for configuration control [6] and project management [1].

In ENCOMPASS, software is specified using a combination of natural language and the PLEASE [10, 12, 13] family of wide-spectrum, executable specification languages. The design of these languages is a compromise between a number of conflicting goals. First, PLEASE must allow the specification of software using pre- and post-conditions written in predicate logic; the more powerful the specification method, the better. Second, the language must allow the rapid, automatic construction of executable prototypes from these specifications; the prototypes should be as efficient as possible. Unfortunately, there is a conflict between these goals. A fairly powerful specification method would use pre- and post-conditions written in the full first-order, predicate logic. A resolution theorem prover for first-order logic could be used to construct prototypes; however, the performance of these prototypes would be very poor. The emergence of logic programming as a technology, most notably Prolog [2], suggests a good compromise. Although in one sense not as powerful as full first-order logic, Prolog allows much more efficient implementation techniques to be used. By restricting the specifications to a logic with an efficient, Prolog-style implementation, reasonable specification power is combined with implementation efficiency.

Our approach to executable specifications has changed dramatically since we began our work. Our initial vision was of a purely declarative specification language and an extremely intelligent translation system that would automatically produce Prolog procedures from predicate logic assertions. Our experience has led us to believe that this is not a realistic approach in the short term. Therefore, the

latest member of the PLEASE family, PK/C++ [13] (Please Kernel on C++), differs from its predecessor by being based on C++ rather than Ada, by having an operational as well as declarative semantics, and by being based on flat (unification but no backtracking) rather than standard Prolog. We feel these changes will significantly enhance its use as a practical specification vehicle. Being based on C++ rather than Ada makes the power of object-oriented programming available. Having an operational as well as declarative semantics allows the programmer to hand optimize PK/C++ specifications for improved execution. Basing our execution strategy on flat rather than standard Prolog allows much simpler and somewhat more efficient implementation techniques to be used.

PK/C++ supports the *rigorous* [5] development of programs. Although detailed mechanical proofs are not required at every step, the framework is present so that they can be constructed if necessary. Proof techniques may be used that range from a very detailed, completely formal proof using mechanical theorem proving to a development "annotated" with unproven verification conditions. Detailed, mechanical verification may be used on parts of a project, while other, less critical parts may be handled using less expensive techniques. Our experience so far leads us to believe that the complete, mechanical verification of large programs will be prohibitively expensive; however, inexpensive methods can certify a large percentage of the verification conditions generated during a development. By eliminating these "trivial" verification conditions, the total number is reduced so that those remaining can be more carefully considered by the development personnel. Using PK/C++, some modules of a system may be developed using formal methods, while others are developed using conventional techniques. This flexibility allows formal methods to be used only on the most critical portions of the system where the increased expense is justified.

In the rest of this paper we describe the implementation of PK/C++ in more detail. In section 2 we introduce the central features of our language. In section 3

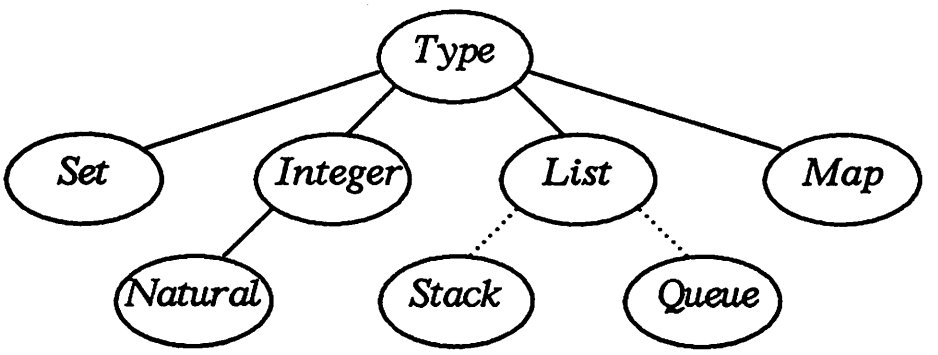


Figure 1: PK/C++ Data Types

we describe these features in more detail, and discuss how C++ helped or hindered us in our implementation efforts. We present an example in section 4, and give system status and draw conclusions in section 5.

2. Overview of PK/C++

PK/C++ has been under development since the Spring of 1988. The architecture is simple; no compiler or pre-processor for the language is used. The entire language is expressible using standard C++ syntax.

The system provides a number of predefined classes for data types which are linked in with all PK/C++ programs. These classes include definitions of high-level generic types such as *set*, *list*, and *map*. PK/C++ programs can be annotated with pre- and post-conditions, which can use either predefined predicates or newly written ones specific to the task at hand. The predicates operate by comparing the values of their operands and employing Prolog-style unification to provide a value to those operands which have not yet been assigned one. The Prolog features are incorporated using overloaded operators. PK/C++ also provides run-time type checking and error handling, to permit type-safe use of its data types.

In this section, we will give a high-level description of the essential features of our system, and in the next section we will present the implementation details of these features.

2.1. The Type Hierarchy

PK/C++ supplies the user with a predefined set of types which can be used to build executable specifications. This hierarchy is easily extensible; new object classes can be added as descendants of the predefined types, thereby inheriting the unification, type checking and error handling routines implemented there. We use the term "class" here in the sense of "datatype"; we will see in section 3 that this idea of a class directly corresponds to the *class* concept in C++.

Figure 1 shows the class hierarchy for the data types in PK/C++. The root of the tree is the class *Type*; all object classes used in PK/C++ specifications must be descendants of *Type*. Fundamental types immediately descended from Type are *integers*, *lists*, *sets*, and *maps*. The latter three types are provided since they are fundamental to VDM, and their inclusion permits the user to immediately proceed with the specification at a higher conceptual level than would be possible without providing equivalent abstractions. From these types, other more specialized types are then defined. *Natural* numbers are derived from integers, being a restriction of the integers to non-negative numbers. A *stack* is defined using a list, since it can be seen as a list with restricted operations; elements can only be added or removed in the first position. Likewise, a *queue* can be defined as a restricted list which can only have elements inserted at one end, and removed from the other. These types provide a good base from which to build specifications.

2.2. Pre- and Post-Conditions

A procedure or function can be formally defined by a pre-condition, which states the requirements the input data must meet before execution begins, and a post-condition, which states the relationship the input and output data must

satisfy after execution is complete. In a logic-based executable specification language such as PK/C++, when a routine (procedure or function) is first specified, it consists of only a pre- and post-condition, which syntactically resemble function calls. For example, consider the specification in Figure 2 of a function to increment by one an integer value on the top of a stack of integers. The pre-condition checks that the stack supplied is non-empty. The post-condition assumes the existence of the standard functions *top*, *pop*, and *push*, with the call to *push* returning a stack which is a copy of the original stack with its top entry incremented. Through the use of *unification* (described in the next sub-section), evaluation of the post-condition causes the output variables to be instantiated to values which make the conditions true; thereby permitting the executable specification to return a result. Later, when code has been produced to implement the function, the same post-condition verifies that the values produced by this code satisfy the requirements.

Figure 2a shows a "pure" PK/C++ function; in other words, one which maps inputs to outputs without any side effects. This function takes a stack S1 as input, and returns a stack S2 which is a copy of S1 with its top entry incremented. Note that variable S2 is apparently tested in the post-condition without ever having been set. But in PK/C++, the equality operator has been extended to additionally perform the unification function of Prolog. In this case, since S2 has no value when it is accessed in the test, the system attempts to find a value for it which will make the comparison true; that value is precisely the result of the expression to the right of the equality operator, which is assigned to S2. Had S2 already had a value,

```
void inc(S1, S2)        // input: S1; output: S2.
{
        pre(!(S1 == empty_stack));
        post(S2 == push (top(S1)+1, pop(S1)));
}
```

Figure 2a: A specification of a function to increment an integer on the top of a stack.

```
void inc(S)                    // increment S in place.
{
        Stack Si;
        pre((Si == S) && !(S == empty_stack));
        post(unset(S) == push (top(Si)+1, pop(Si)));
}
```

Figure 2b: An alternate form of the same specification, which modifies the stack in place. The call to *unset* returns its argument, now available for unification.

which would be the case if a code body existed for this function, then this test would verify that the value being returned does indeed satisfy the post-condition.

While returning a copy of a stack to prevent modification to the original has nice mathematical properties, it is hardly desirable in an implementation. But pure functions are not the only constructs we can specify in C++. In Figure 2b, we show a routine to modify a stack in place. The pre- and post-conditions are now more complicated, since in the pre-condition we must save the initial value of the stack in variable Si, so that in the post-condition we can *unset* S so that it can be unified with the final stack after the increment operation is performed. Although the form of the conditions still result in copying, a code body can be written which makes no copies, and once the routine is tested and the conditions removed, the routine will execute efficiently.

2.3. Prolog-style Unification

In Figure 2, we saw an example involving the use of the unification operator in a post-condition. We will now more precisely define what we mean by unification. *Unification* is a technique used in theorem proving to make two variables (or terms) equivalent. In Prolog, a simplified version of unification subsumes the functions of parameter passing and the assignment statement. In Prolog, the implementation of unification is complicated because "assignments" performed during unification may have to be undone during backtracking, this occurs when assignments need to be redone because the original value is incorrect. Backtracking can be eliminated, resulting in "flat" Prolog, which allows simpler implementation techniques to be used. PK/C++ is based on flat Prolog.

In PK/C++, a variable can play the role of either a *program variable* or a *logical variable* at any given point in time, and this role may change during the execution of the program. Each type of variable can be either *instantiated* or *uninstantiated*, depending upon whether or not it has been given a value.

A program variable must be set before it is used, or an error results. A logical variable may be used whether it is set or unset: if it is set, then just like a program variable its value can be referenced. If it is unset, and its first use is by a unification operator, then it is linked to the other operand of the unification. If the other operands have been instantiated (to a common shared value), then this operand becomes set to this value as well. If none of the operands are instantiated, then this operand remains uninstantiated as well; later, when any of the linked operands receives a value, all of the linked operands will be immediately instantiated to the same value. Once instantiated, the value of a logical variable does not change.

In an executable specification, the variables are treated as logical variables by the unification operators, and as program variables otherwise.

2.4. Run-time Type Checking

A run-time type checking mechanism for generic types is provided. This feature permits type-safe use of derived data types when passed as base types, eliminating the common problem where type safety is sacrificed in order to pass types generically. Each PK/C++ data type has a unique mnemonic tag associated

with it, this tag is tested to determine the type of a variable. Composite types such as lists and maps have a composite type tag structure associated with them, indicating the type of the data at each level in the composite type.

In the example in Figure 2 above, suppose our *inc* function had been passed a stack containing sets instead of integers. A run-time error would occur during the evaluation of the post-condition, when the system attempted to call the plus operator, which is not defined on an operand of type *set*. The error is caught by the system in the case that a call to an inappropriate function is made by the user.

2.5. Type Extensibility

The PK/C++ type system is easily extensible. The addition of a new type requires the introduction of a new type tag into the system, as well as the definition of the type itself and a set of routines to provide an implementation for a common interface to type-checking and unification. This new type can then be used with no further changes to either the existing type hierarchy or routines.

For example, suppose we extend the type system to incorporate the notion of a *symbol*, which is a character string name and integer value. First, we define a new mnemonic type tag name SYMBOL, to represent the new data type. Second, we define a new object class, derived from the base class Type, which contains data elements for a character string and an integer. Third, we provide code bodies for the virtual functions declared in the base class. Having performed these tasks, our new type is now known to the system, and can be used with other types such as lists and stacks, with no changes required to those types. In section 3.5, we show this extension in more detail.

3. Implementation of PK/C++

Many desirable features of our implementation are due to the object-oriented properties of the C++ language. Inheritance permits us to define new types as specializations of existing ones, automatically gaining access to already written routines, and to an interface for the customizable routines which we provide. The use of C++ virtual and/or overloaded functions and operators provide us with polymorphic routines.

In particular, the ability to implement Prolog-style unification by overloading the comparison operators of the language permits us to incorporate features of logic-based languages while remaining within the standard C++ syntax. We have accomplished our goals without extending the syntax of the language in any way or using a pre-processor.

In this section, we discuss the topics described in general terms in section 2, describe their implementation in C++, and show how the C++ language helped or hindered us where relevant.

3.1. The Type Hierarchy

The type hierarchy shown in Figure 1, and briefly discussed in section 2.1, is implemented as a C++ class hierarchy, with a base class for *Type*, and derived classes for each of the other data types. The connectivity between the types in the figure by solid line indicates direct derivation, and by dotted line dependence on

that type as a data member, with implied derivation from the same class as the class on which it depends. Figure 3 shows part of the definition of the data type *Type*; of particular interest are the declarations of the virtual functions, for it is through these functions that we achieve generality and extensibility of the type system. Note that the *unify* procedure is *not* declared as virtual, but is actually defined as a part of the base class. This definition hides the unification details in one place, permitting new types to be added without requiring any knowledge of how unification actually works, while automatically making the new type unifiable as well. This is possible since the unify routine uses the virtual comparison and assignment functions to perform much of its work. Each of the unification, comparison, and assignment routines is specialized for a particular purpose, with the result being a powerful set of cooperating routines.

The base class contains data fields consisting of a *unification pointer*, a *set bit*, and a *used bit*. The unification pointer accesses a circularly linked list of all variables in the equivalence class. Calls to the *unify* routine manipulate this pointer. The set bit is enabled by a call to the *set* routine, which is invoked whenever the

```
class type {
public:
            type ();          // Construct a "type"

  virtual boolean operator <= (type &); // compare/unify operands
  virtual boolean operator == (type &); // compare/unify operands

  virtual type   &operator = (type &); // assign by copy

  virtual C_type  *ctypeof ();      // Return composite type designator
  virtual S_type  stypeof ();       // Return simple type tag

      void   set ();          // Instance has a value.
      void   used ();         // Instance has been used.
      type   &unset ();       // Make instance undefined and
                              // available for unification.
      boolean is_set ();      // Does instance have a value?
      boolean is_used ();     // Has instance been accessed?
protected:
      void   unify (type &);  // Perform Prolog-style unification
                              // operation on operands. This
                              // routine uses == and = above.
      ...
      type   *t_unify;        // Logical variables to which this
                              // variable is linked.
      char   t_set, t_used;   // Set and used bits: 0=no, 1=yes.
};
```

Figure 3: The definition of class *Type*.

variable is assigned a value. The used bit is enabled by a call to the *used* routine, invoked when the variable is used. The *unset* routine is called to clear the set bit; this routine is invoked whenever we wish to make a variable available for unification again. By placing this data and these routines in the base class, all data types in our type hierarchy automatically inherit these properties, including those which will be defined later, as user needs require.

3.2. Pre- and Post-Conditions

In our implementation, pre- and post-conditions are simple functions which take boolean expressions as arguments. As C++ uses eager evaluation, the expressions are invoked (and return *true* or *false*) before the functions are called. *Pre* and *post* simply check that the expressions returned *true* and raise the appropriate exception otherwise. By default, this is a call to the *assert* library routine to terminate the process.

The routines which the user wants checked need to be defined with a call to *pre* at the beginning and a call to *post* at the end. The one drawback this approach has is that we cannot guarantee testing of the post-condition on every exit path from the routine. Care must be taken that all paths out of the routine pass through the post-condition, since there is no language enforcement of this requirement.

An alternate implementation we considered was to define a class to handle pre- and post-condition evaluation. In this scheme, a local class object is declared in the routine to be checked, with the pre- and post-condition expressions given in the initializer portion of the declaration. The pre-condition would be checked upon entry to the scope, by the class constructor. The post-condition would be checked upon block exit by the destructor (and could be guaranteed to be invoked). Unfortunately, this scheme has one major failing at present, which led us not to use it in this implementation: there is no guaranteed way to get access to the return value of the function to pass it to the post-condition evaluator. Certainly, a convention could be established using a variable of a certain name which could then be placed in both the post-condition and the return statement, but this will not guarantee that this value is actually returned in all cases from the routine, an unacceptable situation.

Language support of some kind for pre- and post-condition evaluation would be very helpful in this situation. One scheme which "solves" the problem would be to permit the overloading of the "return" statement. Then whenever a function exits with a value which we wish to check in a post-condition, we could get a handle on the value through our user-defined routine which could inspect (and perhaps change) the value being returned. Of course, one would have to be sure not use a return statement from within this routine, lest the program "return" in the wrong direction down the stack, deeper and deeper!

The observant reader will have noted that there are problems with this idea, namely that (1) "return" is not an operator, but a statement; (2) overloading works on the basis of argument type, so many overloaded return functions would be needed; and (3) there is no good way to limit the scope of the overloaded "return" functions to specific member functions within a class rather than to the

class itself. Since some solution to the problem of accessing the return value of the function is needed, some language support in this case would be helpful.

3.3. Prolog-style Unification

Prolog is the most widely used of the logic-programming languages. Taking Prolog and eliminating the backtracking results in "flat" Prolog, which allows simpler implementation techniques to be used. PK/C++ is based on flat Prolog. This permits us to use C++ to provide a reasonably elegant implementation of the notions of a logical variable and unification.

In PK/C++, program variables hold values and pre- and post-conditions state properties that must hold true for these values. At a particular point during execution of the program, the program variables hold particular values which must satisfy the conditions. PK/C++ logical variables, on the other hand, represent unknowns which are assumed to satisfy the conditions. Using this view we can see the "execution" of post-conditions as the search for values of the logical variables which satisfy the conditions. The only thing missing is a way to set program variables to reflect the results of this search. In PK/C++ we provide an explicit operation, *unset*, to transform a program variable into an uninstantiated logical variable so that it can be set in a post-condition.

Logical variables and unification are implemented in the *Type* class so that this code can be reused in all descendant classes. Logical variables are implemented using a *set* bit: if the bit is set then the object represents a program variable and has a value; if it is not set then the object is a logical variable and can be instantiated during unification. To implement unification, each object contains a *unify* field which is a pointer to an object of the same type. We can view unification as the construction of equivalence classes of variables; when two variables are unified their equivalence classes are merged. In PK/C++, the equivalence class of a logical variable is implemented as a circularly linked list using the *unify* field; when two variables are unified these lists are merged. We can view the unification of a constant to a variable as assigning the value to all variables in the equivalence class. In PK/C++, when a logical variable is unified with a value, the value is assigned to all the variables in the equivalence class using the virtual function defined for that data type, and their *set* bits are turned on. Thus they will be treated as constants from then on for the purposes of unification, until their *set* bits are turned off. The unification operation is represented by the overloaded equality operator (==).

```
enum S_type {
        TYPE, INTEGER, NATURAL, LIST, SET, MAP, STACK, QUEUE
};
```

Figure 4: The definition of the type tags used in PK/C++.

3.4. Run-time Type Checking

An *enum* type, shown in Figure 4, is provided which assigns a distinct enumerator value serving as a *type tag* to each class in the type hierarchy. Each derived class for a data type provides a body for a virtual function to return the appropriate type tag for its data type. Composite types use a composite type designator of type tags to represent their type. For example, the composite type designator for a List of Integers is itself a list structure containing the two tags "LIST" and "INTEGER", and the type designator for a List of Lists of Naturals is a list structure containing the three tags "LIST", "LIST" and "NATURAL". The type designators for objects are checked before operations are performed: to append two lists, their type designators must match.

3.5. Type Extensibility

An important feature of our system is type extensibility, since it will be more useful if a user can easily add specialized types necessary for his/her problem domain, gaining the ability to use these types with the predefined ones, and to perform type-checking and unification on them. To add a new data type, the user must first decide its placement in the type hierarchy, and then modify a supplied class template to define the class and the class from which it is derived. Implementations must be provided for the virtual functions declared in ancestor classes. Lastly, a modification must be made to the existing *enum* type to define a new type tag to represent this type. This new type can then be used with any of the preexisting class types.

Notice that we have declared the assignment operator (=) as virtual in the base class *Type*, taking an object of class *Type* as its second argument. Therefore, we can write a single assignment statement to assign any object derived from *Type* to any other object derived from *Type* without knowing how to perform the assignment. Each derived class supplies an implementation of "operator =", and decides what, if any, limitations it wishes to impose on the types it will accept.

Since the second argument is a base type, we require the routine to check its type at run time and generate an error if it is not appropriate. This permits our type hierarchy to be extensible: new types can be added without requiring any changes to any of the implementations of "operator =" already in place. Alternatively, we could have defined multiple versions of the virtual assignment operator in the base class, each taking a specific second derived type as an argument. In this case we would not need to make an explicit run time type check on the second argument, but we would sacrifice extensibility of the type hierarchy: any new type to be added would require a new virtual operator = to be added to the base class, and modifications to be made to all the classes which need to process this new type. In PK/C++, we elected to follow the former approach.

By taking this approach, we have strived to maintain type checking while permitting the use of "generic" routines. Our present approach uses additional data structures to provide structural type checking[1].

[1]While C++ uses name typing for classes, our use of structural typing allows us to have "generics" without needing to perform an instantiation for each new type.

For example, suppose we are now adding a stack type to the type hierarchy. We can model this relationship by adding a class *Stack* as a class derived from *Type* in the PK/C++ hierarchy, and using an instance of the *List* class as its data member. The *Stack* class uses the member functions and data structure of the list class to implement the new type, but *Stack* defines a new set of operations for its public interface and elects to not propagate the list operations which are not valid for stacks to this interface. The *Stack* class must also be assigned a type tag and provide function bodies for the virtual functions equality (==) and assignment (=); by doing so, the type-checking and unification code defined in the base class *Type* can be reused without modification. The code in the new equality operation simply calls *is_set* to determine if either operand is uninstantiated, and if so, then calls *unify* to perform the unification.

```
void sort_symbol(list &L1, list &L2)
{
        pre( true );
        post( insertion_sorted(L1, L2) );
}
```

Figure 5a: An executable specification to sort a list of symbols.

```
predicate insert_in_sorted_order(type &T, list &L1, list &L2)
{
        list L3(ctypeof(L1));
        return ((L1 == emptylist && (L2 == cons(T, emptylist)))
                                ||
                ((T <= hd(L1)) && (L2 == cons(T, L1)))
                                ||
                (insert_in_sorted_order(T, tl(L1), L3) &&
                (L2 == cons(hd(L1), L3)))
        );
}

predicate insertion_sorted(list &L1, list &L2)
{
        list L3(ctypeof(L1));
        return((L1 == emptylist && L2 == emptylist)
                                ||
                (insertion_sorted(tl(L1), L3) &&
                insert_in_sorted_order(hd(L1), L3, L2))
        );
}
```

Figure 5b: The predicates used by the specification.

4. An Example

To help clarify the features in PK/C++, in this section we present an example of an executable specification to sort a list of symbols, where a symbol is the new data type we added to the system. Figure 5a gives the specification. Routine *sort_symbol* takes a list of symbols and returns a sorted list.

The work is performed by the predicate *insertion_sorted*, shown in Figure 5b. It returns true if either both lists are empty, or the tail of the input list is sorted and the head of the sorted list appears in its correct position. Note the use of the equality operator in the body of the predicates; it is this operator that sets the uninstantiated variables to values. The reader will note that L2 and L3 are passed as parameters without ever having been assigned values; this is correct provided that the first access to these variables is through the unification operator.

The predicate *insert_in_sorted_order* takes a list L1 and an element T, and creates a new list L2 containing the element in its proper position. Either the

```
void sort_symbol(list &L1, list &L2)
{
        pre( true );

        list old_list(ctypeof(L1));
        list new_list(ctypeof(L1));
        list prev_new_list(ctypeof(L1));
        symbol t;

        L2 = emptylist;
        for(old_list = L1; !(old_list == emptylist); old_list = tl(old_list)) {
                t = hd(old_list);
                prev_new_list = emptylist;
                new_list = L2;
                while(!(new_list == emptylist) && !(t <= hd(new_list))) {
                        prev_new_list = new_list;
                        new_list = tl(new_list);
                }
                new_list = cons(hd(old_list), new_list);
                if(prev_new_list == emptylist)
                        L2 = new_list;
                else
                        prev_new_list . set_cdr(new_list);
        }
        post( insertion_sorted(L1, L2) );
}
```

Figure 6: The executable specification from Figure 5, coded using the *set_cdr* function to improve performance.

initial list is empty, in which case the new list L2 contains the single element T, or T is smaller than the first (and every) element of L1, in which case it can be placed at the head of the list, or T needs to be placed after the first element of L1, in its proper position, producing list L3, onto which the head of L1 is reattached. The reader is invited to work through a simple example or two, sorting a 2 or 3 element list, to get a better feel for this algorithm.

Although the predicates in Figure 5b produce the correct result, they do so in a fairly inefficient manner. Inserting an element into the middle of a list requires building an entire new beginning portion of the list, since we cannot have side-effects in our calls. To improve the performance, we would like to be able to use the *set_cdr* procedure to break a list in the middle and place the new element in its place without copying any of the other cells in the list. We can write a simple insertion sort to more efficiently produce the sorted list, as the body of our sort_symbol routine. Such an algorithm is presented in Figure 6. Note that now we are assigning L2 a value before the post-condition is encountered; when the condition is checked, the predicates will now manipulate both L1 and L2 to try to determine whether L2 is indeed a sorted version of L1 or not. Again we invite the reader to work through these predicates using short lists for both L1 and L2, and to try both the case where L2 is sorted and is not sorted, to see the behavior of the post-condition.

It turns out to be the case that the program in Figure 6 actually runs more slowly than the one in Figure 5, because in the program with the insertion sort implemented we are actually sorting the list twice, once in the body of the routine, and once in the post-condition. But the time taken in the body of the routine is much less than the time taken by the post-condition, so that after running this routine through a representative set of test cases to verify its operation, we can compile out the pre- and post-conditions and produce a program with reasonable efficiency while having greater confidence that it correctly performs its function.

5. System Status and Conclusions

We have completed a prototype implementation of PK/C++; it is written in C++ version 1.2 and runs under both System V UNIX and Berkeley UNIX, on both AT&T 3b2s and Sun workstations. Although we do not believe the current implementation demonstrates the full potential of the language, we are pleased with its performance. The execution data we have gathered shows that PK/C++ specifications run about two orders of magnitude faster than the previous PLEASE implementation, but are still two orders of magnitude slower than C++ programs which perform the same functions. We feel we can improve their performance with further research. We believe our choice of features for PK/C++ was wise. Since PK/C++ is based on C++ the power of object-oriented programming is available. C++ classes and inheritance allow us to implement generics while retaining type checking and permitting extensibility; however, we are not sure our use of run-time structural typing is the final solution to the problem. Basing our execution strategy on flat (unification but no backtracking) rather than standard Prolog allowed much simpler and somewhat more efficient implementation techniques to be used, although in some sense it decreased the logical power of the language. We believe that the use of methods similar to those based on PK/C++ specifications will enhance the design, development, validation and verification of software.

References

1. Campbell, R. H. and R. B. Terwilliger, "The SAGA Approach to Automated Project Management", in International Workshop on Advanced Programming Environments, Carter, L. R. (editor), Springer-Verlag Lecture Notes in Computer Science, New York, 1986, 145-159.

2. Clocksin, W. F. and C. S. Mellish, Programming in Prolog, Springer-Verlag, New York, 1981.

3. Fagan, M. E., "Advances in Software Inspections", IEEE Transactions on Software Engineering SE-12, 7 (July 1986), 744-751.

4. Fairley, R., Software Engineering Concepts, McGraw-Hill, New York, 1985.

5. Jones, C. B., Software Development: A Rigorous Approach, Prentice-Hall International, Engelwood Cliffs, N.J., 1980.

6. Kirslis, P. A., R. B. Terwilliger and R. H. Campbell, "The SAGA Approach to Large Program Development in an Integrated Modular Environment", Proceedings of the GTE Workshop on Software Engineering Environments for Programming-in-the- Large, June 1985, 44-53.

7. Loeckx, J. and K. Sieber, The Foundations of Program Verification, John Wiley & Sons, New York, 1984.

8. Meyers, G. J., The Art of Software Testing, John Wiley & Sons, New York, 1979.

9. Terwilliger, R. B. and R. H. Campbell, "ENCOMPASS: an Environment for the Incremental Development of Software", Report No. UIUCDCS-R-86-1296, Dept. of Computer Science, University of Illinois at Urbana-Champaign (also to appear in the Journal of Systems and Software), September 1986.

10. Terwilliger, R. B. and R. H. Campbell, "PLEASE: Executable Specifications for Incremental Software Development", Report No. UIUCDCS-R-86-1295, Dept. of Computer Science, University of Illinois at Urbana-Champaign (also to appear in the Journal of Systems and Software), September 1986.

11. Terwilliger, R. B., "ENCOMPASS: an Environment for Incremental Software Development using Executable, Logic- Based Specifications", Report No. UIUCDCS-R-87-1356 (Ph.D. Dissertation), Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1987.

12. Terwilliger, R. B. and R. H. Campbell, "PLEASE: a Language for Incremental Software Development", Proceedings of the 4th International Workshop on Software Specification and Design, April 1987, 249-256.

13. Terwilliger, R. B. and P. A. Kirslis, "PK/C++: an Object- Oriented, Logic-Based, Executable Specification Language", Technical Report CU-CS-400-88, Department of Computer Science, University of Colorado at Boulder, June 1988.

14. Terwilliger, R. B. and R. H. Campbell, "An Early Report on ENCOMPASS", Proceedings of the 10th International Conference on Software Engineering, April 1988, 344-354.

15. "Special Issue on Rapid Prototyping: Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop", Software Engineering Notes 7, 5 (December 1982).

DEBUGGING AND INSTRUMENTATION OF C++ PROGRAMS

Martin J. O'Riordan
Glockenspiel Ltd.
19, Belvedere Place
Dublin 1, Ireland

Phone: 353 - 1 - 364515
FAX: 353 - 1 - 365238

martin@puschi.uucp
..!mcvax!iclitc!puschi!martin

ABSTRACT

Debugging of C++ programs is both conceptually and technically more difficult than traditional 'C' programs. We must learn how to make good use of our traditional tools, and understand better the nature of our debugging problems. However, the very language that presents these new problems, also presents us with the tools to build new and better solutions to our debugging needs.

Introduction

Debugging programs in any language is not easy, but at least in the traditional programming languages including 'C' we are aware of the programming goals and the probable areas of trouble we may incur. C programs are primarily functional in nature, by which I mean, they are composed of units of code called "functions". These functions collectively describe the solution to a given problem. Data where it exists is relatively uncomplicated, and has probably been derived as a bi-product of the functions which use them. Consequently, the primary approach to debugging C programs has been to trace the execution of the code or functions sometimes using very sophisticated debugging tools. These tools also allow us to examine the state of the data used, in terms of their hex values, or sometimes in more type specific formats. But the main broadside has been aimed at the code, and the way in which code is likely to go wrong.

With C++, and indeed other Object-Oriented programming languages, the subject which we must debug is not so likely to that of the code. In fact the OOPLs have tended to make the code more likely to be correct, or at least a correct representation of the problem described by the programmer. Instead it has brought into existence, the subject of design and data debugging, something which has generally been ignored by the debugger designers. We now have the problem being described in terms of objects and how these objects interface with other objects. This is much more akin to the problem of debugging network architectures or other forms of communications related designs.

2. TRADITIONAL METHODS AS APPLIED TO C++

Currently the major implementations of the C++ programming language are in the form of a C++ to C translator, which makes optimum use of the fact that C is already well established and already has excellent optimising compilers, libraries, third party tools, and indeed debugging tools. To introduce a new language as a compiler restricts it's propagation through the programming community, and is also unlikely to be as good as the well established C compilers with which in this case C++ would have to compete. As the translation is to C, we may firstly apply our C debugging techniques to C++.

2.1 Instrumentation

The first method of simple *Instrumentation*, is the insertion of code into the program, to report status or some other programmer defined statistic. Typically, this in the form of a simple 'printf', a method of debugging not to laughed at, as it can convey knowledge about a specific program, that no general purpose debugger can presently do. However, it does mean editing the original, and potentially while doing so there is the problem of breaking something else. Also, it is very easy to leave some of this instrumentation in the production version of a large piece of software, which may easily escape notice until a customer sends you an enquiry.

The second most widely used form of instrumentation is the profiler which is combined into the executable by the compiler and gathers certain usage statistics about the runtime behaviour of a program generally in conjunction with some specially designed input data, to exercise specific parts of the program.

2.2 Debuggers

Ranging from the very simple assembly level non-symbolic debuggers, right through to the more sophisticated source level debuggers which allow interaction with the program though the symbols and expressions of the target language. These latter debuggers allow us to examine data according to the type it has been declared as, to enter expressions as they may appear in C, and to put breakpoints not just at individual instructions in the assembly code, but instead at lines as they appear in the original source code.

But these debuggers are inadequate when it comes to C++. The obvious problems are :-

Name-Mangling, which is a side effect of C++'s extended understanding of scope and the introduction of overloaded function names. This makes it difficult for us to recognise the correlation between the original C++ name, and the generated C name, which the debugger sees.

Inline-Functions, expand to complicated expressions which bear no resemblance to the simple call they appear to be in the source code. And to compound matters, they cannot be breakpointed, as they do not exist as real functions.

Virtual-Functions, may be called purely by an index into a table indicated by the structure to which they belong. This means that we don't know until we actually invoke the function, which function it is. This is made even more difficult when you consider the passing of pointers to virtual functions, for which the information is just a simple number. Breakpointing virtual functions is extremely difficult, as it involves searching for all such functions in a potentially large hierarchy, and setting a breakpoint on each.

Inheritance, introduces the problem of an object, especially pointers, not being what they appear to be. Even the very clever C debuggers which use type related formatting for presenting an objects state, cannot work correctly here, as each class in a type hierarchy appear as discrete types to the C debugger, indeed some of the newer C compilers object to derived class pointers being passed to a function which declares a base class pointer as being expected, although C++ is clearly generating correct code according to it's model.

Header Files, add another new complication. It is generally considered bad practice to include code in a header file when programming in C, and indeed many source level C debuggers cannot track into a header file as they load only the file described by the last "#line" directive in the program. But where else can you put an inline function for a third party object? For good correlation between the C++ source, and the code being debugged, the current

translators emit #line directives in and out of header files, as code described in them is executed. This new aspect causes many debuggers to fail, and others to merely track apparently random lines in the wrong source file.

But despite these difficulties, most of us will have become familiar with these aspects, and will have compensated for them in our everyday debugging needs.

3. MAKING THE OLD WAYS EASIER TO USE

For those of us with access to the original AT&T translator sources there are many things we can do without great difficulty. Indeed we at Glockenspiel have done much work in this area to improve the lot of debugging, by passing on these changes to our customers.

3.1 Name Mangling

Changes in the name mangling algorithms is one simple way of improving debugging. Data members of a class do not always need to have the class name of which they are members prepended, as no conflict can exist in classes not involved in inheritance, and name conflicts are surprisingly rare even in classes involved in object type inheritance hierarchies. For this reason the class name may always be removed from the names of data members in the base class, or in classes not involved in inheritance, and indeed one can go farther, by omitting the class name even from members in a derived class when no name conflict exists. This can be determined simply by scanning for a member of the same name in the base, and in the base's base ... and if no such name exists, there is no conflict. If a conflict exists, then the class name may be prepended. This facility can be made optional, so that the default naming is always available.

The following sample illustrates this name reduction scheme in operation :-

```
// Without name reduction

// Original C++ Code          |   // Generated C Code
class Base {                  |   struct  Base    {
        int     i;            |           int     _Base_i;
        char    *name         |           char    *_Base_name;
};                            |
```

```
class Derived : Base                |   struct  Derived {
        Base    *pBase;             |           int     _Base_i;
        long    i;                  |           char    *_Base_name;
};                                  |           Base    *_Derived_pBase;
                                    |           long    _Derived_i;
                                    |           };
```

```
// With name reduction

// Original C++ Code                |   // Generated C Code
class Base {                        |   struct  Base    {
        int     i;                  |           int     i;
        char    *name               |           char    *name;
};                                  |   };
                                    |
class Derived : Base                |   struct  Derived {
        Base    *pBase;             |           int     i;
        long    i;                  |           char    *name;
};                                  |           Base    *pBase;
                                    |           long    _Derived_i;
                                    |   };
```

Clearly, the generated names bear a much stronger resemblance to the original with the name reduction scheme on, than they do without.

Similar approaches may be used to reduce the prefixing of the 'auto' lexical level on function arguments and internal variables, by checking for name/scope conflicts, which in most well designed programs don't exist. It is generally confusing to use the same name for different variables in the same function at different lexical levels, as the fact may easily go unobserved, generating error messages due to incorrect use at best, and unexplainable behaviour at runtime, which is more likely. C++ needs to have this mechanism of lexical prefixes, as it allows for greater scope flexibility than merely lexical, and the two times they are most needed are for allowing access to a global variable of the same name, and for invoking the destructors for dynamic objects to be correctly invoked. But if no name/scope conflict exists the prefix may be removed, thus local variables and function arguments may possess the same names as they had originally.

The 'this' variable is a curious anomaly in C++. Only one 'this' variable may exist in a function, and it is always at the outermost lexical level. Further more, it may only exist in a function, thus a global instance of 'this' is not possible. The program may not simply declare one when desired, as it is an implied variable. For this reason there is never a need for the auto-prefix, and it may be unconditionally dropped, further simplifying the interface to the debugger.

3.2 Inline Functions

Debugging code containing inline functions can be extremely difficult, as the general control flow appears to have been altered depending upon the complexity of the inline function. When the inline functions interfere with debugging, then it is good idea to switch them off. This allows you to view a call as a call, and not as an expanded, and often complicated expression bearing little resemblance to the original code. At this level, efficiency is not an issue as the sort of program being debugged using traditional debuggers is probably not time critical anyway. For time critical code, the issues of debugging remain greatly unchanged since C and all the old favourite techniques are probably still best.

Suppressing inlines has a number of benefits. Not only does it allow the general control flow of a function appear mostly unchanged in the emitted C and thus object code, but it also enables breakpointing to be used where previously it was not possible. It also means that we have a real symbol by which we may refer to an inline function, instead of the "Trained and Familiarised Optical Pattern Recognition" schema on which we otherwise have to depend upon, in order to identify an inline function expansion.

But this has side effects too. Suppression of inlines causes a static instance of the function to be created, thus different instances, although functionally identical, exist in each module processed in this way. Thus breakpointing has other problems similar to that of virtual functions, in that all instances must be searched out, and individually marked, an unpleasant exercise to have to undergo.

3.3 Header Files

When a debugger is unable to cope with code in header files, there are three options available. Firstly, we may ignore all #line information and *Source Level Debug* the generated C code, not to be recommended. The other two options involve adapting the translator to emit customised #line directives. These customisations really fall into only two forms. When code is referenced in an included file, simply *Freeze* the current position by emitting #line's at the last position in the primary source file, until control returns to the primary. The other method is similar, but in order to indicate that the code being executed is actually in the included file, the #line's emitted may indicate the #include line for the included file, until control returns to the primary. Thus the source line display will stay on the #include directive until normal control is restored. Obviously these methods are not good if the bug being sought actually exists in the included code, but this measure is only suggested for that situation where the target debugger gets confused by the multi-file method used in C++.

The methods outlined above are simple effective methods for using the existing tools, but with reduced pain. They are obviously not wholly adequate methods for debugging, especially an OOPL, but in the absence of more powerful debugging tools and paradigms, they contribute greatly to the overall usability of C debugging tools with C++.

4. MAKING C++ EARN IT'S LIVING

Some aspects of debugging are best addressed by using the language against itself when debugging "To catch a thief..." and indeed it does provide us with some interesting tools with which to do this. One of the problems that we often want to solve in a program is that of program flow from function to function, and more importantly, in programs using either direct or indirect recursion. For this a method for determining when a function gets control is required, easily provided with one statement at the entry to a function. Less easily provided is a method of determining when it returns control to the caller. If the function has one exit point a second simple statement provides the solution, but when there are many exit points, placing a statement at each is time consuming, and prone to error and incompleteness.

C++ provides us with a very simple, yet powerful way of tackling precisely this type of problem. A class which contains nothing but a constructor and a destructor may be used to solve this :-

```
class Trace {
public:
        Trace ( char* Function_Name );
        ~Trace ();
};
```

When a declaration of an instance of the class "Trace" is inserted into the function, before any other statements, it solves both the problem of entry and the problem of exit with only one statement. It is a very simple trick, easily sneered at, but it provides us with the basis for very powerful tools based on this and other simple mechanisms already provided by the language.

The declaration is inserted as the first statement of the function, with the name of the function as an argument. This causes the *Constructor* to be invoked, informing the runtime support mechanism for the Trace facility, that a function has been entered, and which function it is by name. This is completely automatic. Also completely automatic is the invocation of the corresponding *Destructor*. Whenever an object goes out of scope, it's destructor is automatically invoked, and the Trace object can only go out of scope

when the function reaches the closing brace, or when a return statement is encountered. In either case, the language guarantees the invocation of the destructor. Thus, with the addition of only one line of C++ code, we may monitor the entry/exit behaviour of a particular function, or indeed that of many functions. This is similar to the way the profiler works in present C compilers, but more powerful, in that the simple hook planted in the code has user defined functionality, and may thus provide more specific data about the program under test.

5. INSTRUMENTATION

The previous section introduced a very simple example of C++ *Leger de main.* The language has many simple tricks which we may use to monitor specific aspects of a programs behaviour. These tricks generally involve the insertion of code into the source file. This process, like printf'ing in C is very powerful, but like it's predecessor prone to accidental damage while editing, or left in a vestigial state when the product is finally sent to the customer, a bit like finding a screw-driver in the cabinet of your new computer. Still, instrumenting a program is often the best way of debugging, and never was this more true of a language than of C++. C++ is an OOPL, that is, it is intended for the design of abstract objects. It takes care of a lot of the problems we had in C such as type checking, initialisation of data, destruction of data in controlled and programmed ways. The error messages and warnings are rich in the information they provide, often indicating doubtful code or data design. If the warnings are heeded that is. I would always advise programmers to heed the warnings, as they are a lot easier to understand than the unexplainable behaviour of a runaway program, or the intermittent and often unrepeatable behaviour of a dangling pointer problem.

With the extra support, and the general design methodology of C++ programs, the emphasis for debugging has moved from code level debugging, to *Object Level Debugging*. That is, we must now debug the object in terms of it's behaviour, and of it's intended purpose versus it's actual behaviour.

For C++ Instrumentation seems to be the best approach, certainly combined with conventional C tools, perhaps enhanced to understand C++ syntax. But when it comes to the debugging of the design, only custom made tools can truelly provide the answer. For this reason the automatic insertion of Instrumentation code is by far the best approach. Automatic insertion of instrumentation means you don't have to worry about accidental damage caused during an edit while inserting new instrumentation. A spanner need not be left in the works, as a simple recompile of all the components without instrumentation may be done before production, or indeed, and as a safety precaution, the existing object code could be relinked with a dummy version of the instrumentation support library. As

the instrumentation provides only hooks into the code, this is very easily achieved. The same mechanism allows the test and/or design personnel to customise the operations they require of the instrumentation by providing their own implementation of the instrument hooks, or they can use the default ones provided. Similarly, a dummy version which does nothing may be provided in order to catch those *Spanners* safely and tidily.

6. HI-JACKING THE VIRTUAL TABLE

No not an act of terrorism, not even extending the language. By Hi-Jacking the Virtual Table, I mean *Extending it's Original Purpose*. The virtual table is after all an implementation feature. It's existence is not even implied by the language, and certainly it's construction is not. For this reason the nature of the virtual table may be altered or extended in order to facilitate the needs of the debugging community.

The virtual table contains a list of the addresses of the virtual member functions, for a given instance of a class within a type hierarchy. It allows us to dynamically bind the behaviour of an object to the actual instance at runtime rather than statically bound at compile time. However, how this is implemented is irrelevant to the function described in the language. Using an array is one method, although very efficient, it is not as flexible as say a linked list, nor as powerful in a network distributed system, as a tree topology. What I am saying is that the language itself is not altered by using alternative implementations of the virtual table.

By extending it's functionality, I am suggesting that more information than purely the virtual function addresses could be stored in the table. Essentially, it provides the only dynamic means of identifying an object. By moving all the addresses up one index in the array implementation, the zeroeth element becomes free, and at no loss of efficiency. This zeroeth element may then be used to refer to a structure which can provide the debugger with dynamic information and type identification of the object under examination. This structure could contain the following information :-

> The actual size of the object. Indeed this could be a useful facility in the C++ language, as it would provide a virtual 'sizeof' facility at small expense, and in a non-user definable way.
>
> The name of the object.
>
> A reference the the structure describing the base class of the object if any.
>
> A list of the objects members, their types, and any other useful information.

etc. In fact, the number of potential uses exceeds the needs of the debugging aspects. One such use is that of Persistence, where the lifetime of an object may outlive the invocation lifetime of the program that created it. Further extensions on the idea of persistence are inter-processor communications, a problem that becomes especially difficult when one considers the problems of communicating objects which are part of an inheritance hierarchy, and which have an associated functionality as well as simply datum. And many more.

7. POD -- A Portable Object-oriented Debugger

It is from these and other simple alterations to the *Behind The Scenes* C++ that the design and implementation of POD has been derived. POD has been designed by Glockenspiel and will be available with our next major product revision as part of the C++ Software Development Kit we are forging. Many of the simpler name mangling facilities are already present as simple switchable options, and have been so for quite some time. POD was initially designed two years ago, and implementations of it have been in use within Glockenspiel since April 87. Initially the design could have been called simply **'OD** or **Object-oriented Debugger**, as Portable had not yet made it's appearance.

ith POD enabled, the translator emits Hooks into the generated C code, which provide the programmer with a simple mechanism onto which they may build a special purpose debugging tool. POD comes with a default debugging tool in the library "libPOD.a", which is not intended to be the ultimate debugger, but does provide many of the often used, or often required debugging facilities.

What is more important, is the provision of a the POD header file, from which the programmer may derive their own specific debugger, replacing POD's default implementation with programmer defined versions. The number and nature of hooks embedded by POD are determined at compile time by the programmer.

Many of PODs member functions are virtual allowing different versions of the debugger to be used in different places, thus a context sensitive debugger is possible, a concept not realisable with good old C. This means that a library of different debuggers could be built up over the lifetime of a few projects, and then they would start to become reusable. The debugger most suited to the task could be determined at run-time depending upon the exact nature and context of the program being debugged. Indeed, with the increasing use of dynamically linked libraries being used, the exact selection of the debugger could be deferred until runtime, so complete independence of the precise debugger used is achieved, allowing the debugger to be tailored exactly to the needs of the problem.

7.1 The Hooks POD Provides

POD provides many facilities to the debugger. Hooks may be emitted for a large variety of purposes, but how many hooks, and which hooks are emitted are under the programmers control, as all are enabled using special switches. The types of hooks fall into a few simple categories, which are best discussed separately. The first set of hooks deal will the control flow characteristics of a program, and deal with such aspects as source position, lexical levels, scope and such like.

7.1.1 Positional Hooks

While debugging, it is necessary to know where in the corresponding source code your program is executing. For this purpose a Line/File positional hook is emitted. This is a message to the debugger, which informs it that the current position has changed. The positional information has several forms. Absolute Line/File messages informs the debugger that a complete source file has changed, and provides the name and line number in the new file. This is used when the code being executed moves from one module to another. Relative messages inform the debugger that the source line being executed has changed, but not the file, and secondly, it may inform the debugger that a relative source file has changed. This means that the source file in which it is now executing, actually occurs in an included file, which forms a part of the one object module. This allows for a file stack frame to be maintained be a debugger, enabling rapid context return to the including source file.

The positional Line/File messages in POD are :-

```
// Change the Absolute Source File and Line
void __POD_AbsFile ( char*& FileName, int LineNo );

// Change the Relative Source File and Line
void __POD_RelFile ( char*& FileName, int LineNo );

// Return to the including file, at line #
void __POD_RelReturn ( int LineNo );

// Change the Relative Source Line
void __POD_RelLine ( int LineNo );
```

7.1.2 Function and Lexical Hooks

When a running program enters or exits a function or a new lexical level, POD can send a message to the debugger, informing it of the change. For this type of operation, the simple positional information is inadequate, as the debugger would have to parse the code in the source to determine what action to take, and even then it cannot determine which lexical level the program may be at. At the beginning of each new lexical level, POD can insert a

simple Instrumentation class similar to the "Trace" class mentioned previously. The outermost lexical level is a special case, in that it is the function entry point, and differs only in the way in which it is constructed. This simple mechanism informs the debugger immediately of a new lexical level, or entry to a new function. The destructor for this object informs the debugger that the control flow is leaving this lexical block. Thus the simple function in/out mechanism is easily extended to the more general lexical in/out model.

The Function and Lexical messages in POD are :-

```
// New Function Entry
__POD_Block::__POD_Block (    char*& FunctionName,
                              void* FunctionAddr );

// New Lexical Level Entry
__POD_Block::__POD_Block ( int LexicalLevel );

// Global Entry, one per module
__POD_Block::__POD_Block ();

// Exit Function or Lexical Level
__POD_Block::~__POD_Block ();
```

7.1.3 Scope and Variables

Often the debugger needs to know about the variables in a function, and also the variables available to a function from global data space. For this reason POD can send messages to the debugger each time a new object is declared. This method implies the lexical instrumentation, as the messages are to the __POD_Block object. These messages inform POD about the name, address and type of the new data object being declared, allowing the debugger to examine and monitor specific data in the program. Conveying the type of the object is the most complicated part. If the type is simple, then a simple coded message is sent. If it is a composite type is involved, then a compressed composite code is generated and sent to the debugger. However, if the object is an aggregate, then a pointer to the corresponding type descriptor is sent. This type descriptor is also indicated by the zeroeth element of the virtual table. All classes will have a type descriptor record generated by the Translator, whether they have virtual functions or not. This means that they may all be dynamically typed by POD and it's associated debugging facilities. For global variables, the situation is more complicated, but again the language can help us. C++ guarantees initialisation of user defined objects using their associated constructors. This applies to global data also. POD will generate a static constructor containing messages to the debugger informing it of the global data declarations.

The Scope and Variable messages in POD are :-

```
// Declare a compound variable in this scope
__POD_Block::AddVar ( char*&        VarName,
                      void*         VarAddr,
                      __POD_Type*   VarType );

// Declare a composite variable in this scope
__POD_Block::AddVar ( char*&        VarName,
                      void*         VarAddr,
                      char*&        VarType );

// Declare a simple variable in this scope
__POD_Block::AddVar ( char*&        VarName,
                      void*         VarAddr,
                      int           VarType );
```

Note that the destructor must also clean up all variable entries entered into it's scope. Thus the whole scoping rules may be automatically controlled for lexically allocated variables. Formal arguments are declared before any other statements are encountered, thus POD must emit references to these before anything else other than the __POD_Block object for function entry. Thus the formal arguments and first lexical local variables appear in the same space. Thus POD must cater for variables being hidden in this way.

Global variables are entered by a static constructor into the scope of a global __POD_Block object which operates in a similar fashion to the lexically bound __POD_Block objects, and removed by the corresponding static destructor.

The above hooks provide for most of the needs of control flow debugging, as well as allowing for actual user definable typing. But even these do not address all the needs of Object-Oriented debugging. Consider the problem of breakpointing virtual functions, and indeed inline functions. This is one area which is very difficult to address to address using conventional debugging tools. Also, the ideas involved really require that the debugger should be able to communicate with the objects themselves. The following section deals with these more advanced types of debugging needs.

7.1.4 Breakpoints in Virtual and Inline Functions

With C a function is a function, and setting a breakpoint on a function is simply a matter of finding it's symbolic name, and placing a breakpoint at the address indicated. When execution causes this function to be invoked, the corresponding breakpoint causes execution to stop, and the debugger changes it's mode of operation. With C++, setting a breakpoint on a virtual function is not only conceptually more difficult, but technically more difficult also. POD simplifies this in two ways. Firstly, it can optionally emit a trap function for a specified virtual function in a type hierarchy. As all virtual functions of this

name and type go through the trap function, it is a simple matter to place a breakpoint only on this function, causing execution to freeze there. Stepping over the function return sequence gets into the real function. This allows POD to interact with conventional debuggers, and ease there task when breakpointing virtual functions. Inline functions are addressed in a similar fashion. These stub functions have the same name as the virtual function to which they correspond, but prefixed by "__POD_".

A second method exists, which is more powerful and flexible as it operates purely within POD. Already available is the Function/Lexical entry instrumentation, which knows when and which function is being entered or exited. The runtime support for these functions can internally breakpoint on a category of functions as breakpoints on all or just selected branches of virtual functions is greatly simplified. Thus it is possible to be more selective about which functions are to be trapped and which are not, possibly examining only one or two branches of a large hierarchy. An analogous method applies to inline functions.

7.1.5 Debugger - Object Communication

Sometimes it is necessary for the debugger to examine the state of an object. For this reason, POD will use the type information it finds in the type descriptor stored off the zeroeth element of the virtual table. It will display are present the information according to a default algorithm for dumping or analysing aggregate types. However, sometimes this is not always appropriate to a particular user defined type.

For this reason, POD will take a closer look at the type and if it has a member function called "__POD_In", which takes a reference to a structure called "__POD_CommType", then instead of performing the default type analyses algorithm, it will call this function for which the programmer must provide a definition. The argument exchanged is the way in which the debugger may communicate with the object. This is a two way mechanism in that the debugger may call the "__POD_In" in order to make a request of the object, but the object may also send messages to POD, using the "__POD_Out" function, which has the same argument and invocation syntax, but which the program may call. The __POD_Out function may only be invoked as a member function, and this function is not provided by the programmer. Instead, the translator will generate the function automatically for POD when enabled.

7.1.6 Other Messages

Finally, POD has a couple of other messages available which the programmer may insert in their code. These are simple, globally available functions which are declared in the associated POD header file "pod.hxx", which describes all the classes and declarations necessary to avail of PODs facilities in the programmers code. This is automatically included by the Pre-Processor when POD is selected. The four main functions available are as follows :-

```
// Force POD off, disable POD for now
void __POD_Off ();

// Force POD on, enables POD after __POD_Off has been used.
void __POD_On ();

// Force POD to restart it's statistics
void __POD_ReStart ();

// Force POD to display it's current statistics
void __POD_Display ();
```

These functions are useful for gaining more selective control over POD's monitoring functions. As POD comes with a dummy library, these may be left in production code without any side effects other than a slight decrease in performance, and a slight increase in size. But not providing a two way method of communication between POD and the program under the microscope is not worth the extra debugging value lost by their exclusion.

7.2 POD Conclusion

As stated earlier, many of POD's functions are virtual. This means that the programmer using POD can design their own debugging facility on the hooks provided. Glockenspiel provide two debugger libraries with POD. The first is a simple general purpose debugger built on these hooks. We may place the implementation of this debugger in the public domain at a later date, but for now it is provided only in object form as part of the POD product.

Virtual hooks enable the debugger to change it's spots mid run if required, adapting to the different needs of different parts of the program. This can be done dynamically as the program executes, or selectively through the debugger's user interface by some interactive means.

7.2.1 Other uses for POD

Uses for POD are not only for debugging and for data monitoring, but it may also be used for various types of stress testing, and validation tools for complex pieces of software. Another incidental spinoff, is for controlling demonstration software. The software may be constructed with POD embedded, and instead of a debugger piggy-backed, it could have some form of demonstration driver instead.

POD may be used for profiling also. Gathering the sort of statistics the standard C compiler collects is relatively simple, but performing such services as topological profiling is a newer and more complex type of profiling. With topological profiling, we are not only interested in the percentage of time a given function is called, or how many times it is called. But also in the way execution progresses. Topological profiling reveals the inter-dependency between functions which regular profiling cannot. This allows us to optimise locality problems, especially on distributed systems, or certain unmentionable small computers using a simple segmented architecture. Furthermore, profiling can be extended to a per line or per statement analyses within a single function, to determine precisely where the resource hog is, in order to optimise correctly.

8. CONCLUSION

The Portable Object-Oriented Debugger "POD", presents a fairly complete set of hooks, most of which are discussed above, but some also lesser ones which I have omitted. However, it also can generate copious numbers of hooks. For many debugging tasks, the full POD support is not required, so POD has a set of switches the programmer may use to enable or restrict many of POD's hooks. These include restricting how many lexical levels are to be monitored in the module, or whether positional information is emitted every line, every statement, etc. The communication facilities may be enabled separately. In fact there is a wide range of switchable facilities for POD support. For this reason, although switches are actually used, a switch composition mechanism is provided through which the novice POD user may construct the switches interactively. This means that POD is available not only to the experienced programmer with, intimate knowledge of the complex switches used, but also to the new-comer who wants to avail of it immediately, without a large learning curve.

libg++, The GNU C++ Library

Douglas Lea

State University of New York, College at Oswego
Oswego, NY 13126
(dl@rocky.oswego.edu)

ABSTRACT

The GNU C++ library is a collection of C++ classes and support tools. The paper describes several general strategies for structuring and designing GNU C++ library classes, along with an informal taxonomy of library classes and their implementations.

1. Introduction

The GNU C++ library, *libg++*, is a collection of C++ classes and support tools designed to be used with the GNU C++ compiler, *g++*. It is being produced on a volunteer basis with the assistance and support of the Free Software Foundation GNU project. However, as of this writing, neither *libg++* nor *g++* are official GNU products. All distributions thus far have been labeled as test releases.

While the GNU C++ library accommodates and/or exploits minor differences, extensions, and limitations of GNU C++ versus the AT&T version, it is by-and-large designed as a general C++ programming support system in its own right.

This paper describes several basic design and implementation decisions and strategies made in the GNU C++ library that may be of interest to others creating and using C++ libraries and support tools.

2. Architecture of a class library

An especially attractive virtue of the C++ language is that it directly supports many different programming styles and methodologies, any of which may or may not be considered as "truly" object-oriented by those who like to argue about such matters. This fact impacts a library designer in countless ways, but most importantly in the ways that inheritance is employed in structuring the library.

One approach, taken by Keith Gorlen's *OOPS* library [1], is to exploit inheritance in order to create a single rooted tree of classes that together constitute a consistent programming environment in which all classes are somehow derived from the root class, *Object*. This is also the route taken in libraries for several other object-oriented languages, especially Smalltalk.

A less ambitious-looking approach is taken in the GNU C++ library, which consists of a collection of generally independent classes available for individual selection and use. This "forest" of classes employs inheritance only when needed to express inherent class dependencies.

There are many good reasons for choosing either approach.

2.1. Consistency versus flexibility

Because the "tree" approach creates a consistent programming system in which many decisions about class structure, intercommunication, usage and inheritance conventions, etc., have already been made, such a system is often easier to use than the "forest" approach, but can also be less flexible. A forest structure is often more amenable to the integration of new classes written by different programmers for different reasons, as is the case with the GNU C++ library.

A forest builder cannot fully enforce some conventions that are worth enforcing in many applications. An important example of this is the issue of error handling. Since the C++ language does not yet contain an exception handling mechanism, any library must incorporate some kind of mechanism itself. In an integrated hierarchy, enforcing a particular error recovery scheme is simple, since the basic methods are contained at the root of the tree, while in a forest this may only be enforced by mere convention. (Of course, the design and construction of a good exception mechanism is not at all simple in either case. Currently, the GNU C++ library relies on a simple client-resettable error function for each class.)

Because of the constant potential for inconsistencies across classes, users of forest libraries are more susceptible to the "glue" problem: How does one construct, coerce, or intercommunicate between members of two thoroughly unrelated classes?

A system presenting a consistent hierarchy may not be able to structure the tree of classes in a way that is best for every application. A tree builder must make occasionally arbitrary decisions about exactly where certain classes fit in the hierarchy, in ways that sometimes poorly reflect the interdependencies desired in a given application program. Generally, single hierarchies appear best suited for systems in which each class adds or further specifies functionality with respect to its parent, but other uses of inheritance are also possible within C++. The introduction of multiple inheritance will do much to alleviate such logistic difficulties.

A forest builder does not worry as much about such problems. For this reason, multiple inheritance will not impact the design of the GNU C++ library to a great extent, but may heavily impact programmers who use it to tie together otherwise unrelated classes. On the other hand, a forest builder should often make provisions so that independent classes are designed in a way that applications programmers can, with only a modicum of effort, interconnect classes into bigger forests and trees that make sense in a given program. Several such design considerations are described below.

2.2. Efficiency and redundancy

Classes living in forests are sometimes more efficient, more internally coherent, and more redundant than those living in a single tree. This is in part due to the fact that forest classes may be designed from first-principles, and need not contain otherwise unnecessary overhead required to implement methods and conventions used in the rest of the hierarchy. On the other hand, not being part of a hierarchy can limit code reuse, resulting in otherwise avoidable redundancies. These characteristics are also partly based on the fact that a single hierarchy must

make heavy use of virtual functions, which are significantly less prevalent in a forest. Indeed, in the forest approach, it is difficult for the library implementor to decide whether any particular method in any particular class might make most sense as a virtual since most often, clients, not other parts of the library, will create subclasses. Fortunately, C++ is often forgiving of errors in such decisions. In most cases, a programmer wishing to create virtual subclasses of any non-virtual class X may simply derive a new class VX in which no capabilities differ except that selected methods are virtual, and then use VX as the basis for all other subclasses. A similar strategy is available in the corollary case of proceeding from virtual to non-virtual.

2.3. C++ Limitations

C++ does not fully support the single tree approach because builtin types (like int) are not classes, and thus fall outside of any hierarchy.

C++ does not yet contain a parameterized class generation mechanism for the creation of basic container classes, for example a Stack class. This is not a major problem in the tree approach, since a Stack class can be declared and defined only once to operate on members of root class *Object*. This is a much harder issue in a forest, where class designers must rely on some variation of either of the two pseudo-generic class construction methods described by Stroustrup [3] (section 7.3).

In the first, "pointer" method, which is an approximation to the approach taken in "parametric polymorphic" languages like ML (see [4]), void* pointers are used as the actual elements of the container, and subclasses are constructed by coercing the void* elements into pointers to the elements of interest. In the second, "macroization" method, an approximation to the approach taken in Ada, nearly all the work is done by preprocessors, so that, for example, one Stack class prototype is transformed at the source-code level into the particular class declaration of interest via macro expansion or simple name substitution. Neither of these solutions is perfect: The pointer method introduces too much awkwardness and inefficiency to be useful for such everyday purposes as declaring a simple stack of int, while the macroization method can lead to unrestrained code redundancy and makes it impossible to cleanly express certain class interrelationships. Nonetheless, one of these methods must be used.

In the GNU C++ library, the prototype macroization method is employed, for one overriding reason: It is possible for programmers to generate a void* (or any other pointer type) container class from its prototype, and thereafter use the pointer method from this generated class, while the reverse is impossible. However, in terms of library design, this strategy does have a few disadvantages. Because users of container classes generate these classes in source file form from prototype files (using a simple text-replacement utility rather than via #define macros), this reduces the degree of control that the library designer has over use. Programmers are implicitly encouraged to alter the generated files in any way they see fit, both for better and worse.

3. Object-oriented libraries and bottom up design

The bottom-up software engineering methodologies encouraged by object-oriented programming languages affects basic support library designers perhaps more than other programmers. An object-oriented library provides common classes and utilities written in the hope that they may prove to be useful components of larger programs, but, generally without any special knowledge about how or why such objects might be employed. Indeed, during the brief existence of the GNU C++ library, there have already been several examples of "unintended" applications that library classes have found themselves in.

There are many attractive aspects of this kind of bottom-up design. Library builders ought to be able to concentrate much of their attention to the coherent design, implementation, and testing of classes in and of themselves, without being tied to any particular application. However, pure bottom-up axiomatic development is rarely possible in practice. There are several competing pressures, for the most part peculiar to C++, that interact with general class design strategies.

3.1. Featurism and data hiding

Given that most intrinsic operations are both more naturally and more efficiently performed from within a class rather than by client functions, it is much more difficult to limit the number and kinds of methods supported in a general-purpose class (say, a String class) than in one designed for special purposes. Thus the temptation to introduce a bit of "creeping featurism" enters into the design of virtually every general-purpose class.

An unthinkable "remedy" for this problem is to make all class data and support functions public, so that clients can operate on the underlying representations themselves in order to perform functions not directly supported by a class. But, of course, such an approach violates just about every virtue that object-oriented programming presents to software engineering efforts.

In the GNU C++ library a compromise position is taken. A certain amount of featurism is tolerated. However, in awareness that a general-purpose class might not exactly fit a client's needs, most underlying data are declared as protected. In this way, a client may efficiently implement additional functionality, but only via the introduction of new subclasses that both isolate such access and more properly reflect their motivation.

3.2. Design elegance and special casing

C++ programmers probably ought to expect that a class implements common operations about as efficiently as possible within its other design constraints. This can lead to the introduction of a nearly unbounded number of special-case operations in a general-purpose class, all of them logically superfluous. A good example of this can be found in the GNU C++ library String class: A String can be created from a simple char via a provided constructor. Also, a concatenation operator is provided that takes two Strings and returns their concatenation. Given only these, along with the C++ coercion rules, a client function can, of course, concatenate a

String with a char. However, a special purpose method for specifically concatenating a String with a char could perform this in a significantly faster way, without going through the constructor overhead.

In the GNU C++ library, this kind of special-case function proliferation, and corresponding near-redundancies of code are perhaps more prevalent than otherwise desirable. This is probably the right approach in C++. One of the major advantages of C++ over other object-oriented languages is its efficiency. Support libraries should not nullify this advantage.

3.3. C++ classes and C

Many C++ functions, including C++ class library functions, invoke standard C library functions. This fact influences the content of C++ support tools in several fairly obvious ways, but also impacts library class design.

First, the obvious ways. Provision of C++ "translations" of standard C header files is one of the minimal components of a library support system. The task of providing these has been made a bit of a challenge given the fact that C libraries and header files are currently undergoing revision in order to conform to ANSI and/or POSIX standards. Steps are underway to automatically provide GNU C++ compatible header files based on those from the forthcoming ANSI standard GNU C library. The portable use of these C library functions within C++ library classes is also a constant pragmatic concern.

A C++ library support system can also help serve as a "better C library" by providing minor extensions of common C library functions that operate on classless builtin types. As a convenience, the GNU C++ library contains headers and implementations for overloaded and/or inlined versions of abs, pow and the like.

A more interesting offshoot of the special relation between C and C++ lies in how C libraries may be used in designing C++ classes "by theft". Standard C libraries contain functions that have, for the most part, proven to be useful and reliable tools for programmers. However, such libraries are not always good C++ programming tools due to the fact that groups of related C functions often do not present themselves in a consistent, natural manner to C++ clients.

This line of reasoning suggests that a useful component of a C++ library might be classes that serve as "front ends" to groups of C library functions that might usefully be considered as classes. So long as the front end merely "repackages" C functions by calling them through inlines, etc., and thus serves mainly as "syntactic sugar", such a class allows C++ programmers to view the component functionality in a class-based, object-oriented manner without sacrificing efficiency or, perhaps more importantly, perfect C compatibility.

Currently, the library contains one major set of classes designed under this sort of strategy. Consider the fact that the C stdio library is, or would be if written in C++, a class: It possesses a state represented by private data (struct _iobuf — not actually declared as private, of course), a collection of methods (fread, fprintf, etc. — actually regular C functions), several of which are inlined (getc, putc — actually macros), and selected state access and control functions

(feof, setbuf, etc.). This stdio "class" is surely the most well-used set of C library functions.

The GNU C++ library repackages C stdio as class File. However, it does so while also providing somewhat more security, convenience features, and consistency. Moreover, several mainly cosmetic changes are implemented to make the File class use Stroustrup's stream package state and error conventions. These modifications allow the File class to serve as a base class for the *libg++* istream and ostream classes as well as other I/O related classes, such as one that generates *plot(5)* format files.

The istream and ostream classes thus perform I/O via File class operations that call stdio functions. This strategy has the advantage of minimizing annoying minor discrepancies between the C and C++ I/O, while maintaining the distinct advantages of C++ stream I/O over stdio function calls. However, this has the disadvantage of not being 100% compatible with the *de facto* standard AT&T stream package, since File based streams do not have streambuf components. The GNU C++ stream package is undergoing modifications to comply with the AT&T version while otherwise maintaining this general design strategy.

The GNU C++ library does or will contain several other C library based classes, including such "natural" ones as those based on the GNU C library versions of the getopt, env, and dbm functions.

3.4. Class interfaces

Of course, classes should be designed in a way so that they are easy and natural for client functions to use as ready-made "off-the-shelf" software components. Unfortunately, there are no hard and fast guidelines that ensure successful translation of this cliche into practice. Generally, only weak and obvious rules of thumb apply, including (1) minimizing of the use of overloaded operators in non-standard contexts (as in the poor choice of the += operator to perform a stack push); (2) using default parameters in order to provide flexibility in special cases without requiring overly detailed specification in the most common cases; (3) using references rather than pointers; (4) minimizing "surprise" by making clear those cases where simple-looking methods do not have simple behavior and cases where implementation code performs significant behind-the-scenes memory allocation and/or computation; (5) minimizing cases where client functions must themselves deallocate space allocated within class methods; (6) limiting the number of methods and friends within a class; (7) using friend functions or operators for operations that construct new objects without modifying their argument(s), members for those that do mutate the object, and minimizing the number of operations that do something in-between; (8) minimizing conflicts with other C and C++ libraries and utilities; and (9) maintaining proper class documentation and clear header file formatting.

4. Library classes

No taxonomy of objects perfectly categorizes the kinds of classes one can construct in C++, nor cleanly separates what makes a good library class from an applications class. However, the following categorization of the contents of the

GNU C++ library helps to organize discussion of some basic commonalities (see also [2]).

4.1. Algebraic types

Surely a library is the right place to implement classes that extend C++ to support additional general-purpose basic types found in other type-rich languages. Such types include complex numbers, rational numbers, multiple precision integers, strings, and the like. These types generally possess widely agreed-on properties and capabilities based on algebraic specifications.

There is a simple goal in the design and implementation of such classes: Excepting minor differences between classes and classless builtins in C++, a client function should be able to employ an extended type in exactly the same ways as a similar builtin type. Moreover, this generally ought to be the preferred and best-supported usage of the extended type. Thus, for example, a client function should be able to replace all int variables with instances of the multiple precision integer class Integer wherever meaningful, with a minimum of additional effort.

4.1.1. Scalar semantics

Algebraic classes ought to possess "scalar semantics": Clients should never need to perform "manual" storage management for instances, should be able to use parameter passing by-value when meaningful without penalty, and should be able to rely on the natural use of binary and assignment operators.

These criteria are easy to meet for a class like Complex. Since it contains a small amount of fixed-length data, all storage allocation is performed via the usual C++ rules without any dynamic allocation. Call by-value (versus by-reference) may or may not be the more efficient parameter passing mechanism for Complex variables being logically passed by value, depending on the function and context, but is in any case close. Simple copying and construction can be used to efficiently implement assignment and operators.

4.1.2. Representations for variable-sized objects

Scalar classes like multiple-precision Integers and Strings containing variable-sized data present more of a problem. Perhaps the most common compromise solution for representing such types is an indirect pointer scheme in which class instances merely contain a pointer to other structures actually containing the data, and which are dynamically allocated and managed entirely behind the scenes. The challenge of this kind of solution is how to implement a transparent storage management system that contains enough advantages so as to outweigh the constant cost of the additional level of indirection imposed via pointers.

The GNU C++ library adopts one version of this approach. The basic technique used across all variable-size scalar types is the same, although various details necessarily differ from class to class. The general strategy for representing such objects is to create chunks of memory that include both header information (e.g., the size of the object), as well as the variable-size data (an array of some sort) at the end of the chunk.

Class instances themselves contain only a pointer to a chunk. Storage management is performed via reference counting. All headers possess a reference count that is maintained in the usual way for storage reclamation. This representation scheme (which is similar to one often used in implementing SNOBOL) has several notable features.

4.1.2.1. Allocation

Combining headers and data results in generally fast memory allocation, since only one, not two allocations are necessary. Expanding and contracting the variable part is also straightforward, as are techniques for automatically preallocating and overallocating space for variables that appear to be changing. For example, Strings are usually overallocated space whenever they grow, as a heuristic aimed at minimizing future allocation overhead and free store fragmentation.

4.1.2.2. Constructors and assignment

Use of pointers allows both by-value and by-reference argument passing to be equally efficient. A by-value argument invokes the X(X&) constructor, which consists only of a pointer copy and reference count increment. The assignment operator requires only slightly greater overhead.

4.1.2.3. Sharing

The mechanism allows several variables to point to the same chunk. The sharing of values across, especially, elements of arrays, appears common enough to make this an effective mechanism for reducing overall storage requirements.

4.1.2.4. Chunk reuse

Operator-based assignment (e.g., operator +=) is also efficient. Because of reference counting, objects that are the results of various self-modification operations (like destructive String concatenation) may often be refit back into their sources if they are big enough and are not shared, without any additional allocation. If they are unshared, but not big enough, they may be reallocated.

4.1.2.5. Manual allocation

Since neither this mechanism nor any other strictly behind-the-scenes dynamic storage management scheme can be optimal for all applications, methods are available for clients to take over some allocation chores in order to fine-tune their code. Each class includes a member function that forces data copying (not pointer sharing) from one variable to another, a function that forces a variable to point to an unshared chunk, and a function that preallocates chunk space for a variable. In the library documentation, programmers are given examples of cases where manual tuning of storage allocation is likely to be worthwhile.

4.1.3. Integration

Algebraic types should be integrated into C++ in ways that hide, as much as possible, the fact that they are not truly "built into" the language. In *libg++*, most such classes possess converters that allow automatic coercion *both* from and to

builtin types (e.g., char* to and from String, long to and from Integer, etc.). There are pro's and con's to circular converters, since they can sometimes lead to the conversion from a builtin type through to a class function and back to a builtin type without any special attention on the part of the programmer, both for better and worse.

Because these kinds of classes bear such close relation to builtin types, they are most susceptible to the problems (and/or advantages) of special-case function-proliferation discussed above. It is often conceivable to support all possible combinations of mixed mode operations between builtins and class variables. It is rarely clear which of these are actually worth implementing.

A final concern is that of consistency with commonly used syntax for algebraically defined types. This sometimes leads to the choice of established "friend" syntax when "member" syntax might make more logical sense. In fact scalar types often possess many more "intrinsic" friends than do other kinds of classes, again mainly owing to established conventions.

Classes like String, in which many different operator notations are in use, require a certain arbitrariness. Should the concatenation operator be +, *, &, | or perhaps something else? In *libg++*, it is +. No choice will "look" right to a casual reader of String application code who expects + to indicate some kind of addition, * multiplication, etc.

C++ does impose a few limitations on the degree to which extended types can directly emulate the behavior of builtins, including for example, the fact that operator ++ always works as if applied as a prefix operator. Among pragmatic limitations is that the natural declaration of operator = to return a reference to self, so as to support constructs like a = b = c; and if ((a = f()) == 0)...; rather than its value (so as not to waste copying values in order to support such relatively rare constructs), can lead to minor anomalies, such as the fact that (a = 1) = c; would be a C++ error if a were a builtin, but not if it were an extended type. Clients should also be made aware that while the op= operators (+=, *=, etc.) often result in only small efficiency improvements over constructive operators (+, *, etc.) when used with builtin types, they are nearly always substantially more efficient for extended types, since their use eliminates the need to construct, destroy, and/or copy compiler-generated temporary objects.

4.2. Container classes

Container classes are objects like stacks and linked lists that hold collections of elements, all of the same known type. These kinds of structures are also a natural component of a C++ library, for the pragmatic reason that they are very commonly useful in applications programs, but more importantly because, they, like algebraic types, possess widely agreed upon properties based on their characterizations as Abstract Data Types (ADTs). The majority of common container classes are based on some variation of a basic *Set* ADT, supporting insertion, deletion, the inspection of elements in a collection with the following properties.

4.2.1. Reference semantics

Once created, container classes normally undergo sequences of mutations. Thus, they are nearly always logically passed to functions, etc., by reference. Moreover, for both efficiency and soundness, they possess essentially no constructive operations. Even binary operations like set union are probably most desirable if implemented destructively (via operator |=) rather than constructively (via operator |). Generally, the only constructive operation that requires support is a X(X&) constructor so that clients may create new containers based on existing ones.

4.2.2. Mutable elements

Container classes normally require the existence of a method for clients to inspect their elements one-by-one. This usually requires the creation of a companion traversal class in order to maintain information about the state of a traversal. Other element access or inspection methods are also sometimes desirable as well, as in the need for a method to retrieve an element based on its key in a keyed dictionary class, and the convenience of supporting a method that applies a given function to each element of a container without the need for a traverser.

There is a data security problem inherent in the design of any element access method. The most sensible implementation of an access method (and the one employed in *libg++*) is to have it return a reference to the stored data, and not the value. The alternative of having access functions return values is inappropriate for the majority of container class applications. Yet the return by-reference strategy is fundamentally insecure. A client function may use this reference to modify the item in a way that could corrupt the underlying data structure, as in the case where a key field is modified in a container that stores items in ascending order of keys. There is no language mechanism by which the container class itself can ensure that only non-key fields are modified by clients. Clients must build in such security themselves.

However, there are container classes that avoid such problems entirely. Variations on associative tables in which the parts of elements that are used as the basis for structuring the container (i.e., the keys) are securely maintained separately from the rest of the elements are attractive kinds of container classes in C++ for this reason, as well as the overall simplicity of their client interface.

The converse case of a container class guaranteeing that it will preserve its elements without modification is seldom a problem. It is natural and not at all difficult to guarantee that the container will never itself perform any operations that modify its elements.

4.2.3. Element characteristics

Container classes possess operations that function in the same way regardless of the exact type of their base elements, and are thus logically parametric types. As discussed above, since C++ does not yet support parametric types, the GNU C++ library generates such classes via macroization. However, any container class must make some kinds of assumptions about the characteristics of its constituent elements in ways that are not directly provided by such a mechanism.

The minimal requirement for an element to be eligible for use in a container class is that it possess an X(X&) constructor. Otherwise container slots holding the element could not be created. All GNU C++ library containers are designed to insert elements via a constructor. Clients desiring classes that maintain pointers to elements stored elsewhere may declare the containers to hold pointers.

A second requirement is that the container must know how arguments will be passed into basic methods, for example an element deletion method: Should the argument be passed into method del by-value or by-reference? The best answer to this depends on both the type of the element and on the application. While neither choice is logically incorrect, a user of a list of integers would prefer by-value, but a user of a list of stacks would prefer by-reference. For this reason, the GNU C++ library textual substitution mechanism requires users to declare whether by-value or by-reference argument passing is to be used, and adjusts the generated source code accordingly. Note that this strategy is not designed to eliminate the use of constructors during element insertion.

Container classes are designed to make one additional accommodation to this issue. Classes like stacks provide a method to simultaneously delete and return a value (i.e., the pop operation). In cases where the elements are large or normally passed by reference, a superior alternative is to provide a method top, that returns a reference to the stored item, as well as a void method delete_top that silently removes it, thus minimizing value copying. These methods are useful for containers of objects that spend their entire existence within the container and are never used or constructed elsewhere. The GNU C++ library classes support both kinds of access.

4.2.4. Element inspection

Container classes that are structured around particular properties of elements must somehow be informed about how to access these properties. For example, sets supporting a membership operation must know how to determine whether two elements are considered equal, ordered sets must know how to compare two elements, and hash tables must know how to compute hash functions. There are several possible approaches to gathering and using such information.

Perhaps the easiest method to implement is to require that elements possess particular operators. Thus a container with a membership operation might contain code using operator ==, which would simply fail to compile when applied to base type not possessing a == operator. This method appears entirely appropriate for containers like unordered set classes in which no additional structuring information is required, and for which the notion of equality is fundamental to class operation.

This approach is less justifiable for many other container classes. Consider an ordered set implemented via a binary search tree. The minimal requirement for base elements might be that they support a <= operator. However, the underlying algorithms are generally best implemented in ways that require a full *compare* function that returns whether one element is less than, equal to, or greater than another. Such a function can most reasonably be provided via function pointers provided by a client and held by the class. However, it is not particularly desirable to design constructors that require comparison function pointers as arguments, nor

for each instance of a binary search tree to hold its own pointer to a comparison function. One important drawback is that such a strategy can preclude the implementation of operators like set union that need to assume that the ordering functions for both sets are the same. Similar remarks hold for classes that require hashing functions and the like rather than comparison functions.

To avoid this problem in the GNU C++ library, such function pointers are declared as static, and thus shared across all instances. This strategy is not without drawbacks: Correct class operation depends on the client function properly initializing the function pointers. The need for initialization by an unknown function requires that they be declared as public, which is otherwise undesirable. Also, it is only possible to create instances based on the same elements but employing different ordering functions by generating completely separate classes.

On the other hand, there is an additional positive byproduct of this strategy. Because comparison functions are designed to be shared across all instances, the GNU C++ library allows the functions to be fully specified at compile time rather than run time via C++ preprocessor statements that force the class comparison functions to use a supplied function if one is #defined, else to go through the static pointers. This allows users to "hard-wire" comparison functions into the code if so desired, and also makes it possible to perform comparisons entirely inline, with no indirect function call overhead.

For container classes structured on only parts of elements (i.e., keys), this strategy is extended one further step. Keyed classes require two function pointers, one to compare two elements, and one to compare a key with an element. (There is an alternative: one could specify two different functions, one to compare two keys, and one to extract a key from an element, but this is potentially less efficient and often harder to use.) In fact, in the library, both keyed and unkeyed ordered sets are provided using the same class prototypes. Two function pointers are required, but in cases where items serve as their own keys, both should point to the same function. Again, similar remarks hold for classes based on hashing, etc.

4.3. Vector-like classes

Classes like vectors, matrices, and Lisp-style lists behave like algebraic types in some respects and container classes in others. Vectors, for example, support standard constructive operators, like +, are frequently assigned, and so on, yet serve as simple container classes as well, providing an element membership test.

The main conflict in designing such classes is the storage management policy and its degree of transparency to the client. Thus far, GNU C++ library classes have merely followed convention in such matters.

Vectors are implemented in the usual way, as dynamically allocated arrays deleted upon destruction. Clients are expected to know how to perform common efficiency measures like passing by-reference and preallocating space.

Lisp-like lists are, on the other hand, managed in a basically Lispish manner, completely transparently to clients. ("Lisp-like" is actually a slight mischaracterization of this class, which implements homogeneous lists like those found in statically-typed functional languages, but otherwise supports operations similar to those found in Lisp.)

This categorization of classes concerns the way in which clients view classes and their capabilities, not necessarily the primary data being represented or the implementation algorithms themselves. For example, the GNU C++ library currently supports three different classes for manipulating sequences of bits: Class BitString is designed using the "scalar" model, class BitSet using a kind of "set" model, and class BitVec using a "vector" model. All three contain arrays of data manipulated via bitwise operations, but differ in how these data and operations are managed and presented to clients.

4.4. Storage Management Utilities

C++ was designed with the philosophy that that programmers, not compilers or run-time systems, are best able to perform storage management. Given this, one may reasonably desire that a C++ library include general-purpose support classes and utilities to assist in storage management details. However, there is less opportunity to do so than might appear, for essentially the same reasons that C++ does not offer this kind of support itself: Without knowing the particulars of their application, general-purpose utilities for say, garbage collection-based freestore management, are bound to be both suboptimal and inconvenient (at best) to apply. There are some exceptions.

The GNU C++ library contains class Obstack, designed to be a general purpose allocation mechanism for dealing with variable-sized objects that must have space allocated but are encountered, one at a time, "head-first", without knowing their ultimate size. Nearly any text scanning or parsing application fits this scheme: words or tokens are read letter-by-letter without knowing their lengths. Obstacks handle this by preallocating larger chunks of space and breaking them off into their components, while also reallocating and copying to new freestore space when current chunks become exhausted in the midst of the growth of a new object. When a client has determined that an object has finished growing, a finish method returns its final start address. The space is allocated as a stack, so can be freed by popping off no longer needed objects. This storage allocation method was discovered by Richard Stallman and is used extensively inside the GNU compilers.

The library also contains a simple general-purpose class designed for allocating fixed-sized objects (as might be useful for linked list nodes) out of a preallocated chunk of memory.

4.5. Service Classes

Service classes are those designed to encapsulate a particular collection of services, and whose internal state representations are essentially irrelevant to their clients. Good examples of service classes are the stream classes, other classes repackaging C Library functions described above, random-number generators, and many applications-oriented classes.

The design of service classes within a library can be problematic because of the arbitrariness of so many of their characteristics. Even given a clear idea about the constituent representations and algorithms, one is never sure whether too much or too little functionality is supported in a given class, whether to break classes up into finer units or integrate several as a whole, whether a collection of several unrelated service classes are usable together, or even if the names chosen for the

classes and their methods will be considered appropriate. Of course these are the kinds of "little" design issues that haunt object-oriented applications programming, and for which there are few principled guidelines.

5. Conclusions

The GNU C++ library is an interesting experiment in managing a basically disorganized collection of classes. The library is still quite new. Additional classes are being added at an increasing pace, as other GNU C++ users begin contributing classes for adaptation into the library. Little can be said as yet about its suitability for particular applications or its general usefulness in practice. Tentatively, however, a few observations can be made.

The emphasis on efficiency and flexibility appears well-placed. Thus far, most substantive correspondence with library users has addressed accommodations revolving around these two issues rather than other aspects of class design. In particular, few concerns about interclass consistency, "glue" problems, or redundancy have been expressed. This is surely in part due to the fact that the GNU C++ library is finding a different audience than the one best served by systems like *OOPS*.

The GNU C++ library appears well suited for small-scale applications programs in which only one or a few library classes are pulled "off the shelf" without otherwise committing to a purely hierarchical programming environment. It appears likely that larger scale software engineering efforts will use (perhaps modified versions of) GNU C++ library classes mainly as ready-made pieces of a larger hierarchy otherwise designed from scratch.

Object-oriented programming languages represent a rather exciting merger of theory and practice. Perhaps even more attractive is the fact that neither the theoretical basis nor the pragmatic consequences of object-oriented programming methodologies are yet particularly well understood. A minor contribution that both the authors and the users of a class library can make toward a better conceptualization of C++ and object-oriented programming lies in the kinds of observations that may be gleaned from experience with a library's "bottom-up" approach to these issues.

References

[1] Gorlen, Keith E. An Object-oriented class library for C++ programs. *Proceedings of the USENIX C++ Workshop, Santa Fe, NM*, 1987.

[2] Rose, John R. Implementing a compiler in C++. *Proceedings of the USENIX C++ Workshop, Santa Fe, NM*, 1987.

[3] Stroustrup, Bjarne. *The C++ programming language.* Addison-Wesley, 1986.

[4] Cardelli, Luca and Wegner, Peter. On understanding types, data abstraction, and polymorphism. *Computing Surveys, 17* (1985), 471-522.

C++ Approach to Real-Time Systems: Task Interface Library

Troy Otillio
Tandem Computers
19333 Vallco Parkway
Cupertino, California 95014-2599

Abstract

CPPARTS (C-Plus-Plus-Approach to Real-Time Systems), a C++ class hierarchy, was the result of an academic project aimed at constructing several tools (objects) to aid concurrent/real-time systems development. The definition of these objects originated from the abstractions put forth in Gomaa's articles on DARTS (Design Approach to Real-Time Systems) [Gomaa 84, 86]. This paper introduces those interfaces in terms of concept and implementation technique. Included is an examination of the difficulties of using C++ in a distributed process environment. The inconsistency between distribution and inheritance is analyzed and solved for specific implementation scenarios.

Introduction

DARTS--a design method for real-time systems--leads to a highly structured modular system with well-defined interfaces and reduced coupling between tasks [Gomaa 84]. CPPARTS uses the specifications for task interfaces as requirements for implementation of corresponding C++ classes. CPPARTS provides a real-time or concurrent system programmer a dependable, complete, and easy to use set of task interfaces with built-in error handling. Although DARTS was defined for real-time systems development the academic investigation of CPPARTS is limited to the scope of multi-process system development and thus some relevant real-time issues, timing and performance, are not addressed.

Unlike many class based systems CCPARTS class instances operate on common data and system resources. *Instance grouping*, the means for specifying a logical set of instances from one class is accomplished through the use of *binding keys*. Binding keys and instance grouping are discussed in detail. Other problems associated with C++ implementations in an environment where processes occupy separate address space (distributed process environment [Wegner 87]) are examined and solved. In general this paper discusses the problems, solutions and work-around's for the combination of the specifications of DARTS using the C++ programming language in a distributed process environment. To present the topic it is necessary to discuss concurrency, the class structure, binding keys and the implementations of the root and leaf classes.

Because DARTS is an object-oriented solution to real-time systems development, and C is a common systems programming language, C++ was chosen as the implementation language. The development of this set of task interfaces was constructed using AT&T C++ and the Unix system V primitives on a Pyramid 98X at California Polytechnic State University in San Luis Obispo California.

Due to the limited scope of this paper and because there are better sources to describe the value and experiences of using the DARTS task interfaces [Gomaa 84, 86] this paper does

not discuss systems development using CCPARTS. However, the solutions presented are in part a result of major systems development using the CCPARTS classes.

Concurrency

A sequential program specifies the sequential execution of a list of statements; its execution is called a process [Andrews 83]. Concurrent programming differs from sequential programming in terms of three issues: how to express concurrent execution, how processes communicate and how processes synchronize their actions. Concurrent systems consist of several parallel processes. Each process is sequential and concurrency is achieved by having asynchronous tasks running at different speeds. Periodically, processes need to communicate and synchronize with each other. [Gomaa 84]. For the remainder of this paper the term process, the term task and the term program will be considered synonymous. Furthermore, the distinction between parallel process, distributed process, and concurrent process need not be recognized.

CLASS OVERVIEW

The final CPPARTS class structure, as shown in figure 1, was influenced by three factors. The leaf classes were created in accordance with the task interface definition presented in DARTS. The base classes were influenced by the IPC (Inter-Process Communications) facilities needed to implement the leaf classes: message queue, semaphore and shared memory. The lack of a symbolic debugger and an error recording facility as well as the difficulty of debugging multi-process systems created a need for an additional class to allow for tracing and control of error and debugging messages. This resulted in the creation of the root class for all CPPARTS classes: MESSAGE_LOG. The message logging facility provides derived classes a means to record errors, intermediary data conversions and other relevant debugging information.

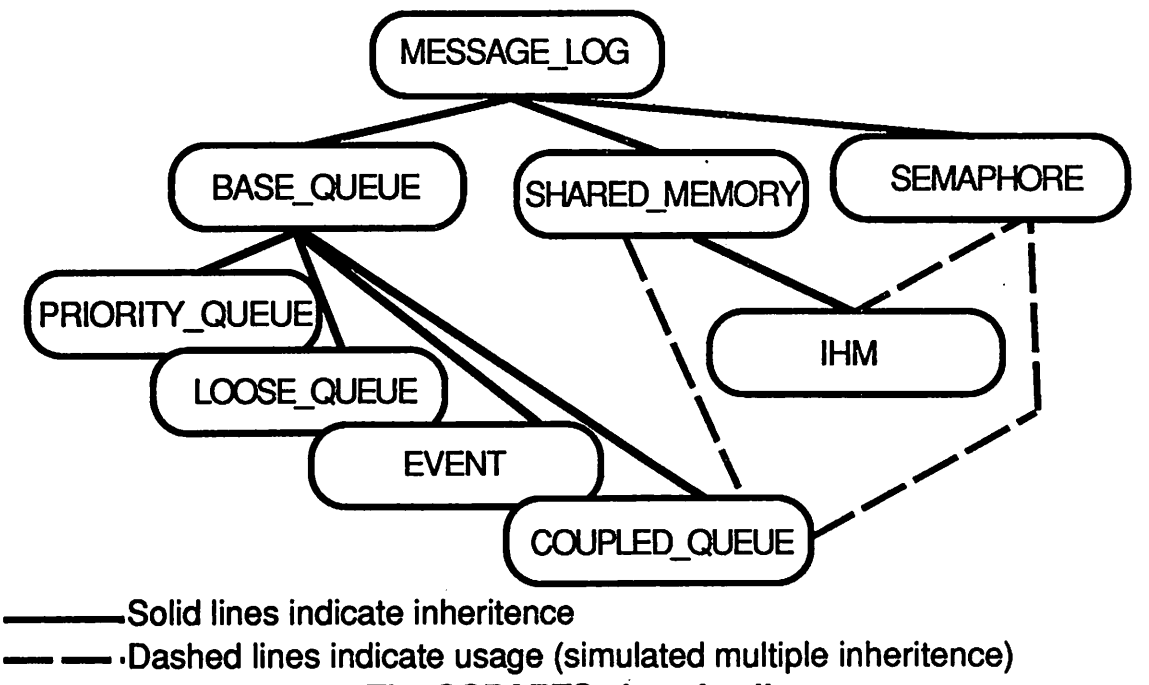


Figure 1: The CCPARTS class family

BASE_QUEUE is an encapsulated version of the IPC message queue facility that utilizes the UNIX signal/alarm call to implement timeouts for blocked queues. SHARED_MEMORY and SEMAPHORE are similar to BASE_QUEUE in the sense that they are modeled after the matching IPC facilities semaphore and queue.

EVENT is a class to implement the DARTS notion of task synchronization where one task can wait for any one of several tasks to signal an event.

IHM (Information Hiding Module) is another DARTS task interface providing a section of shared memory with guaranteed protection of mutual exclusion.

LOOSE_QUEUE and COUPLED_QUEUE were both derived from BASE_QUEUE. They provide DARTS task interfaces tightly coupled and loosely coupled queue respectively. PRIORITY_QUEUE was not defined in DARTS, but was a logical and fairly trivial class to implement in addition to BASE_QUEUE.

CLASS INSTANCE LINKING: BINDING KEYS

Specific to the CPPARTS task interfaces is the need for groups of one or more class instances to reference common system resources. For example, the IPC facility semaphore requires a logical key to be passed to the semaphore allocation function: semget(). Semget() returns a system key which is used as a parameter to the signal/wait function: semop(). The base classes for all the CPPARTS objects are encapsulated IPC resources and thus all CPPARTS objects require a grouping mechanism.

An elegant solution might be to use the class instance name to logically bind a set of peer instances (peer in this case means of the same class type). Specifically, this would be accomplished by using the instance name string to create the key.

The following page provides an example:

```
TASK  A                          |     TASK  B
---------------------------------------------------------------
QUEUE disk_read_request();       |     QUEUE disk_read_request( );

>>Within the QUEUE constructor
QUEUE::QUEUE () {
      this->key = string_to_unique_key(INSTANCE_NAME);
      request_system_queue( this->key );
      // etc.
}
```

Because C++ does not provide a means to access the instance name, as prototype INSTANCE_NAME above suggests, an alternate scheme must be adopted.

In one sense this problem of binding peer instances is similar to a problem encountered in languages that support the decomposition of a program into modules but do not support "user-defined types". For this environment the solution to creating user types, grouping data and operations, requires the creation of type manager modules [Stroustrup 87]. When using this simulated data abstraction scheme one must always include an identification key to the parameter list for any operation. CPPARTS uses a similar solution: class instance groups are bound by providing an identification key to the constructor's parameter list. The CPPARTS method for binding a set of class instances is to send a *binding key* to the class constructor.

Problems using Binding Keys with embedded classes

Object-oriented solutions should hide details of implementation from the user. Since the CPPARTS class hierarchy maintains several inter-dependencies (see figure 1) a problem arises in the use of binding keys to bind the embedded classes. Keep in mind that all CPPARTS classes require a binding key to allow access the same base IPC facility. It is reasonable to expect the CPPARTS user to group two different types of task interfaces with the same key (see figure 2). In fact this binding key convention, re-using a key for different CPPARTS types, is good style; instead of using a number (44 in figure 2) a constant called bind_process_A_and_B could be used for all interfaces needed between process A and B. However, the user supplied binding key must be used to bind parents of the derived class, but not for the created class since this may induce side-effects.

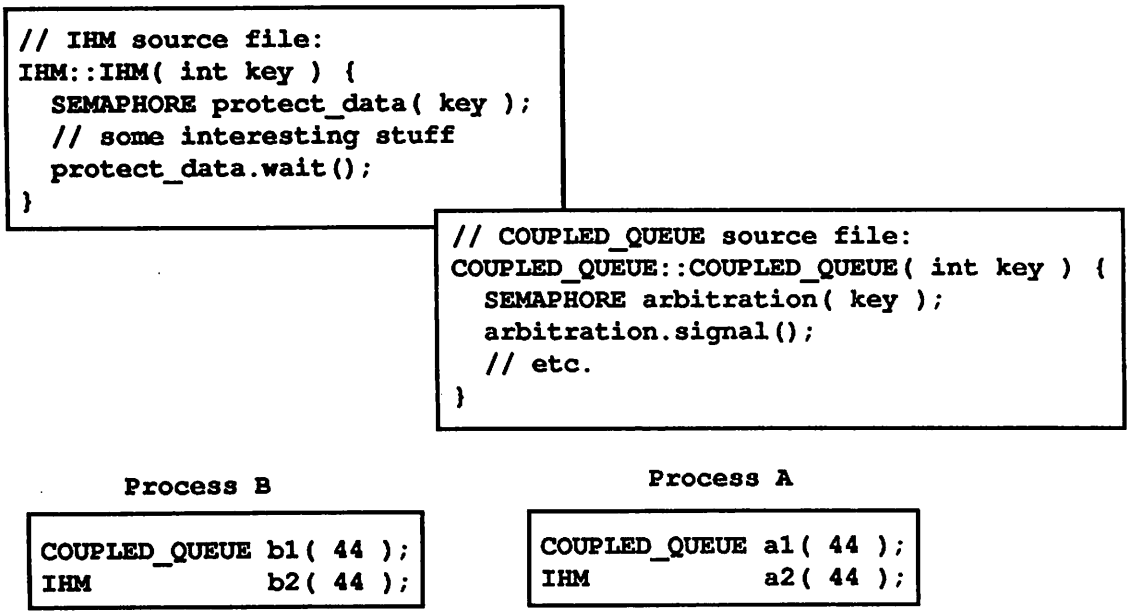


Figure 2: Use of Binding Keys

Realizing both IHM and COUPLED_QUEUE create an instance of class SEMAPHORE the implementation shown in figure 2 would causes havoc since two unrelated class groups, (a1,b1) and (a2,b2) would have the undesired side-effect of signaling and waiting on the same semaphore. The CPPARTS solution is to create key sets for each leaf or interface class through use of one standard "key" file. Each CPPARTS class adds a constant offset to it's key to insure no cross-binding occurs among different CPPARTS instance groups. The offset, which is unseen and unnoticed by the user, forces each class's binding key into an unique physical range. The result is added elegance and improved reliability at the cost of forcing the user to use keys from a single source.

Key file

```
// Standard shared key file
const key_type A_KEY = 1;
const key_type B_KEY = 2;
const key_type C_KEY = 3;
const key_type LAST_KEY = C_KEY + 1;

const key_type SEMAPHORE_KEY_OFFSET       = LAST_KEY;
const key_type STANDARD_QUEUE_KEY_OFFSET = LAST_KEY * 2;
const key_type IHM_KEY_OFFSET             = LAST_KEY * 3;
```

```
// IHM source file
COUPLED_QUEUE::COUPLED_QUEUE( int key) {
 // some interesting stuff
 SEMAPHORE  arbitration( key + COUPLED_QUEUE_KEY_OFFSET );
    // used constructor's binding key and
    // the key offset in "keys.h"
}
```

```
// IHM source file
IHM::IHM( int key ) {
// some interesting stuff
 SEMAPHORE  protect_data ( key + IHM_KEY_OFFSET );
    // used constructor's binding key and
    // the key offset in "keys.h"
}
```

Figure 3

Now the user is hidden from the implementation details and thus the code in figure 2 would not produce any unwanted side effects. The responsibility of the user shifts from understanding the interdependencies of CPPARTS to modifying LAST_KEY in the key file when the number of keys must be increased

Closely and loosely coupled queues

A loosely coupled queue, as per DARTS, is an interface in which a task may wait for a message to arrive at any one of several message queues [Gomaa 84]. This interface is implemented in the LOOSE_QUEUE class. Closely coupled message communication is accomplished via class COUPLED_QUEUE where sending and receiving of replies are supported by having a one-element message queue in each direction -- one for messages and one for replies [Gomaa 84]. As with all the CPPARTS classes binding keys are supplied to the constructor to group peer class instances to persistent system queues.

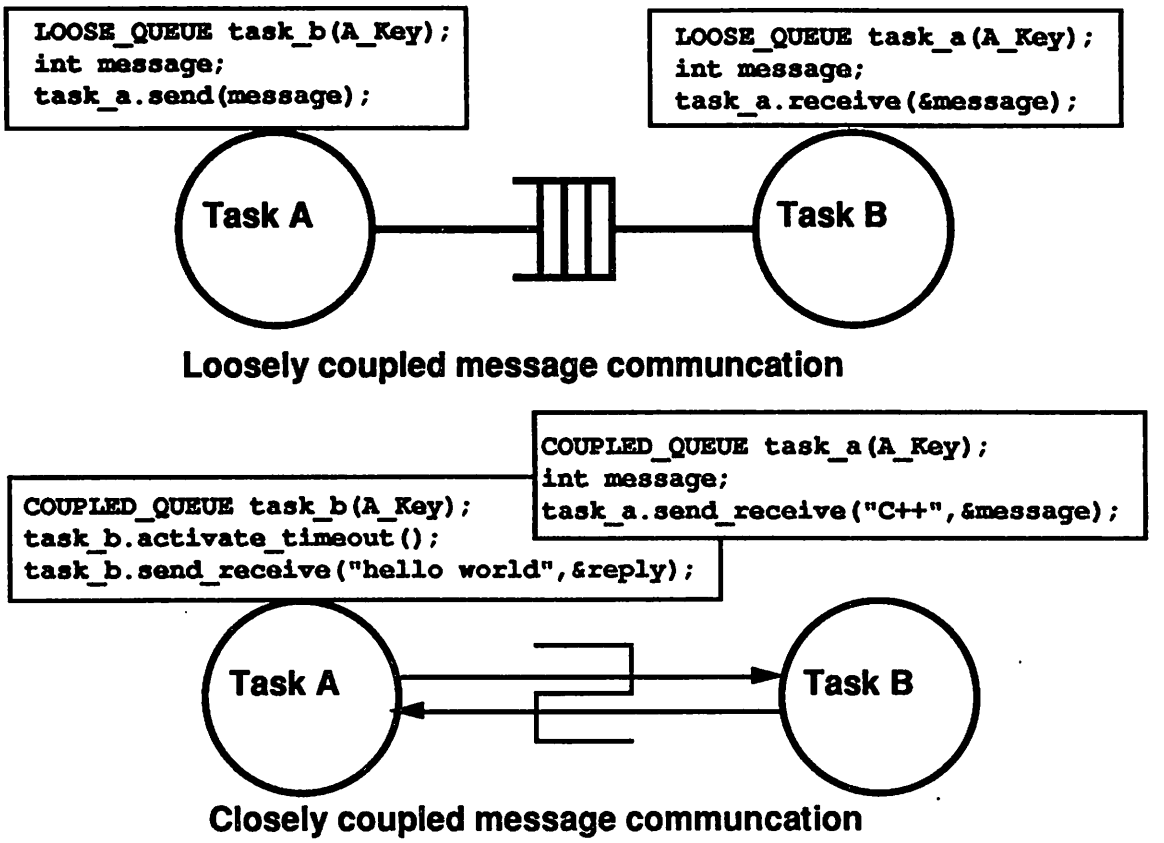


Figure 4

An interesting user error that can be foreseen but not eliminated at compile time is the message type consistency required for peer instances for both types of queues. As shown in figure 2 for Coupled Queue, Task A is sending a string, but Task B is expecting an integer. The implementations for all three queues are able to simulate type consistency per instance pair by embedding type information in the queue element. An alternative would be to create an added class for each type of queue element. Since elements might typically be structures and thus infinite in number of different types this exercise should be left to the user. Even if C++ supported parameterized types the problem of insuring each instance pair used a consistent types would still exist. By embedding the type information in each queue element a run-time error can be detected and instance pair compatibility can be enforced.

A coupled queue allows the simulation of sequential execution between two tasks. Sometimes a logical process needs to be physically divided between processors to enhance performance; the simulation of sequential execution is a simple and powerful means to affect division [Gomaa 84]. Implicit in the definition of this task interface is the need to exclude the use of the parent class's, BASE_QUEUE, timeout facility. If a timeout was allowed to occur the queue would become unsynchronized and the data in the queue would lose meaning. The translator used for the CPPARTS project provided no compile time ability to exclude the timeout member functions. The only alternative was to re-define the base class timeout functions to indicate an error and terminate. Thus figure 4's accompanying code for coupled queue would compile but cause a run-time termination since Task A invokes the member function activate_timeout().

EVENT

Events are used for synchronization purposes between tasks where no actual information transfer is needed [Gomaa 84]. A destination task may wait for an event occurrence, or a source task may signal an event that activates the destination task. The EVENT class is an implementation of this concept allowing a process to signal or to accept an event via member functions.

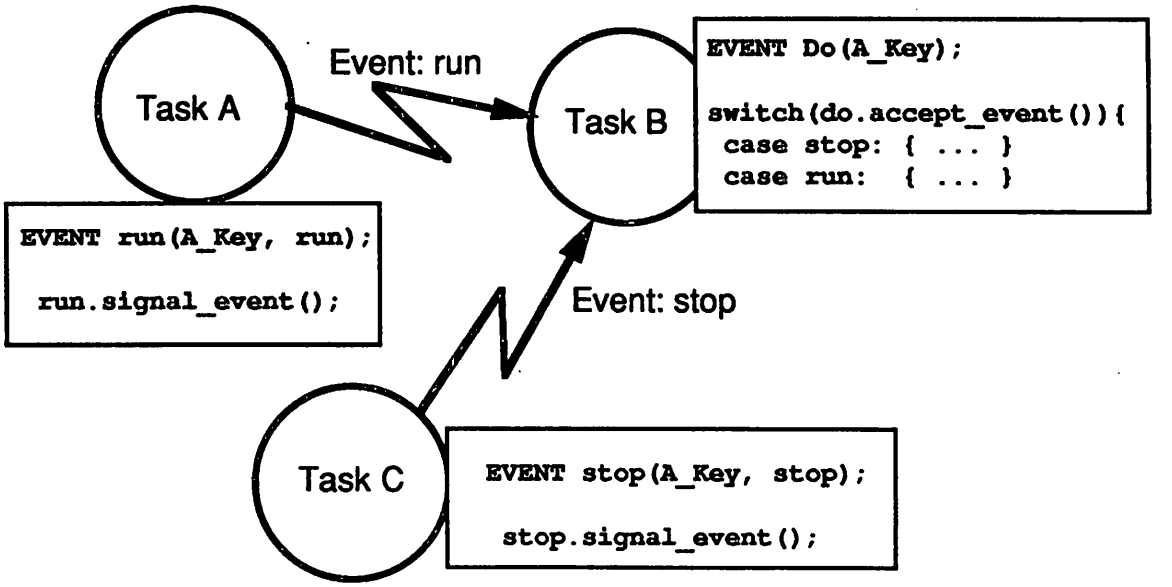


Figure 5

The current implementation of class EVENT contains both accept and signal constructs. The signalling constructor accepts two parameters: a binding key and an event type. The member function signal_event() sends the signal to the receiving task. To differentiate the signalling event instance from the destination event instance the constructor for the destination instance is of a differing polymorphic type (i.e. it has one less parameter).

Since the BASE_QUEUE class was used as the parent class the implementation itself was trivial and error free. The destination task is blocked on an empty queue until a signaling task sends an event to the queue. The EVENT class supports integrity checking for accept_event() and permits the event to be of differing polymorphic types.

Further integrity should be provided to insure that for any given event group there exists only one destination instance and one or more source instances system wide. One solution could be to create separate child classes of EVENT: one having the accept construct and one having the signal construct. This would cause the user to explicitly state the type of class (signalling or accepting) at definition. Although it might prevent some coding errors there is still no guarantee that only one destination instance exists and one or more source instances exist per instance group. This problem could be addressed at run time via an integrity table in shared memory. The scope of the CPPARTS interface is limited and this scheme is not currently implemented. Could a compiler/linker perform this type of check? What form would the syntax take and what semantics could be expressed? It raises the question about the kinds of type checking an object oriented language supports.

Class IHM

The IHM class provided the most challenging implementation since it nearly required the support of parameterized types when of course C++ does not support parameterized types [Stroustrup 87]. The IHM class was defined in DARTS as a data store used for reference purposes. In DARTS an IHM defines the data store as well as the access procedures to it [Gomaa 84]. The CPPARTS object IHM is accessed as an array of elements where each element is a set of bytes. Thus the user is forced to view the data pool as a one dimensional array of homogeneous elements. A binding key binds a set of class instances to the same common memory. The IHM facility insures mutual exclusion for reading and writing of data as well as run-time bounds checking. The '>>' and '<<' operators are overloaded to invoke reading and writing to the data store. For completeness here is a selected listing of the interface file:

```
~IHM();
IHM()(int key, int unit_size, int low, int high = 0);
                    //if high is defaulted the bounds are 0..low
int low();          // returns low bounds
int high();         // returns high bounds
int size();         // returns the cardinal size of the array
IHM operator[](int index);    // element selector
void operator << (void* some_struct);     //Writing an IHM element
                            // usage: <IHM element> << <some_struct>
void operator >> (void* some_struct);     //Reading an IHM element
                            // usage: <IHM element> >> &<some_struct>
```

Although mutual exclusion, bounds checking, and shared memory coordination is provided, the user is still unprotected against several coding mistakes. The user must insure that all instance sets use the same parameters in the constructor, especially unit size. In general this points to the lack of support for instance group type integrity checking. The user has further responsibility to use the same record type for both the definition of unit_size and parameter for the overloaded read and write operations (ie. t_rec consistency per figure 6). It was hoped some mechanism could have allowed the IHM class to behave as a built in array of struct in terms of access to the struct fields. In the CPPARTS experience it is impossible to overload the dot operator to provide access to fields allowing the IHM to appear as a specialized array.

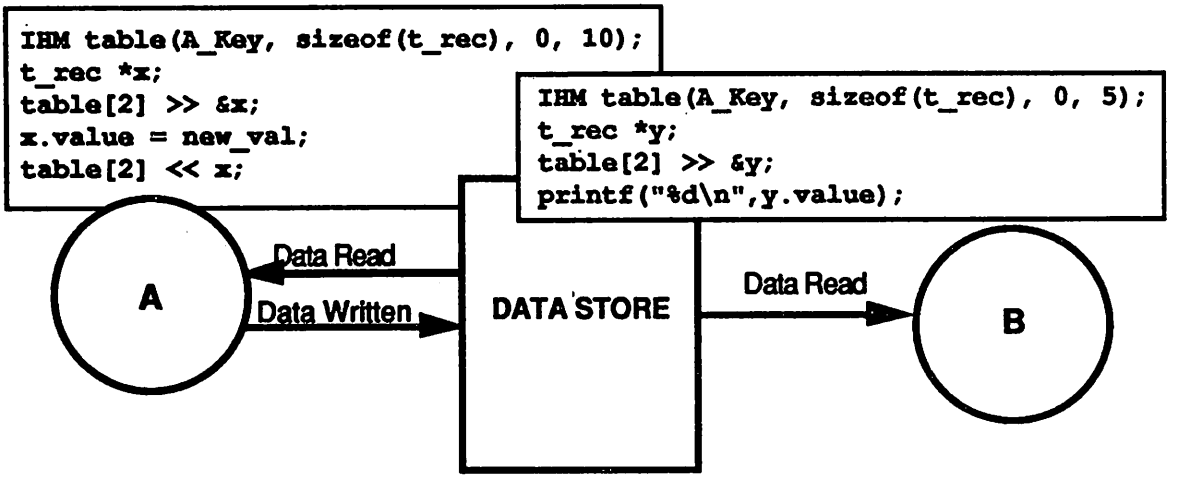


Figure 6: IHM - Information Hiding Module

Message Log

MESSAGE_LOG is a base class for derived classes that require a flexible error logging facility. Messages can be routed to the screen as well as a named file. Each instance can use separate error files providing a means to debug and monitor real-time systems. Within MESSAGE_LOG there are four levels of message severity: red, yellow, green and blue. As a consequence the derived classes can record messages based on the severity level of messages to be recorded in a file or generated on the screen. In addition there is the ability to turn on and off the message generation so that one can concentrate on specific areas. Most UNIX debuggers (SDB, DBX, etc) are not architectured to handle a multi-process environment so subsequently the tracing and error/event monitoring facility is a valuable feature. Additionally, to create the ability to record external interrupts, the message_log constructor builds a routine to catch the SIGINT (interrupt) signal and perform a graceful exit.

Conventions Used By Derived Classes

All the message sending members (red_message(args...), blue_message(args...) etc.) use the first two parameters to indicate the class and member names of the derived class using the facility. The additional two parameters are used to describe the specifics of the message. Flexibility for message format is provided via polymorphic parameterization.

Blue and Grenn Messages

The blue and green messages are used to trace and debug the member functions of all CPPARTS classes and are of little use to a user not in control of CPPARTS source code. Although for those who have read access to CPPARTS source code the messages could be of value.

Derived classes use blue messages to signify entry and exit from their member functions. Thus by setting blue messages to "on" one can trace the internal path of calls through all member functions. A tracing capability was most useful to find the specific location of a run time crash especially in light of the lack of a run time debugger. By using the blue message tracing capability most run time errors can be pinpointed quickly since the user can see the last member function entered.

The green messages are used to record internal formats of successful data translations and traditional debugging statements (eg. a shared memory section begins at location x). This caveat was of immense help for testing and development since it allowed a clean and organized method for retaining debugging statements in the code.

Red and Yellow Messages

Red and yellow messages display diagnostic messages and constraint errors. The yellow messages indicate non-fatal errors such as timeouts or assumed defaults (such as the creation default error file when none is specified by the user).

```
if (( fp = fopen (file, "w")) = = NULL) {
      red_message(class_name,member_name,
                  "Could not open file for writing: exiting",file);
      exit(-1);
}
```

Red messages are used to record a fatal error such as a range error or queue failure. Red messages are typically issued prior to an exit() statement when the process can no longer proceed in the current state. For example when a user program of IHM tries to access a data element out of the shared memory range a red message is issued along with the value of the index.

Choosing Message Routing

Every message has the capability, as controlled per derived class instance, to route messages to the screen and a file. Therefore two types of messages exist: verbose messages and error_log messages. These print to the screen or a file respectively. Operations on these types include set_(verbose/error_log)_on() (level) and activate(verbose/error-log)(). In addition, the user can specify a file per CPPARTS objects that the instance of MESSAGE_LOG or its derived class, will write to via create_error_log_file(char * filename). Through use of the member functions one can tailor the type and location of message recording.

Example:

```
LOOSE_QUEUE queue (A_KEY);  // LOOSE_QUEUE is derived from
                            // BASE_QUEUE which is
                            // derived from MESSAGE_LOG.
queue.set_error_log_level_on(all);  // Blue, green, yellow and
                                    // red are set.
queue.create_error_log_file("LOOSE_QUEUE_A_KEY");// opens the file
queue.activate_error_log();   // activates writting
                              // to file: "LOOSE_QUEUE_A_KEY"
queue.set_verbose_level_on(red);    // Red message are set, but
                                    // not activated.
// some interesting stuff

queue.activate_verbose();   // Now screen logging is activated
queue.send("hello world");  // Red "errors" will go to the screen.
queue.deactivate_verbose();// Here the screen logging of red
                           //messages is turned off
```

Inadequacy of Message Log

Although a variety of polymorphic message formats are provided a more general facility such as that provided by format/ostream would be helpful. The current release of the C++ translator (1.7) had problems with ostream and thus the errors were sent via printf/fprintf. Furthermore the ellipse parameter declaration, forcing the C++ translator to turn off type checking, was not available on this version of cfront. Message_Log was initially created as an on the fly solution to the lack of debugging tools but turned into an invaluable aid in the development of CCPARTS.

Initialization and cleanup

Problems arise in use of the constructor/destructor facility when several processes share a resource that has dynamic criteria for persistence. Persistence is a property of data that determines how long it should be kept[Wegner 87]. If we expand the definition of persistence to apply to objects/classes how then does one express or implement persistence

beyond the scope of the object. Many system resources, such as semaphore, require initialization only once thus the constructor must employ some logic to discover if it is the first to execute, and is subsequently responsible for initialization. Cleanup via the destructor has a similar chore since one cannot assume that the system resource is no longer needed by other instances. Depending on the support provided by the operating system this may or may not be a trivial chore. How does one derive from the system process table what processes are associated with a given object? What is the overhead associated with that type of system interrogation? What about common persistent resources that aren't conveniently tracked by the operating system? If the overhead to acquire this information during execution of the constructor or destructor is high the user of that object might place the definition for that object (eg. sem_object), in the outermost loop of a process to improve performance. Although this can be avoided by declaring the class type static the solution fails for processes containing a high overhead class constuctor that are frequently spawned in a loop and then "die". In this case the outermost loop is in a different, and perhaps separately compiled module and thus the only work-around is to create "initialization modules". The user now is forced to retreat from object oriented expression caused by the need of implementation knowledge. A better solution is to create a section of shared memory to keep a running count of the common processes using the same persistent object. Each process consults this common structure on construction or destruction.

Concurrent programming in C++ requires common memory for resource allocation and deallocation of persistent objects.

This too proves inadequate for a object such as sem_object since the reading and writing of that shared memory cannot be protected via a semaphore (ie.the cart before the horse). Furthermore who is responsible for creating the shared memory? To support the object model in a multi-process programming environment C++ needs a method for specifying object persistence beyond an individual process life. This would require a compiler/linker that understood how to create common memory, which is of course system dependent. Ease of portability could be enhanced for systems using UNIX V.2 that included the AT&T System V Interface Definition. However this support is unlikely since all IPC shared memory operations require special hardware support [Haviland 87].

C++ support for distributed programming

C++ (as is the case for other procedural and object-oriented programming languages) does not contain synchronization primitives needed to address multi-process and concurrent programing requirements. The use special library, or packaged routines that access hardware synchronization facilities, is needed to fill the requirements [Davis 88]. The CCPARTS interface library satisfies the need for synchronization primitives. A relevant question is how well do C++ features support the expression and maintenance of concurrent systems development? Certainly much of the support must be supplied by the operating system [Andrews 83], as is the case for the development, implementation, and testing of CCPARTS. If we take the view that object-oriented programming is programming using inheritance while data-abstraction is programming using user defined types [Stroustrup 87] then the question becomes what C++ features, in terms of those paradigms, improve or discourage the development of concurrent systems.

The class feature is highly useful for the development of the CPPARTS task interfaces since those interfaces naturally fit the object concept: a queue, a data store etc. These objects can be coded into C++ classes and conveniently hide and localize the typically complex operating system interfaces. However, the support for classes weakens in a dynamic multi-process environment since objects cannot easily share data [Laursen 87].

It is impossible to share information between compilations causing the need for global data at run-time. In sequential programming the use of static member variables allows separate class instances to share data. This feature must be simulated in multi-process environments.

C++ does not support static class variables in a multi-process environment

Thus it has been the experience of the CPPARTS project that it is necessary to provide "binding keys" to constructors to allow processes to share common persistent system resources. (ie. shared memory, semaphores, etc.). In some cases shared memory must be created for those instance "groups" to effect arbitration for certain resources (see section on COUPLED_QUEUE).

Since the separate compilation of modules is common in multi-process environments the method of achieving consistency for declarations in different files via including header files provides an adequate solution. If separate processes communicate via operating system resources as opposed to absolute memory locations the need for binding of an entire system (a set of different processes) is reduced. However some type checking must be accomplished at run-time errors to insure instance pair parameter consistency (sections on LOOSE_QUEUE and EVENT provide further depth on instance pair parameter consistency).

C++ does not support instance group parameter consistency.

The above statement is actually a corollary to the following observations. An instance group is a set of class instances with the ability to coordinate operations via shared objects.

C++ enables, but does not support, instance binding.

Instance grouping/binding is enabled through use of binding keys supplied to the constructor, which are made part of the private data for that instance and used to logically group peer class instances. Through use of partitioned key files the user code, or derived class, of a class hierarchy with interdependencies caused by instance binding can be protected from the side effect of unintentional binding.

Summary and Conclusions of CCPARTS value

The DARTS methodology proposes a software development approach that addresses the requirements of real-time systems consisting of several concurrent processes that need to communicate and synchronize their operations [Gomaa 86]. The DARTS methodology proposes the use of well defined interfaces which serve as the basis for the CPPARTS classes. The purpose of the CPPARTS classes is to provide the real-time/concurrent programmer with reliable tools to implement real-time software systems. As the philosophy of object-oriented design boasts the simplistic interface provided via the member functions reduces a programmer's spectrum of required knowledge to the use of the member functions. This is an extreme benefit for real-time programming since many of the concepts, and especially the implementations of those concepts, are complex and system dependent. This particular implementation is compatible with AT&T System V Interface Definition (SVID) generically known as IPC (Inter-Process Communication) facilities. However, systems implementation using CPPARTS task interfaces would enhance portability since the system dependent code for inter-process interfaces would be localized within CPPARTS classes.

References

[Andrews 83]
Andrews, G. and F. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys*, Vol. 15, No. 1, March 1983.

[Davis 88]
Davis, Helen and Hennessy, John, "Characterizing the Synchronization Behavior of Parallel Programs," *Proceedings of the ACM/SIGPLAN PPEALS, Parallel Programming: Experience with Applications, Languages and Systems*, Vol. 23, No. 9, September 1988.

[Gomaa 84]
Gomaa, H., "A Software Development of Real-Time Systems," *Communications of the ACM*, Vol. 27, No. 9, September 1984.

[Gorlen 86]
Gorlen, K., *Object-Oriented Program Support*, Computer Systems Laboratory, Division of Computer Research and Technology, National Institutes of Health, May 1986.

[Haviland 87]
Haviland, K. and B. Salama, *UNIX System Programming*, Addison-Wesley, 1987.

[Laursen 87]
Laursen, J. and T. Merrow, "A Pragmatic System for Shared Persistent Objects," *OOPSLA 87' Conference Proceedings, SIGPLAN NOTICES*, Vol. 22, No. 12, December 1987.

[Stroustrup 86]
Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986

[Stroustrup 87]
Stroustrup, B., *What is "Object-Oriented Programming" ?*, AT&T Bell Laboratories, 1987.

[Wegner 87]
Wegner, Peter, "Dimensions of Object-Based Language Design," *OOPSLA '87 Conference Proceedings, SIGPLAN NOTICES*, Vol. 22, No. 12, December 1987.

A C++ Library for Infinite Precision Floating Point

Jerry Schwarz

AT&T Bell Laboratories
Murray Hill, NJ

The Real library supports infinite precision floating point computation in C++. Arbitrary precision rational arithmetic and transcendental functions are supported.

1 Introduction

The primary class supported by the Real library is `Real`. Arithmetic operators and many math library functions that are defined for floating types (float and double) in C++ are defined for `Reals`. In most contexts all that is necessary to replace ordinary floating point computations by infinite precision computations is to include the header file `real.h` and declare variables to have type `Real` rather than float or double. Values of type `Real` should be thought of as exact real numbers.

This paper's main concern is to describe the library interface. The details of the implementation are not described in this paper. From the users point of view, the important feature of the implementation is that it is based on the idea of "lazy evaluation". A `Real` should be thought of as having a true mathematical *value* represented as a (potentially) infinite binary expansion. This is (perforce) fiction. The truth is that as the computation proceeds, data structures are built representing expressions. In certain circumstances (output, control flow tests, and the like) it is necessary to *expand* these expressions. Leading bits are computed, and the original data structure is replaced by one containing those leading bits, and (a pointer to) another expression representing the remaining (unexpanded) bits. This process may continue indefinitely. A further refinement of this model is that expansions are not really binary. They use a balanced positional notation in which each *bigit* (large binary digit) may be positive or negative. A representation requires such redundancy to be used effectively in infinite precision computations. (See Vuillemin[1] for a proof that redundancy is required.) Balanced positional notation provides this redundancy while ordinary binary representation would not. Despite this representation, I will use phrases such as "binary point" and "bit" when the intended meaning is clear.

The techniques used in this library are related to those of Boehm et. al.[2], Boehm[3] and Vuillemin[1].

Potential uses of this library include the following.

- When there is some part of a computation in which high precision is required. In this case, the program might use `Reals` in some places and floating point types in others.
- When an analyst wants to determine if a program using ordinary floating point is suffering from a numerical instability. Declarations of `Reals` can replace declarations of floating everywhere and the resulting numbers compared with those generated by the floating point computation.
- When an analyst wants to compute some numbers to high precision, either because of intrinsic interest or to serve as reference values for methods using ordinary floating point numbers.
- Algorithms that depend on infinite precision, such as that described in Witten[4].

- For a simple computation where the programmer doesn't want to bother about round off. For example, a program to compute the day of the Jewish holiday of Yum Kippur that is based on certain astronomical equations.

A naive first approach to using the library is simply to replace some `floats` or `doubles` with `Reals`. This may not work for several reasons. Most of this paper is concerned with explaining these reasons. Section 2 discusses the `Plimit` class, which is used to specify limits or tolerances on computations. Sections 3, 4 discuss how `Plimits` are used, and variations of ordinary operations, such as splitting a number into "exponent" and "mantissa", that depend on the representation. Methods for calculating using infinite series are discussed in section 6.

2 Precision Limits

For most operations the program can ignore the precision of intermediate values. However, in some operations a `Plimit` (precision limit) is used to limit the work that might result from an attempt to expand an infinite sequence of bits. For example, in comparing two `Reals` that are equal, but not "obviously" so, the package might keep extending the precision of the difference, looking for a non-zero bit but never finding it. Similarly, the operation that rounds to the nearest integer might run forever when confronted with a `Real` whose value is 1/2, but not obviously so.

Low limits are sometimes appropriate. For example, trig functions typically want to transform their argument by subtracting the nearest multiple of π. However, when the value is close to a multiple of $\pi/2$ no harm is done by subtracting the neighboring multiple of π that is slightly farther away. Thus the rounding is done with a low `Plimit`.

There are two flavors of `Plimit`, absolute (created by the function `abs_plimit`) and relative (created by `rel_plimit`). Absolute and relative limits may be combined with `operator+`. In general, if `p1` and `p2` are `Plimits`, then `p1+p2` is a `Plimit` that forces at least as much computation as either `p1` or `p2` would have. The function `plimit` returns a value that is both an absolute and relative limit. That is, `plimit` has a definition equivalent to

```
Plimit plimit(int n) { return abs_plimit(n)+rel_plimit(n) ; }
```

An absolute `Plimit` requires that a number be expanded to at least a certain absolute precision. For example, when used in a comparison a `Plimit` of `abs_plimit(n)` requires the difference between two `Reals` to be expanded to at least `n` bits after the binary point. In other words, the comparison operation will always report the correct result if the two numbers differ by more than 2^{-n}. Similarly in truncating (rounding towards 0) with `abs_plimit(n)`, the result may be incorrectly rounded away from zero only when it is within 2^{-n} of an integral value.

Relative limits specify that a certain number of bits must be expanded. Unfortunately, this has a precise meaning only in terms of the internal representation of `Reals`. Attempts to describe it based on real values flounder because of the presence of leading zero bits and the possibility of representing a true zero as an infinite sequence of zero bits. However, relative limits may still be useful when the program wants to be sure that at each step of an iteration it is computing with more accuracy than an earlier step. For example, relative limits are used for this purpose in the code implementing `Log2` presented in section 6.

Each function that uses a `Plimit` uses a default value if one is not specified explicitly. There is also a global variable `inf_plimit` which contains a `Plimit` that forces exact results. That is, a calculation that is carried out to this limit either yields a correct result or loops forever.

2.1 Underflow

When a computation is stopped because it has reached a `Plimit` an underflow may be declared. This will happen only when the relevant `Plimit` is at least as large as the value of the variable `underflow_plimit`. Introducing the notion of underflow into the library is a compromise between the idea that exact results should always be produced, and the preference to stop computation when exact results are impossible rather than looping forever.

An underflow also said to occur when the attempt is made to divide by an exact zero. (The library treats an exact zero divisor as if the divisor suffered from an underflow.)

The default action when an underflow occurs is to print a message and call the library function `abort`, which will usually result in a core dump. However, this behavior may be changed by the program. There is a function pointer declared as

```
class Real {
public:
        static void (*uflowaction)(Real& x) ;
        } ;
```

If this variable is non-null, it will be called in place of the default action. When it is called, `x` will be the `Real` whose expansion has been stopped by a limit. If the called function returns, the value of `x` at that point will be used to continue the computation.

3 Other Operations

In addition to the ordinary arithmetic operations, the Real library contains operations that take into account the infinite precision in some way, and those whose computation may be terminated by a `Plimit`.

3.1 Constructors

The `Real` constructors are declared

```
class Real {
        Real(long) ;
        Real(double) ;
        Real() ;
        Real(Series*) ;
        } ;
```

The constructors that take a `long` or `double` argument construct a `Real` with the corresponding value. The constructor without an argument constructs an uninitialized Real, whose value is not defined. The class `Series` is disussed in section 6.

3.2 Splitting

The math library contains a function `frexp` to split a floating value into an "exponent" and a "mantissa". A `Real` may be split using `frexp`, but the pragmatics of infinite precision values suggests some variations that are not relevant for ordinary floating point. These are declared

```
class Real {
public:
        void split(Exp_t& e, Real& r, Plimit t = split_plimit ) ;
        void split(Exp_t& e, long& n, int p,
                                      Plimit t = split_plimit ) ;
} ;
```

These member functions separate the `Real` into an integral(`Exp_t`) "exponent" and a `Real` "mantissa". For the first variant the value of `x` is exactly $2^e r$ and for the second variant the value is approximately $2^e n$. The details of the variants are slightly different.

The first form attempts to assign a value to `r` satisfying $1/2 \leq |r| < 1$. It first expands the `Real` with limit `t`. It next does one of four things

- If the result is an exact zero it assigns zero to `r` and a large negative value to `e`. This is not regarded as an underflow.
- If the expanded values contains only zero bits, but is not yet an exact zero `split` assigns `e` and `r` appropriate values with `r` having a small value (being 0 for several bits after the binary point). This is regarded as an underflow if `t` is at least `underflow_plimit`.
- If it can determine the correct value of `e` and `r` it assigns these values to them.
- In the remaining case, the `Real` is close to a power of 2, $|r|$ may be less than 1/2, but if `t` is at least `rel_plimit(j)`, $|r|$ will never be less than $(1/2) - (1/2)^{-j}$.

Following these rules means that when the computation stops prematurely the split may be wrong, but $2^e r$ has the correct value.

Like the first variant, the second variant of `split` expands the `Real` with the specified limit and then selects an action.

- If the `Real` is or appears to be zero, it assigns zero to `n` and a large negative number to `e`. This is not regarded as an underflow. In some special cases the value may appear to be non-zero but the possibility of zero cannot be completely excluded. In these cases it assigns 1 or -1 to `n` and a large negative number to `e`.
- Otherwise it selects appropriate values for `e` and `n`. With the value assigned to `n` satisfying $2^{p-1} \leq |n| < 2^p$.

Whichever action is selected the value of the original real is always between $(n-1)2^e$ and $(n+1)2^e$. (An attempt is made to avoid the boundary points of this interval, but because of the possibility of an infinite sequence of 1 bits, this cannot always be done.)

For example

```
x.split(e,n,1,abs_plimit(2))
```

can be used to get an order of magnitude approximation to `x`. Since `p` is 1 the value of `n` will be -1, 0 or 1. If `n` is assigned the value 0, the absolute value of `x` is less than 1/4 because `abs_plimit(2)` requires an absolute precision of two bits past the binary point. Using `plimit(2)` instead of `abs_plimit(2)` would have required a certain amount of expansion even for small numbers, and thus would have increased the chance of having a negative value assigned to `e`.

3.3 Rounding

Functions are available for truncating (rounding) reals.

```
enum Round_dir {
        round_down,     // toward zero,
        round_up,       // away from zero,
        round_near,     // toward nearest,
        round_pos,      // toward positive infinity
        round_neg       // toward negative infinity
        } ;
class Real {
public:
        Real round(Round_dir=round_down, Plimit p = round_plimit ) ;
        long iround(Round_dir=round_down, Plimit p = round_plimit ) ;
} ;
```

These functions round **x** to an integer value in the indicated direction. If **p** is at least **abs_plimit(n)** the error in determining the value of **x** will be at most 2^{-n}. For example,

```
x.round(round_down,1) ;
```

Might potentially round a value that is wrong by 1/2. That is, the result might have value n, even if the value of **x** is between n and $n - 1/2$. Similarly,

```
x.round(round_near,2) ;
```

might potentially round a value that is wrong by 1/4. So the difference between the value of **x** and the value of the result may be up to 3/4.

In practice the representation usually does much better than suggested above.

iround returns an ordinary integral(**long**) value. It is possible for the value of the **Real** to be too large to fit, which constitutes an overflow. The results of such an overflow are not defined.

3.4 Conversion Operators

Given a **Real** it is possible to calculate a corresponding **double**.

```
class Real {
        operator double() ;
        double dbl(Plimit p = split_plimit) ;
        } ;
```

If **x** appears to be zero after it is expanded to this using limit **p**, then **x.dbl(p)** will be 0.0. An underflow is possible.

operator double uses limit specified by **split_plimit**. Conversion to an ordinary integral value is supported by **iround**, which is discussed in section 3.3.

3.5 Testing Ranges

Member functions are provided to report whether a **Real** is zero and if not, whether it is positive or negative.

```
class Real {
public:
        int sign(Plimit p = relation_plimit) ;
        int obvious_zero() ;
} ;
```

The value returned by `sign` is 1, 0 or -1 depending on whether `x` is positive, zero or negative respectively. As usual, `p` specifies the amount of computation to do before deciding the value is zero. An underflow is possible.

`sign` has an important side effect. It will normalize the `Real`. That is, as it searches for a leading non-zero bit it will adjust the Real's exponent field.

`x.obvious_zero()` returns true (non-zero) when `x` has a representation that immediately implies it has the value zero. Otherwise it returns false. It never forces any expansion. It is intended primarily for use when computations can be avoided when some value is 0. For example, the code that expands `Series` tests for an obvious zero to allow finite series to be dealt with properly.

The ordinary C++ comparison operations are defined to use certain member functions.

```
class Real {
public:
        int equal(Real y, Plimit p = relation_plimit) ;
        int gt(Real y, Plimit p = relation_plimit) ;
        int ge(Real y, Plimit p = relation_plimit) ;
} ;
```

`x.equal(y,b,a)`, `x.gt(y,b,a)`, and `x.ge(y,b,a)` report whether the value of `x` is equal to, greater than, or at least as great as that of `y` respectively. They work by applying `sign` to `x-y` with limit `p`. Underflow is possible.

3.6 Shifts

The Real library contains shifts of `Reals`

```
Real operator<<(Real& x, long n) ;
Real operator>>(Real& x, long n) ;
```

These return a `Real` whose value is that of `x` multiplied by 2^n or 2^{-n} in the case of `operator<<` or `operator>>` respectively.

4 I/O

An `iostream` inserter(`operator<<`) and extractor(`operator>>`) are defined for class `Real`. The extractor will input an arbitrary number of decimal digits from the input stream. The inserter will output as many digits as are requested by the `precision` format state variable of the `iostream`. Before inserting any decimal digits the inserter attempts to determine an exponent by splitting `Real`(see section 3.2)the `Real` using the `Plimit` contained in the variable `io_plimit`.If the expanded value is an exact zero, the fixed representation, `0E0` is used.

A function `ator` is provided to convert from an ascii string to a real.

```
Real ator(char*) ;
```

The iostream package is available as part of release 2.0 of the AT&T C++ translator.

5 Math library

The following functions from the "math" library are available with the same meanings as the corresponding `double` functions. They produce exact results not approximations.

```
Real acos(Real) ;
Real asin(Real) ;
Real atan(Real) ;
Real atan2(Real,Real) ;
Real cos(Real) ;
Real exp(Real) ;
Real frexp(Real,int*) ;
Real log(Real) ;
Real modf(Real v, Real* ip) ;
Real pow(Real,Exp_t) ;
Real pow(Real,Real) ;
Real sin(Real) ;
Real sqrt(Real) ;
Real tan(Real) ;
```

In some cases I have taken advantage of C++ overloading and references to declare obvious variants

```
Real modf(Real,Real&, Plimit = round_plimit) ;
Real frexp(Real x,int&) ;
Real atan(Real,Real) ;
```

Some special values are also available.

```
Real pi() ;
Real natlogbase() ;
```

`pi` returns the value π while `natlogbase` returns e, the base of natural logarithms.

6 Infinite Series

Infinite series are supported by class `Series`. They are used to implement the transcendental functions of the "math library" and may also be manipulated by user programs. This class is intended to be specialized by derivation in which the derived class specifies terms of the infinite series. The protected interface of the class is declared

```
class Series {
protected:
        virtual         ~Series() ;
        virtual Real    next() ;
        virtual void    step() ;
        virtual Real    bound() ;
} ;
```

These functions must be must be supplied by the derived class. If `s` is a `Series` then these functions must have the following meanings.

`s.next()` is the next term of the series. Repeated calls to `next` must return the same value.

`s.step()` advances the series to the next term. That is, the value that was previously returned by `next` is removed from the series, and future calls to `next` must return a new term.

`s.bound()` is a non-negative value that must be larger than the absolute value of the sum of all remaining terms.

The following function given a `Series s` will return its sum to within `eps`. It will also have the side effect of having removed the terms it as accumulated from `s`.

```
Real sum(Series &s, Real eps)
{
        Real accum = 0 ;
        // use "ge" rather than ">=" on the off chance
        // that s.bound()==eps could occur
        while ( s.bound().ge(eps,rel_plimit(16))) {
                accum += s.next() ;
                s.step() ;
                }
        return accum ;
        }
```

However, the intention is not for programs to use `Series` directly in this way, but for them to incorporate `Series` into `Reals`. The constructor,

```
Real::Real(Series* s) ;
```

constructs a `Real` whose value is the sum of all the terms of `s`. In keeping with the general approach of the Real library, the constructor incorporates `s` into a data structure and returns. The virtual functions described earlier will be called as more precision is required of the `Real`. Because of this method, a particular `Series` should be incorporated in only one `Real`. Once it is incorporated it should not be further manipulated by user code. Also `s` must be the result of a `new` operation rather than a pointer to a value in the stack. This requirement is imposed so that the library can delete it when there are no more references to it.

To illustrate the above, I present the definition of a class used in the Real library.

```
class Log2 : public Series {
        // A class whose value is specifically the
        // natural logarithm of 2.
        Real    m ;
        Real    p ;
        long    k ;
protected:
        Real    next() ;
        Real    bound() ;
        void    step() ;
public:
                Log2() ;
                ~Log2() ;
} ;

Log2::Log2() :
        m( Real(1)/Real(9) ),
        p( Real(1)/Real(3) ),
        k(1)
{
        }

Log2::~Log2() { }
```

```
Real Log2::next()  { return 2*(p/k) ; }
void Log2::step()  { k += 2 ; p = p*m ; }
Log2::bound() { return 4*(p/k) ; }
```

An example of the use of this class:

```
Real x = new Log2 ;
...
```

x has the value

$$2\left(\sum_{k=0}^{\infty}\frac{1}{2k+1}\left(\frac{1}{3}\right)^{2k+1}\right)$$

The library will automatically delete the Log2 object when it is no longer needed.

7 Hints

There are some problems with the AT&T Translator that are exposed by common use of class Real. The package can only been compiled with release 2.0 of the Translator because it relies on the new method for linkage of overloaded functions.

7.1 Logical Connectives

There are problems when the second operand of a logical connective (&& or ||) contains an arithmetic expression using Reals. The translator has difficulty placing the constructor/destructor calls for the conditionally executed expression. Release 2.0 will complain, earlier releases silently generated bad code. Various possible fixes include

- Replace the logical connective, (&& or ||), by the arithmetic one (& or | respectively).
- Rewrite the statement to use a nested if.
- Rewrite the statement to avoid arithmetic in the second operand

All these solutions are ugly workarounds.

7.2 Nesting

For every statement containing an arithmetic expression using Reals, cfront based C++ compilers create a nested block in the intermediate C code. This can cause problems because some C compilers have limits on the number of nested blocks they can deal with. A typical error message is "yacc stack overflow". Note that these errors will come from the underlying C compiler, not cfront.

The workaround for this problem is have smaller blocks. Typically this can be done adding blocks that have no semantic effect.

8 Performance

The performance of the Real library must be measured in two dimensions. The first is how many expansions are performed for a particular computation, and the second is how much time and space are required to perform an expansion.

In the current an implementation a bigit contains 13 bits and it takes approximately 300 microseconds on a VAX 8650 to expand. This is around two (decimal) orders of magnitude slower than the corresponding floating point operation.

In determining overall performance the number of expansions performed is just as important as the time to perform a single expansion. The library implements several strategies for improving performance on this dimension and a significant part of the time spent in expanding bigits goes into avoiding unnecessary expansions. Still, it is harder to measure performance in this dimension, since measurements are so sensitive to the details of the program. For example, to compute x^{256} to two bigits (26 bits) of accuracy requires around 100 expansions when the expression is parenthesized as a balanced tree, but around 20,000 when done by repeated multiplication by x. Improving this number may require some new ideas.

9 Status

The library has been used for some small toy programs.

A version of awk has been compiled that uses `Reals` rather than `doubles` to represent numbers.

It has been used to compute π to 1000 decimal digits.

References

[1] Jean Vuillemin. Exact real computer arithmetic with continued fractions. In *1988 ACM Conference on LISP and Functional Programming*, Salt Lake City, Utah, 1988.

[2] Hans-J. Boehm, Robert Cartwright, Mark Riggle, and Michael J. O'Donnell. Exact real arithmetic: A case study in higher order programming. In *1986 ACM Conference on LISP and Functional Programming*, Cambridge, Mass., 1986.

[3] Hans-J. Boehm. Constructive real interpretation of numerical programs. In *SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, St. Paul, Minnesota, 1987.

[4] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *CACM*, 30(6), June 1987.

Iris: A Class-Based Window Library

E.R. Gansner

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The **Iris** library provides a basis for constructing user interfaces on bitmap displays. Written in C++, it provides a *window* class as the main data type, along with support for such graphical classes as *point*, *rectangle*, *font*, *icon*, etc. Windows can be created within windows; this property, along with the derived class notion in C++, allows the user to design complex interface objects built from simpler components in a modular fashion. In addition, a window can specify which input events, ranging from a change in a specific mouse button to a move request, it will accept. This paper provides an introduction to **Iris** and a discussion of its use.

1. Introduction

The **Iris** library provides a basis for constructing user interfaces for bitmap displays. It implements a flexible hierarchical window system, integrated with the standard bitmap graphics primitives such as bitblt, line drawing and text display. In addition, the **Iris** model naturally supports multiple input and output contexts. The model is object-oriented; the programmer using **Iris** deals with "things" that have a specific set of associated operations. The model, with its emphasis on building an interface from pieces with well-defined interfaces, facilitates the design and construction of user interfaces, and provides a base for building interface toolkits.

The principal design goals of **Iris** were simplicity and ease of use and reuse. For this reason, **Iris** is a small, basic library, meant to be understood. It is based on a simple graphics model, an extension of the one used on the Blit [PI1,PI2] terminal. If desired, the programmer can directly access the window system and the graphics primitives, without intervening software layers adding bias and preventing a simple action from being done simply. System details and hardware incongruities are hidden as much as possible, especially when they exist on a single machine.

Dealing with the pieces of an interface, such as a scroll bar, on an *ad hoc* basis can be cumbersome. The programmer has to deal with how the parts combine and communicate with each other; how input is handled; how objects can be refreshed or reshaped. To get the full value out of an interface tool, it is useful if it can be reused, extended or modified in some natural fashion. **Iris** has adopted an object-oriented approach to address these concerns.

In keeping with the goal of simplicity, the number of basic objects and their associated actions is minimal. The objects are sufficiently general as to not constrain the programmer, while providing a common thread for the later combination of objects. The objects are uniform and reflect common functionality. For example, the bitblt function is bound to a single name that works regardless of whether the source and target bitmaps are on screen or off. Similarly, the process of window creation is the same for all windows.

A corollary of **Iris**'s design goals is that an object-oriented system should use a language that supports object-oriented programming and integrate itself with the language's features and support. Indeed, a significant part of **Iris**'s simplicity derives from its implementation in C++, a language sympathetic to its model. The language's data abstraction facilities supports **Iris**'s notion of a graphical object. Class inheritance, especially from multiple base classes, combines with the **Iris** window hierarchy to provide a simple yet powerful mechanism for creating new objects from old. The virtual function polymorphism gives the library a means of implementing the generic window operations in a uniform, type-safe manner for all graphical objects. A related benefit is that **Iris** avoids the mess of names and types, and the type insecurity, that plague systems implementing an object paradigm without

language support.

Another aspect of the ease of use of a window system involves the run-time environment of the program, and the burdens it places on the programmer and the program. **Iris** is designed to operate in a standard UNIX† environment, not in a rarefied graphics environment. There are no assumptions that, for the graphics part, there are actions a program would not or should not take. The familiar set of tools, libraries and facilities for constructing and analyzing programs are all available for the interface code.

2. The Iris Model

Iris was designed to provide a foundation for constructing user interfaces from high-level components while allowing access, when necessary, to low-level graphics primitives. In this, it follows a bitmap graphics analogue to a design consideration of C++ for general programming. The C++ language does not include such constructs as run-time typing or a list class, but it allows them to be built. Similarly, **Iris** provides a collection of simple, generic constructs for bitmap graphics. The programmer can extend this base as desired with the creation of additional graphical classes.

The idea of a *graphical object* (or *widgets*, in X terminology [SA]) is central to the **Iris** model. The programmer should view a graphical object as a physical construct, with limited, well-defined interactions with the rest of the world. A graphical object might closely mimic "real" components, such as meters, switches, buttons and oscilloscopes; provide a graphics abstraction of "real" components, such as a scroll bar or a virtual terminal; or instantiate an object with no physical analogue. New components can be made by tuning the parameters of extant objects, or building a new object out of one or more available objects. In this model, the construction of an interface involves gathering an appropriate collection of graphical objects and specifying how they interact, within the constraints of an object's interface. This model provides an attractive base for using high-level languages and interactive editors in the specification of interfaces.

This view of an object as a physical reality is one of the paradigms for object-oriented programming. In the **Iris** context, this paradigm works particularly well. First, the programmer can actually see the objects. Second, since **Iris** objects are built using a language that supports object-oriented programming, the manipulation and derivation of graphical objects as objects comes naturally.

The basic object supplied by **Iris** is the *window*. (In other systems, a similar object might be called a *viewport*, a *canvas*, a *layer*, or a *pad*.) It provides a surface for all graphical output and a context for input. Lines can be drawn, text written, areas filled in a window, among other primitive graphics operations. A window can specify its interest in certain input events, and provide handlers to deal with the events when they occur. The coordinate system used to specify graphics output and report user input are window coordinates, i.e., the origin occurs in the upper left-hand corner of the window, with increasing y values moving the point down, and increasing x values moving the point to the right. Section 3 gives additional detail concerning the attributes and functionality of **Iris** windows.

Iris windows form a base class in C++: they are meant to be used to derive higher level graphical objects. As is usual in C++, some derivations may simply add functionality to the base class, allowing the base class to show through. Other derived objects will totally hide its base class behind a layer of abstraction or replace some base class functions with its own. Typical **Iris** objects formed in this way include scroll bars, buttons, switches, banners, virtual terminals, text editors, list viewers, tile editors and dialogue boxes.

In addition to being created as derived classes, new objects can come about through the **Iris** window hierarchy. Basically, one **Iris** object can include a variety of other **Iris** objects as children or subwindows. This technique allows an object to use the well-defined services of other objects to enhance its interface. A canonical example of this technique is a virtual terminal object, which contains three

† Registered trademark of AT&T.

children: a text editor object, for displaying character output and receiving keyboard input; a banner object, for displaying status information and providing certain window services; and a scroll bar, for allowing the user to move the text view. In turn, some of these objects may be further subdivided. For example, the scroll bar might consist of a slider object and two arrow button objects, while the banner might contain buttons for deleting or resizing the window. Figure 1 show some graphical objects built using various combinations of C++ subclasses and the Iris hierarchy. The objects include a sketch pad, a tile editor and a virtual terminal.

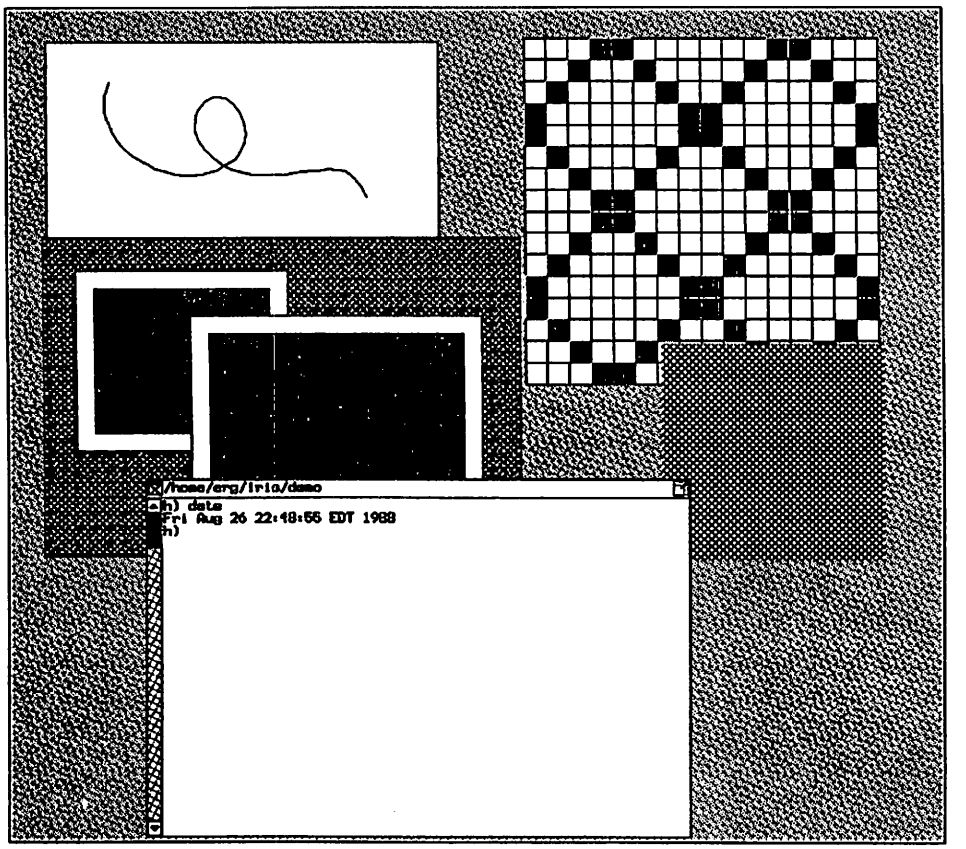


Figure 1. Some graphical objects.

A parent object's role can be passive, in which it creates the children, hooks them together and gets out of the way, except for the occasional special service. In particular, a parent is usually responsible for repositioning its children when it is resized. At the other extreme, a parent object might take an active role, continually providing specific services or managing its children and their communications.

Iris is initialized by creating the root window object corresponding to the display. Additional objects can then be created, optionally specifying a parent and a location in the parent's coordinates. Only Iris objects that are descendents of the root object will be visible. A child object need not intersect with its parent, but only that portion of the child which overlaps with the parent is visible. A child object always lies on top of its parent; a child cannot be moved behind its parent. Sibling objects, however, can be freely shuffled among themselves. In addition, invisible Iris objects can be created; if a window is invisible, all of its descendents are invisible.

Children of the root window, which we shall refer to as *base windows*, maintain a backing store, so that even if they are obscured by a sibling window, the obscured portions are stored in off-screen

bitmaps. This allows **Iris** to handle automatically occasions when part of the window becomes exposed, without requiring the object to assist in the refresh. In addition, these objects can always act as the source of a bitblt operation without additional support from the object itself. The availability of backing store greatly accelerates most screen refreshes, and simplifies the programmer's job when creating new objects.

Though any window can have a backing store, a non-base window usually shares its parent's bitmap. In many cases, base windows will be tiled by their children, essentially providing the benefits of a backing store to the children without additional memory overhead. If a window can become corrupted, it is responsible for redrawing the corrupted portions by supplying a refresh function.

Iris maintains the integrity of its windows, in that output in a given window will always be clipped to that window (and its parent) and will not affect any sibling windows or their descendents, even if the window is obscured by some siblings. In general, output in a window can only corrupt a window's ancestors and descendents.

When the destructor operator is applied to a window, it calls the destructor of all the window's descendents in depth-first order, frees the window's resources and initiates the refreshing of the display. Note that the display is refreshed only once, after the entire subtree is freed. This avoids the visual unpleasantness and time used to update windows that are themselves slated to be destroyed.

Control flow in **Iris** is, by default, *event-driven*. Objects specify the types of events they are willing to handle, and supply a handler for such events. When an event occurs, such as a mouse button click or a timer event, **Iris** takes the event, determines who should get the event and passes it to the object using the object's event handler. In most cases, the object handles the event as expeditiously as possible and returns control back to **Iris**. In this way, the user is presented with an interface built of multiple active contexts cooperating with each other.

There are times when an object decides, for efficiency or for clarity of code, that pure event-driven control is inappropriate. **Iris** allows a program to forego the built-in scheduling entirely, to use it only periodically or to "grab" the event stream for certain critical periods. The **Iris** library itself makes use of this in its implementation of pop-up menus and other high-level input routines.

An object's event processing in **Iris** has an all-or-nothing flavor: accept one event at a time or temporarily take control of all events. This can sometimes constrain the programmer's coding style, forcing object state information to be stored explicitly in the object when it would normally be stored implicitly in the stack frame . In addition, events targeted for other objects living in the same process can be delayed in delivery. The solutions offered by other interface libraries to similar problems are not totally satisfactory. The correct answer is probably to incorporate some multiple thread mechanism, as is done in NeWS [GO].

There are at present several general methods that could be used to support multiple threads in **Iris**. Specifically, one could use either the C++ task library [SH] or the C++ version of the Concurrent C language [GR]. We are currently pursuing a more radical approach (cf. [RG]).

2.1 Related Systems

The interface programmer has a wealth of window systems, libraries and toolkits (cf. [EV,GO,LC,PA,RW,SA,SG], to mention a few) from which to choose. In terms of graphics functionality and networking, **Iris** is not the most developed. However, the two aspects that particularly distinguish **Iris** from similar work are its simplicity and its reliance on C++.

Compared to **Iris**, the models of many window systems are more complex, some to the point of incomprehensibility. To do something simple might require much head scratching and programming overhead. Some packages insist the programmer deal with several layers of system tasks before actually being able to draw something in a window. This complexity can lead to a lack of uniformity, in which the programmer must do different things at different times to achieve a similar effect. At other times, the software layers impose restrictions preventing certain things from being done. Concerning graphics systems, the word sophisticated can sometimes be used in a pejorative sense.

Iris is written in and for C++. Other interface packages, especially toolkits, are object-oriented in flavor, but provide little language support for maintaining the object paradigm. For this reason, they are cumbersome and limiting for the programmer, who must emulate or provide explicitly what would come for free from a compiler. Without language support, the names and types tend to become cluttered, and any use of inheritance will not benefit from type-checking. Also, transliterating a library's header files into C++ is not the same as designing and writing the library using C++.

3. Basic Classes

Any bitmap graphics system needs some notion of points and rectangles. Thus, **Iris** provides the following structures:

```
struct Ipt {
        short x;
        short y;
};
struct Irect {
        Ipt origin;
        Ipt corner;
};
```

The *Ipt* structure stores the x and y coordinates of a point. The origin in *Irect* specifies the upper left corner of the rectangle. The corner specifies the point one below and to the right of the bottom right point in the rectangle. Thus, the width and height of the rectangle are given by *(corner.x - origin.x)* and *(corner.y - origin.y)*, respectively.

A pleasant bonus of using the C++ language is the ability to have operator arithmetic with points and rectangles. Instead of calling a function to add two points or to translate a rectangle, one sees the following, more aesthetic code:

```
Ipt   pt1, pt2, pt3;
Irect r1, r2;
    .
    .
pt3 = pt1 + pt2;            // Add two points.
r2 = r1 + pt3;              // Translate r1 by the vector pt3.
    .
    .
```

3.1 The Bitmap Class

The class *Ibitmap* provides the target for graphical operations. An *Ibitmap* corresponds to a bitmap, along with a set of output member functions, These include the usual graphics operations such as marking a point, tiling a rectangle, drawing a line, writing text, and bitblting from one bitmap to another. All graphics operations are modified by a boolean function corresponding to how the bits are to be combined. In addition, the operations are clipped to the intersection of the bitmap's boundary and its associated clipping rectangle. The geometric parameters passed to graphics routines are always in the bitmap's coordinate system. This is the only convention that makes sense for the programmer, especially in an object-oriented, multi-window environment.

Each **Iris** bitmap has associated with it various attributes. Most attributes have both a *get* function, for querying the attribute value, and a *set* function for setting the value, except in certain cases where the attribute is read-only. For example, each bitmap has a clipping rectangle to limit the effect of graphics operations, but is only used when clipping is activated. **Iris** provides the *Set_clip_rect()* function for setting the clipping rectangle, and the *Set_clip()* function for turning clipping on and off. In addition, each bitmap possesses a current point. This point is used as the initial point in line drawing and text operations. When drawing lines, **Iris** resets the current point to the final point supplied; when writing text, **Iris** resets the current point to the position where the next character would be written.

3.2 The Window Class

The fundamental **Iris** class is the window object *Iws*. This provides the base class for most of the graphical objects using the **Iris** library. In essence, a window is a subclass of both *Ibitmap* and a tree class. This provides a window's output functionality and the parent-child relationship between windows. In addition, the window provides a context for user input.

The window class adds to the functionality it inherits from the bitmap class with attributes and functions for handling the window hierarchy and input. The functions *Top()* and *Bottom()* move a window in front of and below all of its sibling windows, respectively. The *Move()* function moves the window so that its new origin corresponds to the argument point, given in the parent's coordinate system. The *Resize()* function reshapes the window to a given rectangle, again in the parent window's coordinate system.

An **Iris** window can dynamically specify which input events it is interested in receiving. (**Iris** input is discussed below.) Other attributes associated with windows include cursor shape and position, various sizes related to the window, the window's parent and whether the window is obscured.

3.3 Getting User Input

Iris events are divided into several categories: keyboard events, mouse events, window events and system events. Mouse events include not only button clicks and mouse motion, but also the action of the mouse entering or leaving a window. Window events are high-level events connected to requests to change the external state of a window. System events involve timeouts and file I/O. All events are received by a window through an appropriate virtual function. The use of separate handlers for each type of event, as well as for refreshing, simplifies event handling somewhat.

Keyboard events occur when a key is depressed; there are no provisions for handling up transitions. Keyboard events are not totally raw, in that the depression of metakeys (*control*, *shift*, etc.) are not reported when they occur, but only in the context of the depression of another key. A class derived from *Iws* must supply a virtual function *Keybd_fn* to receive these events.

All mouse events are received through a virtual function *Mouse_fn*. The arguments specify the type of mouse event that occurred and the point at which the event occurred, in window coordinates, as usual. **Iris** provides other functions to query the current mouse state, buttons and position, at any time. For efficiency and consistency, it is recommended that when tracking the mouse, an object should poll the mouse's position rather than rely on a stream of mouse motion events.

With window events, a window is informed that someone would like to move, reshape, shuffle or delete it. This gives the object an opportunity to perform various cleanup operations. For example, an edit window about to be deleted might wish to give the user the opportunity to write its contents into a file. In addition, the window can use the return value to indicate its acceptance or rejection of the proposed action.

The final category of events is system events. These include timeout events, in which a window asks to be notified after a certain time period; file events, corresponding to the need to service some open file; and requests for notification of the deletion of some object.

Certain common input methods are supplied as member functions of **Iris** windows. The *Get_pt()* routine allows the user to pick a point within the window, using a specified mouse button. The routine can be set to return immediately when a button is pressed or to wait for the button to be released. The *Get_rect()* routine allows the user to specify a rectangle within the window, using a given button. The user moves the cursor to some position, corresponding to one corner of the rectangle, depresses the button, moves to another position corresponding to the opposing corner and releases the button. The current rectangle shape is indicated by rubber banded lines. The *Move_rect()* routine allows the user to place a given sized rectangle within the window, again using a given button. If the user depresses one of the unspecified buttons, all of these routines return immediately. By convention, this indicates the user's desire not to make a choice and, by extension, not to invoke any related action. Another useful input operation, *Track()*, allows an object to loop on the state of a given mouse button while receiving

the current mouse position.

Iris also provides a pop-up menu class *Imenu*, with semantics roughly modeled on the menu library of the Blit. A menu is created by supplying the constructor with a NULL-terminated array of strings. There are general facilities for inserting, removing and replacing menu items. To display the menu, its *Use()* member function is invoked, supplied with a mouse button. As long as the button is down, the user can move the mouse cursor over the menu items, each highlighted when it is under the cursor. When the button is released, the index of the current item is returned, with 0 being the index of the first item. If no item is selected, -1 is returned.

The library uses basic Iris functions to implement all of the high-level input routines described above. They are provided as part of the library because of their frequent use, especially in providing the interface designer with basic input tools for prototype software. It is a simple matter to add alternative input methods and menu systems in the Iris environment.

3.4 Damage Control

All window objects should supply a *Refresh_fn* virtual function. This routine will be called by Iris whenever it needs help restoring a window. This may occur if the window does not have an associated backing store, or if the window's size has been changed, or if a child window is externally altered. One argument to *Refresh_fn* specifies the type of restoration necessary, in particular indicating whether the window has been resized. This alerts the window to reset any size dependent parameters it uses and, if necessary, to reshape its children. After being reshaped, each child will have its own *Refresh_fn* called. Another argument specifies the region to be repaired.

3.5 Additional Objects

Other useful Iris classes include *Itile*, *Icursor* and *Ifont*. An *Itile* represents a pattern of bits that can be used to fill a closed polygonal region. An *Icursor* is an icon that can be used to represent the mouse location on the display. An *Icursor* can be associated with any Iris window and is typically used to represent the current state of the window. If a window does not choose its own cursor, it inherits the one used by its parent.

The *Itile* and *Icursor* are derived from the *Ibitmap* class, and can therefore be used and altered with bitmap member functions. Normally, though, the programmer will use bitmap constructors that take a character-based representation of the bitmap or the name of a file containing such a representation.

The Iris font class represents a font used for writing text. The user creates an *Ifont* by specifying the pathname of the file containing the font's representation. Once a font exists, the user can ascertain the maximum height and width of the characters in the font, as well as other information. There is also a function for determining the dimensions of a string if it were printed using the specified font. Iris provides a default font *Idefont*.

4. Using Iris

The structure of a program using Iris can be very simple: create a global object, create some initial graphical objects and turn control over to Iris. Using Iris, the traditional program to print "Hello, world" looks like:

```
#include <Iris.h>

main ()
{
    Iws     *globalWin = new Iws ();
    Irect   globalRect = globalWin->Get_rectangle ();
    Iws     *win = new Iws (0, globalWin, globalRect>>10);

    win->Text (Ipt(2,2), "Hello, world", Idefont, Istore_op);

    Iris_loop ();
}
```

This creates the default global window, determines its rectangle and creates a base window inset by 10 pixels from the global window. The string is then written in the base window, translated slightly from the window's origin. Finally, the program passes control of the program over to **Iris**. The routine *Iris_loop()* collects events and dispatches them to the appropriate objects.

Of course, a more typical program would use a variety of graphical objects, and establish a more complex initial scene before calling *Iris_loop()*. Frequently, the global window itself is some derived class. Figure 2 illustrates a prototype browser for trees built using **Iris**. It uses some ten different subclasses of *Iws*.

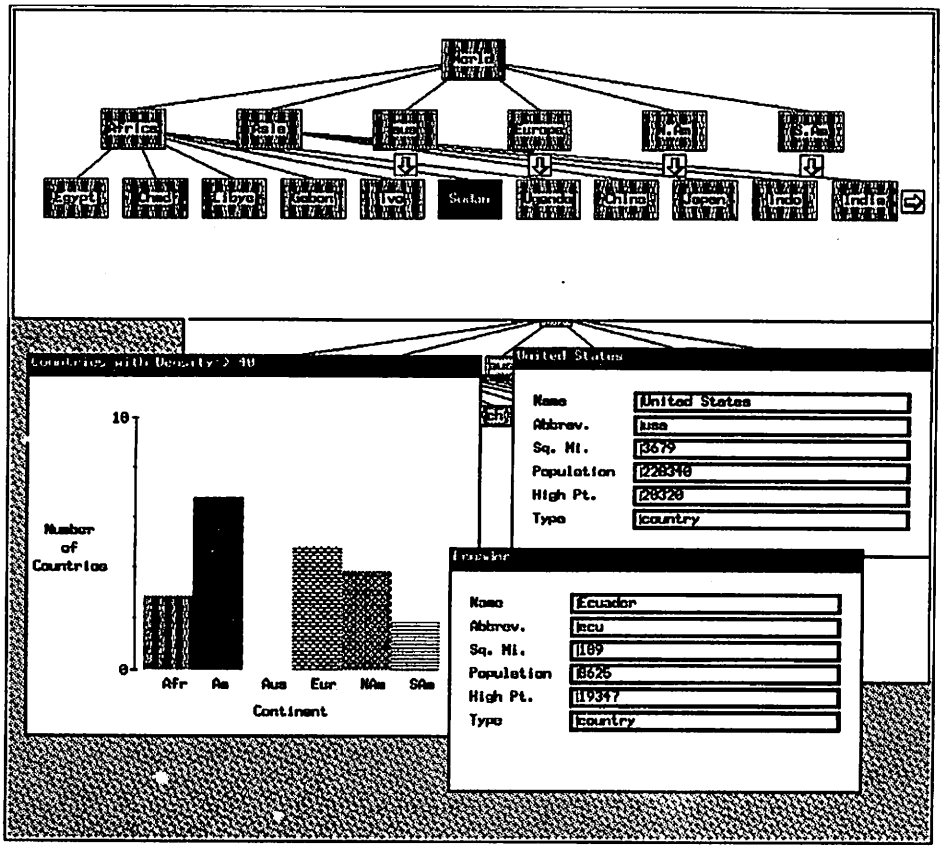


Figure 2. A tree browser.

There are various nuances in creating graphical objects using Iris. The code above exhibits a standard way of composing one object out of others using the window hierarchy. This technique applies well for a container class that knows what it "looks like" and creates its own children, a top-down approach. It is also possible to create child objects first, essentially unparented, and later insert them in a container class. Creating new objects using C++ is done with the standard derivation syntax:

```
class FormField : private Iws { ... };
```

But there are times when the programmer may find it advantageous to use a reference instead:

```
class FormField { Iws *win; ... };
```

with less binding between classes.

For certain applications, an object may need to have more flexibility than is provided by *Iris_loop()*. In these cases, the program can read directly from the stream of Iris input events using the function *Iget_event()*. This is the routine used by *Iris_loop()* when it distributes events.

5. Nuts and Bolts

Iris has been used as the interface library for a variety of programs, including a C source analyzer, an economics analysis package and a programming environment generator. It has also been used as the foundation for the windowing system in the Pegasus environment [RG]. At present, Iris runs on AT&T 5620 or 630 terminals, paired with hosts running the UNIX operating system, and on various UNIX-based workstations. It consists of about 4000 lines of C++ code, with an extra 4000 lines of C code necessary for the terminal server when used with the 5620 or 630.

5.1 Maintaining Windows

A major design decision in most window systems is how to maintain window integrity, especially if a window can be written on when partially obscured. Iris uses an extension of the *layerop()* method [PI]. The Blit uses this method to provide a single level of windows with backing store. The method is elegant, lends itself easily to most primitive bitmap graphics operations, and, in some sense, uses the minimum cost to maintain a backing store. In addition, it can be naturally extended to support the general hierarchy and semantics of Iris windows.

5.2 Iris and C++

Iris was designed with C++ in mind. It reflects the syntax and semantics of the language and, as explained in Section 1, attempts to use language features to the fullest to support its model. In particular, inheritance in C++ complements the Iris hierarchy in the creation of new objects. With this mixture of types, Iris depends critically on the virtual functions in C++ in order to cleanly manipulate objects as windows.

The data hiding provided by classes and member functions has been a help for writing portable Iris-based programs. Specifically, Iris has been implemented using a library model on workstations while using a client-server model with the terminals. Yet, program source can be moved unchanged between machine types.

The biggest disadvantage in using C++ has been its mutability. In attempting to remain current with the language design and fixes to the translator, one found that legal code one week might be illegal the next, and maybe legal again the week after. In addition, as useful new features appeared, it had to be decided if, when and how they could be incorporated into the design.

6. Conclusions

The Iris library presents a flexible and simple model for constructing bitmap graphics interfaces. The model is general enough so that many graphics interfaces could be built using Iris. This generality is tempered by the object-oriented nature of Iris, especially as it is supported by C++. This nature encourages the construction of interfaces using "graphical objects", components whose reuse, extension and combination are facilitated by the C++ class and Iris window hierarchies. In this context, Iris

provides a good example of the desirability of building object-based systems using a language that supports an object paradigm. Finally, we note that Iris blends well with the standard UNIX tools and techniques.

Acknowledgments

The author would like to acknowledge his debt to Jonathan Shopiro, who was a pioneer in the use of graphical objects in C++ and whose ideas provided a strong impetus for Iris. The comments and complaints of Fahim Jalili, Tom Kirk, Dave Korn, Minsky Luo and John Reppy, among others, were very helpful in providing additional direction for Iris.

REFERENCES

[EV] Evans, Steve, "The Notifier", *Proc. USENIX Tech. Conf.*, Atlanta, GA, June 9-13, 1986, pp. 344-354.

[GO] Gosling, James, "SunDew - A Distributed and Extensible Window System", **Methodology of Window Management**, F. R. A. Hopgood et al., eds., Springer-Verlag, Berlin, 1986, pp. 47-57.

[GR] Gehani, N.H. and W.D. Roome, "Concurrent C", *Software-Practice and Experience*, 16(9), Sept. 1986, pp. 821-844.

[LC] Linton, Mark A. and Paul R. Calder, "The Design and Implementation of InterViews", *Proc. USENIX C++ Workshop*, Santa Fe, New Mexico, 1987, pp. 256-267.

[PA] Palay, Andrew et al., "The Andrew Toolkit - An Overview", *Proc. USENIX Tech. Conf.*, Dallas, TX, February 9-12, 1988, pp. 9-21.

[PI1] Pike, R., "Graphics in Overlapping Bitmap Layers", *ACM Trans. Graphics*, 2(2), 1983, pp. 135-160.

[PI2] Pike, R., "The Blit: A Multiplexed Graphics Terminal", *AT&T Bell Laboratories Tech. Journal*, 63(8), 1984, pp. 1607-1631.

[RG] Reppy, J.H. and E.R. Gansner, "A Foundation for Programming Environments", *Proc. 2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, Palo Alto, CA., 1986, pp. 218-227.

[RW] Rao, Ram, and Smokey Wallace, "The X Toolkit: The Standard Toolkit for X Version 11", *Proc. USENIX Tech. Conf.*, Phoenix, AZ, June 8-12, 1987, pp. 117-129.

[SA] Swick, Ralph R. and Mark S. Ackerman, "The X Toolkit: More Bricks for Building User-Interfaces or Widgets for Hire", *Proc. USENIX Tech. Conf.*, Dallas, TX, February 9-12, 1988, pp. 221-228.

[SG] Scheifler, R.W. and J. Gettys, "The X Window System", *ACM Trans. Graphics*, 5(2), April, 1986, pp. 89-97.

[SH] Shopiro, J., "Extending the C++ Task System for Real-Time Control", *Proc. USENIX C++ Workshop*, Santa Fe, NM, Nov. 9-10, 1987, pp. 77-94.

Lexical Closures for C++

Thomas M. Breuel *

Abstract

We describe an extension of the C++ programming language that allows the nesting of function definitions and provides lexical closures with dynamic lifetime.

Our primary motivation for this extension is that it allows the programmer to define iterators for collection classes simply as member functions. Such iterators take function pointers or closures as arguments; providing lexical closures lets one express state (e.g. accumulators) naturally and easily. This technique is commonplace in programming languages like Scheme, T, or Smalltalk-80, and is probably the most concise and natural way to provide generic iteration constructs in object oriented programming languages. The ability to nest function definitions also encourages a modular programming style.

We would like to extend the C++ language in this way without introducing new data types for closures and without affecting the efficiency of programs that do not use the feature. In order to achieve this, we propose that when a closure is created, a short segment of code is generated that loads the static chain pointer and jumps to the body of the function. A closure is a pointer to this short segment of code. This trick allows us to treat a closure the same way as a pointer to an ordinary C++ function that does not reference any non-local, non-global variables.

We discuss issues of consistency with existing scoping rules, syntax, allocation strategies, portability, and efficiency.

1 What we would like to add... and why

We would like to be able to nest function definitions in C++ programs. In this section, we will discuss a number of reasons why the ability to nest function definitions is desirable. Almost all other modern programming languages such as Scheme, T, Smalltalk-80, Common Lisp, Pascal, Modula-2, and Ada offer this feature.

To illustrate the utility of this language feature, we have to agree on syntax. To indicate that a function is defined inside another function, we will simply move its definition inside that function but otherwise write it the same way we would at global level. For example, the following fragment of code defines `function2(int)` inside `function1(int)`:

```
function1(int x) {
    // ...
    function2(int y) {
        // ...
    }
    // ...
}
```

*Author's address: MIT Artificial Intelligence Laboratory, Room 711, 545 Technology Square, Cambridge, MA 02139, USA. The author was supported by a fellowship from the Fairchild foundation.

Unless `function2` declares the identifier `x` itself, any mention of `x` inside `function2` will refer to the argument of `function1`.

To allow the definition of mutually recursive functions at inner lexical levels, it is necessary to provide some way of declaring functions that are not defined at global level. We suggest that this is done by prefixing the function declaration by the keyword `auto`. Currently, it is illegal to use the keyword auto before a function declaration. This extension is therefore compatible.

1.1 Nesting and Modularity

The ability to nest function definitions encourages modular design of programs. It allows the programmer to keep functions and variables that are only used by one function local to that function. In C, such modularity is only possible at the level of files: the scope of the identifier of a function can be limited to a file by declaring it `static`. The scope of a variable's identifier that is to be shared among functions must encompass at least a compilation unit since it must be declared at global level. C++ supports limiting the scope of certain kinds of identifiers for functions and variables to member functions by making those functions and variables members of a class.

However, it is often not natural to introduce a new class simply to limit the scope of an identifier. Consider, for example, the heapsort algorithm[Wir79]. It consists of a `sort` function that repeatedly calls a `sift` function to insert elements into a heap. In C or C++, we would express this as follows:

```
sift(int* v,int x,int l,int r) {
   // insert x into the heap
   // formed by elements l...r of v
}

// sort an array of integers v
// v is n elements long

heapsort(int* v,int n) {
   // code that calls sift
}
```

This is unsatisfactory, however, because the function `sift` is unlikely to be of any use anywhere else in the program. The function `sift` ought to be visible only inside the function `heapsort`. Furthermore, we would like to refer to the variable `v` inside `sift` without having to pass it as a parameter. Nesting allows us to rewrite this as:

```
// sort an array of integers v
// v is n elements long

heapsort(int* v,int n) {
   sift(int x,int l,int r) {
       // insert x into the heap
       // formed by elements l...r of v
   }
   // code that calls sift
}
```

Notice in particular that any use of the identifier v inside the function `sift` refers to the argument v of the lexically enclosing function `heapsort`[1].

As another example, assume that we have a function `integrate` that integrates a given function between two bounds and we would like to integrate a parameterized family of functions. Again, the most natural way of expressing this is as follows:

```
integrate_all(int n,double* as,double *bs,double *cs,double eps) {
    double integrate(double low,double high,
                     double epsilon,double (*f)(double));
    double a,b,c;
    double f(double x) {
        return a*x*x+b*x+c;
    }
    for(int i=0;i<n;i++) {
        a=as[i]; b=bs[i]; c=cs[i];
        printf("a: %g, b: %g, c: %g, i: %g\n",
               a,b,c,integrate(0.0,1.0,eps,f));
    }
}
```

1.2 Iterators

The ability to nest function definitions and to reference variables declared by enclosing blocks is particularly useful together with iterators over collection classes. Consider the following simple collection class:

```
class BagOfInts {
public:
   // add an int to the collection
   void add(int);

   // test whether an int
   // is in the collection
   int member(int);

   // apply a function to every
   // element in the collection
   void walk(void (*)(int));
};
```

For example, to print all the elements in a bag, we would write in standard C++:

```
printit(int x) {
   printf("integer in bag: %d\n",x);
}

main() {
   BagOfInts aBag;
```

[1]unless `sift` declares or defines another identifier v

```
    ...
    aBag.walk(printit);
}
```

Note that we are forced to define the function `printit` far away from the place where it is actually used. The reason why we made `printit` a function in the first place is not that it is a useful abstraction of some process or that we are going to use it in several places, but simply because the member function `walk` demands a function pointer as its argument.

Even more disturbing is that in standard C++ the only side effects a function that is passed as an argument to the iterator can have are to static variables or global variables. If we would like to use the iterator `walk` to sum all the elements in a bag, we would have to use a global variable:

```
int xxx_counter;

int xxx_count(int x) {
    xxx_counter+=x;
}

fizzle() {
    BagOfInts aBag;
    ...
    xxx_counter=0;
    aBag.walk(xxx_count);
    int sum=xxx_counter;
    ...
}
```

If we were allowed to nest function definitions, we could express this simply as:

```
fizzle() {
    BagOfInts aBag;
    ...
    int sum=0;
    count(int x) {
        sum+=x;
    }
    aBag.walk(count);
    ...
}
```

Now, all the identifiers are declared and defined exactly where they are used. No identifiers appear at global scope that should not be visible at global scope[2]. And, no global data space is wasted for the counter; the space for the counter exists only while function `fizzle` is active.

This approach to writing iterators for user-defined classes is actually very commonly used in Smalltalk and Lisp-like languages. For example, it Smalltalk-80[GR83], we would express the function `fizzle` as follows:

[2]It is still disturbing that we had to invent a name for the function count. We will suggest syntax to define unnamed functions in later sections. We used the named version here in order to avoid getting into questions of syntax at this point.

```
fizzle
   | sum |
   ...
   sum <- 0.
   aBag do: [x | sum <- sum + x].
   ...
```

And in CommonLisp[Ste84] we might write:

```
(define fizzle ()
   (let ((aBag ...)
         (sum 0))
      ...
      (map nil #'(lambda (x) (incf sum x)))
      ...))
```

The reader might ask whether there are alternative approaches to iteration just using standard C++ constructs. We will describe one of them shortly. Let us first remind ourselves, though, what we ask of an iteration construct in any language.

An iterator is basically a construct that takes a piece of code and invokes it repeatedly and changes the values of some bindings in the environment of that piece of code. From the built-in iterators of C and C++ we are used to being able to do the following.

1. We can write down the piece of code that is to be executed repeatedly *at the point* in the source code where the iteration is performed.

2. The code that is executed repeatedly can reference identifiers whose scope is limited to the enclosing function.

3. The names of variables modified by the iteration construct can be chosen freely.

These properties are very useful features of built-in iterators, and it would be most unfortunate to have to give up any one of them for user-defined operators.

One way to achieve these goals it to introduce a new class together with a macro, as follows:

```
class BagOfIntsStepper {
public:
   int more();
   int next();
};

#define iterateBagOfInts(bag,var) \
   for(BagOfIntsStepper stepper=bag.makeStepper(); \
       stepper.more(); \
       var=stepper.next())
```

This construct does not violate any of the above constraints. However, it has several disadvantages. It requires us to introduce a new class for the purpose of expressing iteration, it requires two calls to member functions per iteration step, and it requires us to define a macro. Most of all, however, we dislike about it that it is difficult to extend to the case when

we want to iterate over different collection classes with the same base type[3]. For example, we might also have a class `SetOfInts`. If we used the `walk` style iteration construct, we could simply make `BagOfInts` and `SetOfInts` subclasses of a class `CollectionOfInts` and declare `walk` to be virtual. To achieve the same effect with a stepper class is much less straightforward and might also require the extra overhead of two virtual function call for each iteration step.

2 How to Implement It

In order to implement nesting and lexical closures in C++, we have to introduce a static link chain that links each activation record to the correct activation record for the lexically enclosing function (see [AU79] and [Wir77] for terminology). When we invoke a function, we not only have to know its address, but we also have to pass along a pointer to the correct activation record for the lexically enclosing function. There are two exceptions to this rule, however.

No space for a static link pointer needs to be reserved in the activation record of a function defined at global level because its lexical environment is known to be the global environment and because it is clear at compile time whether an identifier refers to a global variable.

Furthermore, a function defined at global level does not need to be passed a pointer to the lexically enclosing environment, because the lexically enclosing environment is the global environment which is unique and has a known address.

These two exceptions together with the fact that C++ allows functions to be defined only at global level make it possible to implement lexical scoping in C++ without using a display, a static link chain, or passing around pointers to environments.

Adding space for the static link chain to the activation record of functions that are not defined at the global level is trivial (they can be thought of simply as additional automatic variables). Since the compiler always knows which absolute lexical level an activation record corresponds to, it is not a problem that the activation record for functions at the outermost lexical level differs from that of activation records of functions at other lexical levels.

What does present a significant problem is the fact that instead of just a pointer to code, the invocation of a function at an inner lexical level in addition requires a pointer to the proper activation record of the lexically enclosing function.

If we want to be able to pass around pointers to functions at inner lexical levels freely, this information must be passed around together with the pointer to the code of the function, since this information cannot be derived in any other way.

As long as we are not using functions as arguments to other functions or assign functions to variables, the compiler can silently take care of making sure that the information about the proper lexical environment is passed along to the callee. However, as soon as we try to pass around functions as data, we encounter a problem. In C++, only enough space for one pointer needs to be reserved and passed around when defining and using a function pointer. However, a closure, i.e. a function with an environment requires two pointers in general, as we have just seen.

Changing the representation of a C++ function pointer to be two pointers large rather than one pointer is unacceptable. This would mean, for example, that a "function pointer" could not be assigned to a variable of type `void*` without losing information.

[3]The base type of a collection is the type of the elements of the collection.

An alternative might be to introduce a new data type, "closure", that must be used to express references to functions and their environment if those functions are not defined at global level. Using implicit type conversions from function pointers to closures would allow us to mix function pointers could be used in place of closures. However, closures could not be passed to existing functions that expect function pointers. Furthermore, the language becomes unnecessarily cluttered by two data types for essentially the same concept.

Clearly, neither of these alternatives is acceptable. Fortunately, there is a simple and efficient solution. In fact, our solution is completely compatible with C++ and C. In particular:

1. our closures can be used anywhere a C or C++ function pointer can be used, even if the code using the closure was compiled with a compiler that does not know about closures

2. the code generated for functions that do not contain nested functions does not change when closures are added to the compiler

The code segment that generates a closure does so by generating a short segment of code that loads the static link pointer into some known register and jumps to the function (lines 0043-0046 in the listing below). At the beginning of a function, the contents of the register is moved into the static link field of the newly created activation record (line 0076). In an assembly language similar to 68000 assembly language[4], this would look something like:

```
0000  ;;; source code:
0001  ;;;
0002  ;;; function1() {
0003  ;;;      function2() {
0004  ;;;                ...
0005  ;;;      }
0006  ;;;      void (*x)();
0007  ;;;      x=function2;
0008  ;;;      ...
0009  ;;; }
0010
0011  ;;; machine registers:
0012  ;;; FP: current frame
0013  ;;; SL: a register that is used to hold the static link pointer
0014  ;;;      temporarily
0015
0016  ;;; instruction to load the SL register with constant
0017  INST_LOADSL      equ 0x77777777
0018
0019  ;;; instruction for direct jump
0020  INST_JUMP        equ 0x88888888
0021
```

[4]Assume that all data and instructions are 32bits wide. Temporary labels begin with a "$". The instruction moveea moves the effective address of its first operand into a location. The link instruction fills in the "$dl" field of the activation record.

```
0022  function1:
0023          ;;; allocate space for the following variables:
0024          ;;; void* $dl -- dynamic link chain
0025          ;;; void* $sl -- static link chain
0026          ;;; int $stub[4] -- space for machine code for closure
0027          ;;; int (*x)() -- a function pointer
0028          ;;; also sets up the dynamic link chain
0029
0030  $dl       equ -4
0031  $sl       equ -8
0032  $stub     equ -24
0033  function1.x       equ -28
0034
0035          link FP,28
0036
0037          ;;; set up the static link chain
0038
0039          move SL,$sl(FP)
0040
0041          ;;; create closure for function2
0042
0043          move #INST_LOADSL,$stub(FP)
0044          move FP,$stub+4(FP)
0045          move #INST_JUMP,$stub+8(FP)
0046          moveea function2,$stub+12(FP)
0047
0048          ;;; x=function2
0049
0050          moveea stub(FP),function1.x(FP)
0051          ...
0052
0053          ;;; code using x
0054
0055          move function1.x(FP),R1
0056          call (R1)
0057          ...
0058
0059          ;;; finish up
0060
0061          unlink FP
0062          return
0063
0064  function2:
0065          ;;; function entry, as above
0066
0067          link 4
0068
0069  $dl     equ -4
```

```
0070  $sl     equ -8
0071
0072          ;;; set up the static link chain
0073          ;;; register SL was set up by the stub
0074          ;;; code in function1
0075
0076          move SL,$sl
0077          ...
0078
0079          unlink FP
0080          return
```

The lifetime of the code array `$stub` is identical to the lifetime of the activation record of `function1`. This is reasonable, since the static link chain implicitely defined by `$stub` becomes invalid as soon as `function1` exits[5].

This is the basic idea. There are several compile-time optimizations possible, some of which we have already mentioned. If `function1` is defined at global level, for example, it does not need to reserve space for a static link chain in its activation record. If `function2` does not reference any variables in the activation record of `function1`, the compiler can leave the activation record of `function1` out of the static link chain handed to `function2`. If `function2` does not reference any non-local, non-global variables, its definition can effectively be moved to the global level (except for the scope of its name), and neither a static link field nor a code stub needs to be generated inside `function1`. If `function2` is only used within `function1` and no closure is passed around, the code generating the stub code can be eliminated since the compiler can generate code to load register SL inline just before calls to `function2`[6].

3 How Efficient is It?

To see how efficient this scheme is, we have to compare it with alternative implementations. The two most straightforward implementations are to represent a closure as either a structure with two elements, a pointer to code and a pointer to the environment, or as a pointer to such a structure.

The additional space required by our scheme consists of only the two machine instructions contained in the stub code, and the two instructions inside `function1` used to generate the two machine instructions in the stub code. Assigning and passing closures is as efficient as in the case where we use a pointer to a structure.

However, the overhead of closure creation is comparatively unimportant. As we have seen above, the compiler has to create a closure only if a function pointer is to be passed around. In our experience, a function pointer that is passed around is usually used repeatedly, so the overhead of invoking a closure is much more important than the overhead of creating one.

Let us look at the instruction sequences that are executed in each of the three different implementations of closures. First, here is the instruction sequence for closures that are

[5] We will later discuss possible extensions to extend the lifetime of a particular static link chain beyond the dynamic lifetime of the component activation records.

[6] In fact, a restricted form of nesting where we disallow taking the address of functions defined at an inner lexical levels can be implemented without extending the compiler to generate code stubs.

represented directly as structures[7]. The closure consists of two machine words (pointers) at offset x in the current activation record [8]:

```
move x(FR),SL
move x+4(FR),R1
call (R1)
```

In the case of a pointer to a closure, the calling sequence becomes a bit more complex. Assume that px is the offset of the pointer to the closure in the current activation record[9]:

```
move px(FR),R1
move (R1),SL
move 4(R1),R1
call (R1)
```

For our proposal, the sequence of instructions encountered is:

```
move px(FR),R1
call (R1)
move #staticLink,SL
jmp function
```

We see that the instruction sequences executed when a function pointer to a function requiring a static chain pointer is called are not too different. The major differences in efficiency will probably come from cache and instruction prefetch effects. Some simple experimentation indicates that on the 80386 a call/return pair that passes a static link pointer along takes about 1.5 times as long as a simple indirect or direct function call. If we use our proposed instruction sequence, this figure is increased to about 1.8 times.

4 Issues of Portability

To implement our proposal, it is necessary for a piece of code to be able to generate short pieces of code at runtime and execute them; the generated code does not necessarily have to be placed on the stack, however.

This is possible and straightforward on most modern computer architectures like the 68000 series of microprocessors[Mot80], the VAX[Dig81b], and the 80386[Int87]. Even on the PDP-11 series processors with separate instruction and data space ("I and D spaces"), the stack is ordinarily mapped into both I and D space to permit execution of instructions on the stack (in the standard PDP-11 instruction calling sequence, the `MARK` instruction is executed off the stack, see the PDP-11 processor handbook[Dig81a]).

There are, however, some architectures and/or operating systems that forbid a program to generate and execute code at runtime. We consider this restriction arbitrary[10] and consider it poor hardware or software design. Implementations of programming languages

[7]These are not assembly language program fragments but traces of the assembly language instructions executed during invocation of a closure.

[8]R1 is some general purpose register.

[9]The contents of locations (R1) and 4(R1) in the following example could conceivably be cached.

[10]Such systems usually provide operating system calls to move data into the instruction space, for example for the benefit of a loader; however, the overhead of an operating system call is too high for the creation of a closure.

such as FORTH, Lisp, or Smalltalk can benefit significantly from the ability to generate or modify code quickly at runtime.

We can use another trick to implement lexical closures even on these architectures. We pre-allocate in instruction space an array of instruction sequence of the form:

```
stub_n    move location_n,R1
          move (R1),SL
          move location_n+4,R1
          jmp (R1)
```

We use this array as a stack to allocate and deallocate closure stubs. A corresponding array of locations in data space holds the actual pointers to the code and the static link chain of the closures. These two new stacks behave essentially like the runtime stack. In particular, `longjmp` must be modified to restore the two stack pointers for the stub stack and the location stack appropriately.

5 Further Extensions

We observed in some of the above examples that often there is no need to name a function explicitly. This is particularly so when we use iterators that take function pointers as arguments. The examples of Lisp and Smalltalk code given above involve unnamed functions. We suggest to express an unnamed function as a cast of a *compound-statement* to a function pointer. For example, the value of the following *expression* is a function pointer or a closure (depending on the context):

```
(int (*)(int x)){ return x+1; }
```

Alternatively, we could introduce a new keyword, unnamed, and write the same construct as

```
(int unnamed(int x){ return x+1; })
```

We prefer the first form slightly, but the second form may be easier to parse and allow better syntax error detection and recovery.

The way we have proposed to implement closures limits their lifetime to that of the activation record in which they were created. In order to make closures a data type of the same standing as any other data type in C++, it should be possible to allocate and deallocate closures. There are good reasons to provide closures that can be treated as data structures. Abelson and Sussman[AS85] argue strongly for this feature, and in the Scheme programming language[RC86] they are often used to build complex data types. However, classes and structures provide much of the functionality of heap-allocated closures.

6 Conclusions

The ability to nest function definitions and to create lexical closures with at least dynamic lifetime is an important part of many modern programming styles. Most modern programming languages provide it, and it can be incorporated into the C++ (and C) programming language without affecting the efficiency of execution or meaning of programs that do not take advantage of the feature. We would therefore like to see nested function definitions

and lexical closures to be incorporated into the C++ language definition. We are currently working on extending the GNU C[Sta88a][Sta88b] and C++[Tie88] compilers to provide nested function definitions and closures. The availability of a good, free compiler with source level debugger should encourage more people to use this feature.

Acknowledgements

I would like to thank Richard M. Stallman, Robert S. Thau, and many others who have made useful comments on the proposal and the paper.

References

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1979.

[Dig81a] Digital Equipment Corporation. *PDP-11 Processor Handbook*, 1981.

[Dig81b] Digital Equipment Corporation. *VAX Architecture Handbook*, 1981.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Int87] Intel Corporation. *80386 Programmer's Reference Manual*, 1987.

[Mot80] Motorola Semiconductors Products, Inc. *MC68000 16-bit Microprocessor User's Manual*, 1980.

[RC86] Jonathan Rees and William Clinger. The Revised^3 Report on the Algorithmic Language Scheme. Technical Report 848a, MIT Artificial Intelligence Laboratory, September 1986.

[Sta88a] Richard M. Stallman. *Internals of GNU CC*, April 1988.

[Sta88b] Richard M. Stallman. *The GNU Debugger for GNU C++ Free Software*, April 1988.

[Ste84] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[Tie88] Michael D. Tiemann. *User's Guide to GNU C++*, May 1988.

[Wir77] Niklaus Wirth. *Compilerbau.* B. G. Teubner, Stuttgart, 1977.

[Wir79] Niklaus Wirth. *Algorithmen und Datenstrukturen.* B. G. Teubner, Stuttgart, 1979.

Pointers to Class Members in C++

S. B. Lippman
B. Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper discusses the syntax, semantics, implementation and use of pointers to members. Pointers to members are necessary for the completeness of the C++ type system and relate directly to key design and implementation issues. This paper shows how the design and implementation of pointers to members is determined by the strategy for object layout, by the implementation of virtual functions, and by the need for an efficient function call mechanism.

1. Introduction

The introduction of a type-safe construct for expressing pointers to class members cleared up one of the remaining holes in C++[1]. Prior to this, there was no implementation-independent way of declaring and using pointers to class member functions. In practice, the programmer would hand translate a class member function into its internal non-member representation. That is, the programmer was forced to trick the type system.

The trick involved making the assumption that a member function was an ordinary C function with an added first argument representing the `this` pointer (as explained and warned against on page 154 of reference [2]). This assumption had to be "wired into" the user's code with casts to override the type system. For example:

```
class X {
public:
        void f( int );
};

typedef void (*pf)(...);        // bad style

f() {
        pf fake = (pf)&X::f;    // assume a lot
        X a;
        (*fake)(&a,2);          // fake call
}
```

As with most uses of casts, this obscures what is going on; the compiler (and often the human reader) cannot spot errors. Worse, by considering this code legal, the compiler is constrained to generate working code for it – and this will in many cases mean sub-optimal code. In particular, this code would not work were a C++ compiler to take advantage of the special status of `this` in member functions in order to generate a calling sequence that puts `this` into a register. Another weakness of this style of "faking" function calls is that it does not cleanly extend to cope with virtual member functions.

This paper discusses what is needed in C++ to ensure that a programmer will never have to make this kind of unsafe assumption, write code this ugly, or force sub-optimal implementation strategies on compilers. Naturally, the solution enables the usual pointers to function techniques to be used for pointers to member functions without breaking the type system, having to write spurious code, or sacrificing run-time efficiency or space.

2. A Very Simple Object Model

Consider a perfectly abstract and general model of an object, such as might be used for a C++ implementation designed to minimize the complexity of the compiler at the expense of space at run-time. Consider this class:

```
class X {
public:
        int i;
        char *pc;
        double d;
        static si;
        void mf( int );
        virtual void vf( int );
};
```

An object is a sequence of slots; each slot points to a member. The elements appear in order. There is an element for each data or function member.

In this simple model we do not put the members themselves in the object, only pointers to the members. This is to avoid problems stemming from the fact that the members are of quite different types and require different amounts (and sometimes different types of) storage. The first slot is left empty so that we can represent a pointer to no member as a plain zero.

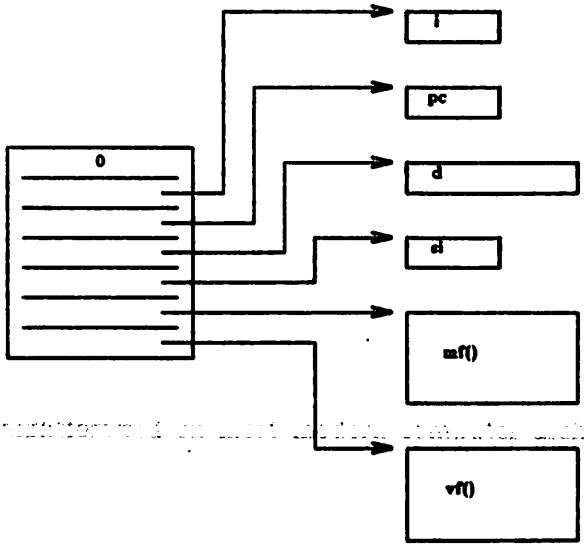


Figure 1. Simple Object Model

We can now address a member within an object by its index. For example `i`'s index is `1` and `mf`'s index is `5`. Given such an index and an object we can apply the operation indicated by the index on the object (ignoring type problems for the moment).

This simple concept of a "index" or "slot number" is the one that has been developed into the C++ pointer to member concept. In essence, everything else is simply syntax and implementation details: A pointer to member is a representation of a position in an object.

3. The C++ Object Model

The C++ object model is derived from this model by optimizing for space and access time. Data members are allocated directly in the object so that a pointer to a data member can be represented simply as the member's location in the object, that is its offset in bytes from the start of the object:

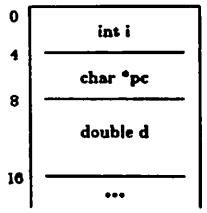


Figure 2. The C++ Object Model

The representation of a pointer to `X::i` will be something akin to `0`; the representation of `X::d` will be something akin to `8`. We need to preserve `0` as meaning "pointing to no member". This can be accomplished either by "wasting" a byte of storage at the beginning of each object or by subtracting `1` at each access. Since there are typically many objects and few member pointer accesses, the latter technique is used, so that a pointer to `X::i` is represented by `1` and a pointer to `X::d` is represented by `9`.

Instead of placing virtual functions in each object, we place a single pointer (the virtual table pointer, or "vptr") to a table of pointers to virtual functions in each object (the virtual table, or "vtbl"). Such virtual function tables can be shared by all objects of the same class:

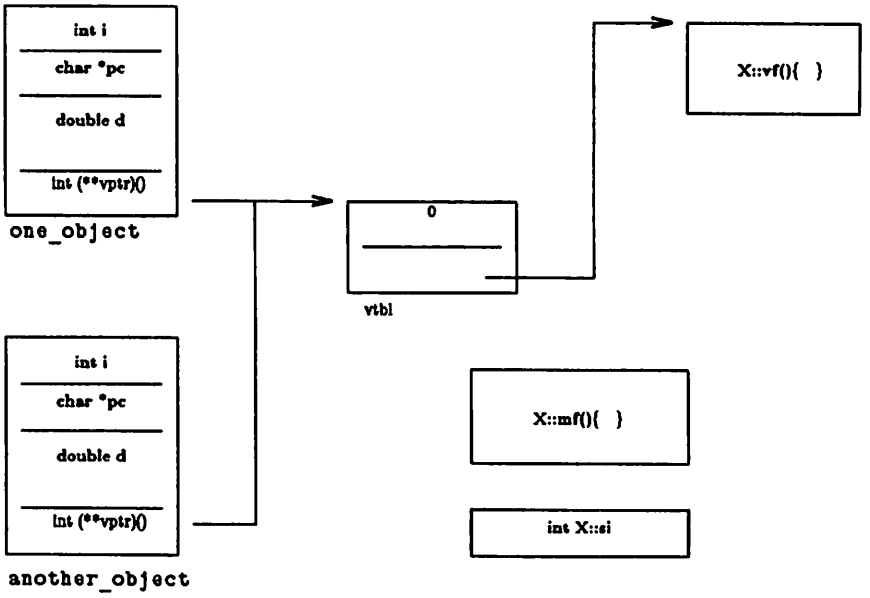


Figure 3. The C++ Object Model with Virtual Table

A pointer to `X::vf` should therefore be something akin to its index in the vtbl. Note that the first entry in the vtbl is left unused to allow `0` to represent "not pointing to any virtual function".

Pointers to non-virtual member functions are not allocated in the object at all; nor are static data members.

Other optimizations of the general object model could be used for C++ implementations. The general model itself might even be ideal for some purposes. Naturally, the implementation details for pointers to members will differ from implementation to implementation. We find it useful, though, to base our discussion on a concrete example: The object model used for the cfront (a.k.a the AT&T C++ Translator) implementations of C++.

4. Pointers to Class Members: Type and Syntax

Consider

```
class X {
public:
        int i;
};
```

The complete type of `X::i` is "`int` member of class `X`". Consequently the complete type of a pointer to `X::i` must be "pointer to `int` member of class `X`". In C++ this type is written as follows:

```
int X::*
```

A declaration of a pointer to member of class X of type `int` looks like this:

```
int X::*pmX1;
```

and can be initialized like this:

```
int X::* pmX1 = &X::i;
```

The notation for pointer to member parallels the *class-name::member-name* notation used for explicit qualification of class members and appears minimal for the amount of information it needs to convey. In the C++ syntax *class-name::** is a component of a declarator exactly like *. Here is a simplified grammar showing pointers to members:

```
declarator:
        * declarator
        & declarator
        class-name :: * declarator
        declarator [ expression ]
```

and similarly for abstract declarators:

```
abstract-declarator:
        empty
        * abstract-declarator
        class-name :: * abstract-declarator
        abstract-declarator [ expression ]
```

Let us consider a larger example in detail:

```
class X {
public:
        int i;
        char *pc;
        const ci;
        double d;
        static si;
        void foo( int );
        void bar( double );
        int foobar( int );
        virtual void vf( int );
};
```

```
int i;
class Y { public: int i; };

int X::*pmXi = &X::i; // ok: legal initialization

// each of the following assignments is a type violation
// pmXi is of type int X::*

pmXi = &i;      // error!!  &i is of type int *
pmXi = &Y::i;   // error!!  &Y::i is of type int Y::*
pmXi = &X::d;   // error!!  &X::d is of type double X::*
pmXi = &X::ci;  // error!!  &X::ci is of type const int X::*
pmXi = &X::pc;  // error!!  &X::pc is of type char X::*
pmXi = &X::si;  // error!!  &X::si is of type int * (!! see Section 7.1)
```

All these errors are caught at compile time.

4.1 Pointers to Member Functions: Declaration

The most common form of a pointer to member is a pointer to member function. The type resulting from taking the address of the member function `void X::foo(int)` is that of a pointer to a member function of class X that takes one argument of type `int` and has a return value of type `void`; that is,

```
void (X::*)(int)
```

Declarations of a pointer object with this type can look like this,

```
void (X::*pmfXVi1)( int ) = &X::vf;
void (X::*pmfXVi2)( int ) = &X::foo;

// ...

pmfXVi1 = pmfXVi2;
```

The following assignments to `pmfXVi1` all result in compile-time type violations:

```
extern void foo( int );
class Y { public: void foo( int ); };

// each of the following assignments is a type violation
// pmfXVi1 is of type void (X::*)(int)

pmfXVi1 = &foo;         // error!! foo is of type void (*)(int)
pmfXVi1 = &Y::foo;      // error!! Y::foo is of type void (Y::*)(int)
pmfXVi1 = &X::bar;      // error!! X::bar is of type void (X::*)(double)
pmfXVi1 = &X::foobar;   // error!! X::foobar is of type int (X::*)(int)
```

Like pointers to functions, pointers to member functions are often declared as arguments to functions. For example,

```
extern int f( int (X::*)() );
```

declares a non-member function `f` to take one argument of type pointer to member function of X taking no arguments and with a return value of type `int`.

Declarations of pointers to "ordinary" functions are often hard to read; as one would expect, declarations of pointers to member functions are no better. As usual, this problem can be ameliorated by using `typedef` declarations.

```
typedef int (X::*pmfi_X)();
extern int f( pmfi_X );
```

Pointer to member function arguments may take default argument initializers. For example,

```
class Example {
public:
        void mf_1() { cout << "Example::mf_1 called"; }
        void mf_2() { cout << "Example::mf_2 called"; }
};


typedef void (Example::*PMFv_Example)();
PMFv_Example pmf = &Example::mf_1;

extern void f( PMFv_Example = &Example::mf_1 );

f();                    // Example::mf_1 called (default argument used)
f( pmf );               // Example::mf_1 called
f( &Example::mf_2 );    // Example::mf_2 called
```

Class member function pointers may be class members themselves. For example,

```
class Screen {
public:
        enum { WIDTH, HEIGHT };
        int (Screen::*pmf)();

        getWidth()  { return width; }
        getHeight() { return height; }
        const Screen& operator=(const Screen&);
        void pmfSet(int=WIDTH);
        // ...
};


void Screen::pmfSet( int i ) {
        switch ( i )
        {
        case WIDTH:
                pmf = &Screen::getWidth;
                break;

        case HEIGHT:
                pmf = &Screen::getHeight;
                break;

        // ...
        }
}

MyScreen.pmfSet();                      // MyScreen.pmf = &Screen::getWidth;
MyScreen.pmfSet(Screen::HEIGHT);        // MyScreen.pmf = &Screen::getHeight;
```

5. Pointers to Class Members: Function Call

A member function must always be invoked for a specific object of its class. Consequently, to invoke a member function through a pointer to member, an object must also be supplied. To accomplish this, two new operators (`.*` and `->*`) have been introduced. For example,

```
int (Screen::*pmfi)() = Screen::getHeight;
const Screen& (Screen::*pmfv)(const Screen&) = &Screen::operator=;


Screen MyScreen, *BufScreen;


// direct invocation of member function
if ( MyScreen.getHeight() == BufScreen->getHeight() ) *BufScreen = MyScreen;


// same invocation through pointers to members
if ( (MyScreen.*pmfi)() == (BufScreen->*pmfi)() ) (BufScreen->*pmfv)(MyScreen);
```

The calls `(MyScreen.*pmfi)()` and `(BufScreen->*pmfi)()` require the parentheses because the precedence of the call operator `()` is higher than the precedence of the `.*` operator. Without the parentheses `MyScreen.*pmfi()` would be interpreted to mean: `MyScreen.*(pmfi())`. The `.*` and `->*` operators have a precedence higher than binary `*` but lower than `.` and `->`:

```
...
->      .
->*     .*
*       /       %       .
+       -
>>      <<
...
```

Like most C++ operators, `.*` and `->*` are left associative. Both `.*` and `->*` are treated as single tokens.

Within a member function, the class object that has invoked the member function is available through the explicit use of the `this` pointer. For example,

```
class Y {
public:
        typedef int (Y::*PMFi_Y)();

        // explicit use of this pointer
        int mf_f( PMFi_Y pmf )  { return (this->*pmf)(); }
        int mf_ff()             { return ++i; }
        Y()                     { i = mf_f( Y::mf_ff); }

private:
        static int i;
}


Y a;                    // Y::i = 1;
main() { Y z; }         // Y::i = 2;
```

A pointer to member function passed as an argument to a non-member function can bind its call to a class object in a number of ways:

```
// Example is defined on page 6
Example global;

void f(Example& obj, PMFv_Example p = &Example::mf_1)
{
        Example local;
        Example& ff();

        (global.*p)();  // call through a global class object
        (local.*p)();   // call through a local class object
        (obj.*p)();     // call through class object passed as reference
        (ff().*p)();    // call through a returned class object
}
```

Names of pointers to class member of course obey the usual constraints of information hiding. For example,

```
class X;
typedef int (X::*pmf_X)();

class X
{
friend  Foo2();
public:
        mf_pub();
protected:
        mf_prot();
};

pmf_X pp = &X::mf_prot;         // illegal: X::mf_prot is protected!

Foo2()
{
        pmf_X pp = &X::mf_prot; // ok within the friend member scope
};
```

6. Pointers to Class Members: Data Member Access

A data member of a class must always be accessed for a specific object of its class. This is also true of a pointer to a class data member. Consider the following class:

```
class X {
public:
        X();
        X(X& x, int X::* p);
        operator int X::*();
protected:
        int i;
        double d;
};
```

The second constructor for `X` take a second argument of type pointer to a data member of class `X` of type `int`. For purposes of illustration, it is defined as follows:

```
// x.*p ==> yields the value of x.i
X::X(X& x, int X::* p) : i( x.*p )
{
        // this->*p ==> yields the value of this->i
        d = double(this->*p);
}
```

Conversion operators for pointers to class members may be defined. As an illustration, `X` provides a conversion operator for `int X::*`.

```
// !!not: ( int X::* )this->i;
X::operator int X::*() { return &X::i; }
```

Let's see how these may be used:

```
X a;
int X::* pi_X = a;      // ok: X::operator int X::*() called for a

X b = X(a, pi_X);
X *px = new X(a, b);    // ok: X::operator int X::*() called for b
```

Here are some examples of accessing a data member through `pi_X`.

```
f() {
        a.*pi_X = 1024;
        b.*pi_X = a.*pi_X;

        *px.*pi_X = a.*pi_X + b.*pi_X; // evaluated as (*px).*pi_X
        a.*pi_X = px->*pi_X * b.*pi_X; // evaluated as (px->*pi_X) * (b.*pi_X)
        a.*pi_X = *px.*b; // evaluated as ((*px).*)b.operator int X::*()
}
```

Pointers to class members may be used to access a member of a specific object, cast (see Section 9) or, if they are of the same pointer to class member type, compared using the equality and inequality operators. For example,

```
ff()
{
        double X::* pd_X = ( double X::* ) pi_X;
        if ( pd_X != (double X::*)pi_X )
                ;
}
```

7. Implementation Details

Class membership comes in four flavors: data members, static data members, non-virtual member functions and virtual member functions. We have seen how they are represented in memory (see Figure 3). In this section, we will look at how the program text might be transformed to reflect the internal class representation. (A highly portable method of code translation is the generation of C. This is reflected in our examples.)

7.1 Static Class Members

Only one instance of each static member exists for a given class. A pointer to a static member is treated outside the pointer to member class syntax. Consider:

```
class X {
public:
        static si;
};

int *pi = &X::si;
```

Applying the address-of operator, `&`, to `X::si` yields a value of type `int*` and not `int X::*`. This directly reflects the underlying implementation and allows optimal code generation, but it does except pointers to static members from the usual rules for pointers to class members. One reason for this "abnormality" is that it would allow the concept of a static member to be extended to include static function members. A static function member would be a non-member function within the lexical scope of a class. Such functions could be quite useful.

7.2 Class Member Functions

The following is a simplified definition of a general Screen class:

```
class Screen {
public:
// cursor functions
        void move(int x,int y) { cursor = screen + ( x*width + y-1; }
        void home() { move(0,0); }
private:
        short width;
        short height;
        char *screen;
        char *cursor;
};

Screen MyScreen;
```

How is code generated for the member functions of Screen? A generally portable internal translation of Screen::home and Screen::move might look like this,

```
move__Screen( this, x, y )      /* C code */
register struct Screen *this;
int x, y;
{
        this->cursor = this->screen + x*this->width + y;
}

home__Screen( this )
struct Screen *this;
{
        move__Screen( this, 0, 0);
}
```

Each member function is translated by the compiler into an equivalent uniquely named non-member function. Membership is an association the compiler maintains between a class type and its member functions. One link in this association is a class pointer argument the compiler adds to the argument list of each of its translated member functions. The name given to the argument is `this`.

The `this` pointer provides the binding of the the member function and the object for which it is called. The `this` pointer holds the object's address. To achieve this, the compiler translates each member function call. For example,

```
MyScreen.home();
```

becomes

```
home__Screen( &MyScreen ),      /* C code */
```

while

```
Screen *TmpScreen = &MyScreen;
TmpScreen->move( x, y );
```

becomes

```
struct Screen *TmpScreen = &MyScreen;   /* C code */
move__Screen( TmpScreen, x, y );
```

7.3 Virtual Functions under Single Inheritance

Consider this simple inheritance graph,

```
class ZooAnimal {
public:
        void locate();
        virtual void draw();
        virtual void debug( int );
        virtual isOnLoan();
        // ...
};

class Bear : public ZooAnimal {
public:
        void draw();
        void debug( int );
        virtual Date& hibernates();
        // ...
};
```

ZooAnimal defines three virtual functions, `draw()`, `debug` and `isOnLoan()`. Bear, derived from ZooAnimal, will supply its own instance of `draw()` and `debug()` but wishes to share ZooAnimal's `isOnLoan()` function. In addition, Bear introduces the virtual function `hibernates()`.

The pointer to the virtual table is present in the base class (see Figure 3) and in inherited by subsequent derivations. The address of each virtual function is stored at a fixed index into the table. (The function `draw()`, for example, is always the first entry, `debug()` the second, and so on.) If a derived class defines its own instance of the virtual function, such as Bear's instance of `draw()`, the address of that instance is entered into the virtual table; otherwise, by default, the address of the base class instance is entered.

Each call of a virtual function is translated by the compiler. The virtual function name serves as an index into the virtual table. (The function `draw()`, for example, indexes table slot 1, `debug()` table slot 2, and so on.) The internal function call is the indirect invocation of the function whose address is present in the slot at run time. For example,

```
for ( ZooAnimal *p = pZ; p; p = p->n_list ) p->draw();
```

becomes

```
struct ZooAnimal *p;
// vptr is the object's pointer to its table of
// virtual functions (see Figure 3).
for ( p = pZ; p; p = p->n_list )        /* C code */
        (*p->vptr[1])(p);
```

7.4 An Implementation of Pointers to Member Functions for Single Inheritance

A pointer to a class data member will hold the byte offset into the class object. What will a pointer to a class member function hold? It will hold either an address of a non-virtual member function or an index into the virtual table for that class. On most machine architectures, a simple trick can be used to distinguish between the two flavors without the use of a

discriminant. For example, given the declarations

```
ZooAnimal *pz;
void (ZooAnimal::*pmfv)();
```

pmfv may point to either the virtual function ZooAnimal::draw or the non-virtual ZooAnimal::locate. Its invocation

```
(pz->*pmfv)();
```

needs to distinguish between a function address and virtual table index if not to invoke a run-time disaster. It can do so by the following general translation:

```
(((int) pmfv) & (~127))
        ? (*pmfv)( pz );                          // a real function address!
        : (*pz->vptr[ ((int) pmfv)]) ( pz );      // an index into the virtual table
```

Of course, this implementation assumes a limit of 128 virtual functions to an inheritance graph. This is not desirable but in practice it has proved viable. The introduction of multiple inheritance, however, requires a more general implementation scheme and provides an opportunity to remove the limit on the number of virtual functions.

8. Derived and Base Class Member Pointers

A predefined, standard conversion guarantees that a pointer to an object of a derived class is implicitly converted to a pointer to a public base class. This holds even if the base class is virtual. For example, given the following simplified inheritance chain:

```
class Endangered { public: int adjustPopulation(int); };
class ZooAnimal { public: void draw(); int pz; };
class Bear : public virtual ZooAnimal  { public: void draw(); int pb; };
class Raccoon : virtual public ZooAnimal  { public: void draw(); };
class Panda :  public Bear, public Raccoon, public Endangered
        { public: void draw(); int pp; };
```

pointers to objects of type Bear, Raccoon and Panda may all be assigned to a pointer of type ZooAnimal. This flavor of implicit conversion is safe because each derived class is guaranteed to contain a base class object. (A discussion of the implementation of derived class objects can be found in [3].)

The opposite conversion, however, is not safe and requires an explicit cast on the part of the user. For example,

```
Bear *pb = new Panda; // ok: implicit conversion
Panda *pp = new Bear; // error: requires explicit conversion
Panda *pp = (Panda *) new Bear; // ok: but dangerous
```

A Panda object will always contain a Bear part. However, a Bear object may or may not contain a Panda part depending on what it has been set to last. The following sequence assigns 1024 to an undefined location in memory:

```
Bear b;
Bear *pb = new Panda; // ok: implicit conversion
pb = &b; // pb no longer addresses a Panda object
Panda *pp = (Panda *) pb; // oops: no Panda part
pp->pp = 1024; // disaster!
```

Users familiar with this behavior by habit extend it to pointers to class members. They assume a derived class member may safely be assigned to a public base class pointer to class member. A moments thought, however, reveals that this is not the case.

The class member pointer is dereferenced for a specific object of its class. This means that the base class pointer to class member will be bound to a base class object. This object may or may not contain a derived class part depending on what it has been set to last. If it does not contain a derived class part, the dereference is not safe. All references to memory allocated to the derived class part are undefined.

On the other hand, a derived class object is guaranteed to contain a base class part. Therefore, it is always safe to assign a member of a public base class to a derived class pointer to class member. The behavior of pointers to class members is the reverse of the predefined, standard conversion. For example, it is safe to assign a `Panda` pointer to class member a class member of `ZooAnimal`, `Bear` or `Raccoon`. The reverse assignments, however, will require an explicit cast. For example,

```
int Panda::* pmP = &ZooAnimal::pz; // ok: implicit conversion
int Bear::*  pmB = &ZooAnimal::pz; // ok: implicit conversion

f( Bear &b ) { if ( b.*pmP != b.*pmB ) /* ... */; }

void (Panda::*pmfP1)() = &Bear::draw; // ok: implicit conversion
void (Panda::*pmfP2)() = &Raccoon::draw; // ok: implicit conversion

f( Panda &p )
{
        switch ( p.*pmP ) {
        case GIANT_PANDA: (p.*pmfP1)(); break;
        case RED_PANDA:   (p.*pmfP2)(); break;
        // ...
        }
}
```

9. Casting

Pointers to class member functions may be explicitly cast to a pointer to any non-member function type. The reverse cast, however, is not legal. A pointer to a non-member function may not be cast to a pointer to class member function. For example,

```
extern f( void(*)() );
void (*pfv)();
void (Panda::*pmf_Panda)();

f( (void (*)())pmf_Panda ); // ok: requires an explicit cast
pfv = (void (*)())pmf_Panda; // ok: requires an explicit cast

// illegal: may not cast to a pointer to class member
pmf_Panda = (void (Panda::*)()) pfv;
```

Each explicit cast of `pmf_Panda` tests for whether it contains an address or index, and assigns the proper field to `pfv`. When there is no possibility of `pmf_Panda` addressing a virtual function, the test is not generated and the `f` part is assigned directly. This sort of explicit cast is unsafe but is necessary to avoid breaking existing code.

Pointers to class member functions may be explicitly cast to a pointer to void. The reverse cast, however, is illegal since information is lost in the cast of a `void*`. For example,

```
extern f( void* );
void *pv;
void (Panda::*pmf_Panda)();

f( (void *)pmf_Panda ); // ok: requires an explicit cast
pv = (void *)pmf_Panda; // ok: requires an explicit cast

// illegal: may not cast to a pointer to class member
pmf_Panda = (void (Panda::*)()) pv;
```

The explicit cast of a class member address must include both the class and data type of the conversion. For example,

```
// casting a pointer to a derived member to a
// pointer to a base member requires an explicit cast
int ZooAnimal::* pmZ = (int ZooAnimal::*) &Bear::pb;
f( Bear &p ) { return ((ZooAnimal)p).*pmZ; }

void (ZooAnimal::*pmfZ)() = (void (ZooAnimal::*)()) &Bear::draw;
void ff( Bear &p ) { (((ZooAnimal)p).*pmfZ)(); }
```

Recall, the value of a pointer to a class data member is the member's byte offset into the class object. It is stored as a `short`. A pointer to a class data member may be explicitly cast to any data type a `short` value may be cast to. It is illegal to cast a non-pointer to class data member into a pointer to class data member. An attempt to do so will result in a compile-time error.

10. An Implementation of Pointers to Member Functions for Multiple Inheritance

Consider this multiply derived inheritance graph:

```
class Endangered {
public:
        void adjustPopulation(int);
        virtual void highlight();
        virtual void inform();
        // ...
};

// Bear and ZooAnimal are defined on page 11.
class Panda: public Bear, public Endangered {
public:
        void onDisplay();       // non-virtual member
        void draw();            // redefines Bear::draw
        void debug(int);        // redefines Bear::debug
        void inform();          // redefines Endangered::inform
        virtual Time& feedingHours();
        // ...
};
```

In a portable implementation of virtual functions under multiple inheritance, it is no longer enough to have the index into the virtual table.[3]. For example, an invocation of the `highlight()` virtual function will require an additional offset. Since Panda has not defined its own instance, the Endangered instance will be called. `highlight()` expects a pointer to an Endangered class object. This additional offset will locate the Endangered `this` pointer within a Panda class object. (This is illustrated in Figure 4.)

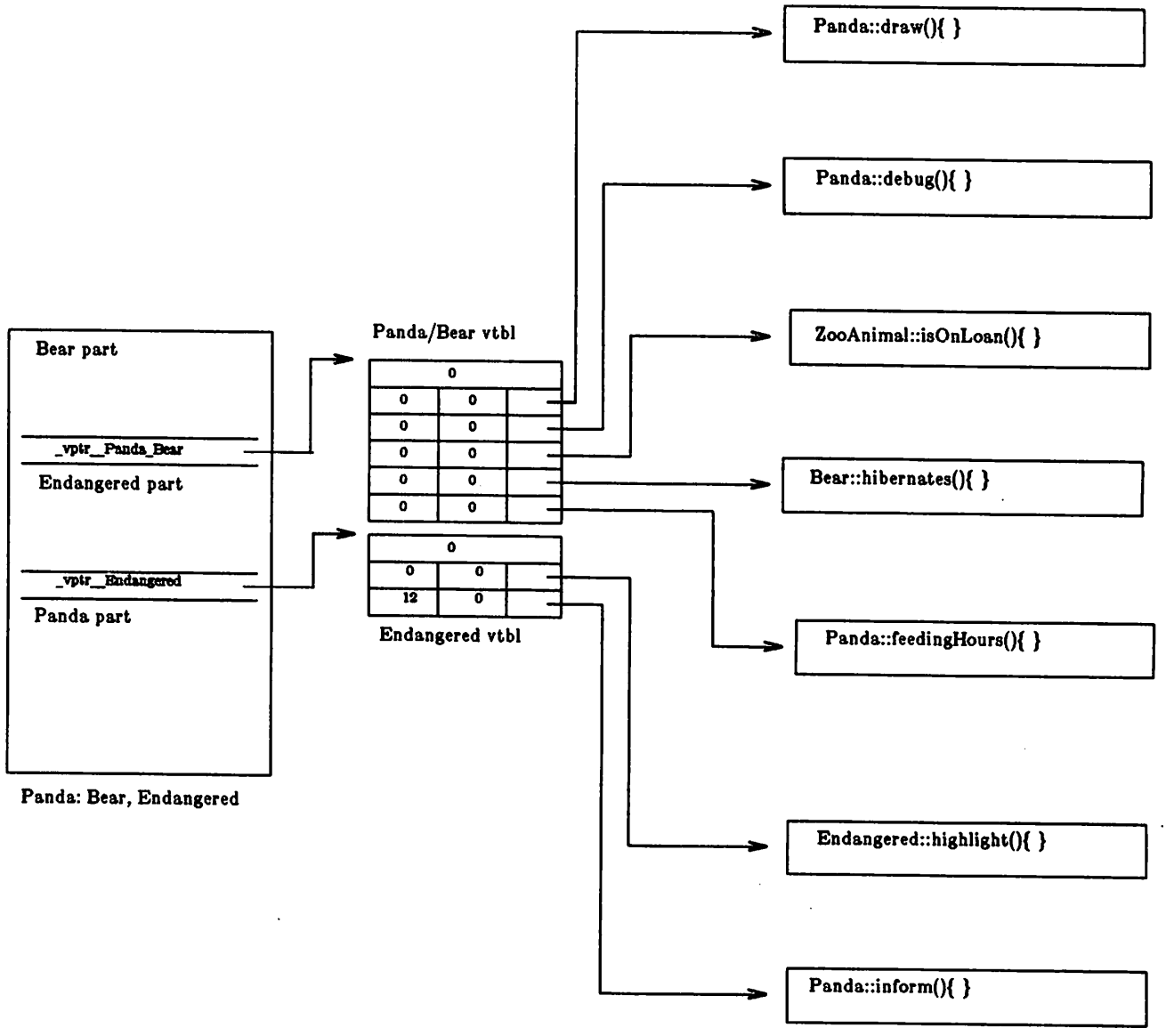


Figure 4. Multiple Inheritance Virtual Function Layout

A pointer to member function, in order to handle non-virtual functions, virtual functions *and* possible multiple virtual tables, requires three elements:

- an offset, or delta, `d`, to permit computing the appropriate `this` pointer,
- an index, `i`, into the virtual table, and
- an address, `f`, of a non-virtual function address, *or* an offset, `d2`, of the `vptr` in the object identified by `d`.

In addition, there needs to be some discriminant to determine if the pointer contains an index or an address. However, since an index is only valid as a non-negative value, a negative index can be used to flag the presence of a non-virtual function address. This allows the representation of a pointer to no function to have the traditional "all zeros" representation.

A pointer to class member function, therefore, is defined as a structure of the form

```
typedef int (* vptr)();
class Memptr {
public:
        short d;
        short i;
        union { vptr f; short d2; }
};
```

Note that the virtual table itself is simply an array of pointers to members (`Memptr[]`). The virtual table pointers inserted within a class object are pointers of type `Memptr`, and of course are set to point to the first entry of the virtual table (See Figure 4).

For example, here is how an initialization of a virtual member function address to a pointer to a class member function is handled:

```
void (Panda::*pmfv)() = &Panda::feedingHours;
```

becomes

```
struct Memptr pmfv = {
        0,      // pmfv.d, offset, finds a base class this pointer
        4,      // pmfv.i, 5th entry in Panda/Bear virtual table
        0       // pmfv.f, address, set for non-virtual member functions
};
```

Similarly, an initialization of a non-virtual member function to a class member function is handled as follows:

```
void (Panda::*pmfv)() = &Panda::onDisplay;
```

becomes

```
struct Memptr pmfv = {
        0,      // pmfv.d, offset, finds a base class this pointer
        -1,     // pmfv.i, negative entry means address of function is stored
        onDisplay__Panda // pmfv.f, address set to program function instance
};
```

The most complicated example is a virtual function declared in a second (or subsequent) base class:

```
void (Panda::*pmfv)() = &Panda::highlight;
```

becomes

```
struct Memptr pmfv = {
        12,     // pmfv.d, offset of Endangered part of Panda object
        1,      // pmfv.i, highlight's entry in Endangered's vtbl
        4,      // pmfv.d2, offset of Endangered's vptr
};
```

Finally,

```
void (Panda::pmfv)() = 0;
```

becomes

```
struct Memptr pmfv = {  0, 0, 0 };
```

A call of the member class pointer switches on a test of the index, as follows:

```
Panda * pp;
(pp->*pmfv)();
```

becomes

```
struct Panda * pp;      /* C code */
char* temp = ((char*)pp)+pmfv.d;
(pmfv.i < 0 )
        ? (*pmfv.f)( temp );    // non-virtual member function call
        : // virtual function call using virtual table
        (*(((memptr*)(temp+pmfv.d2)->vptr[ pmfv.i ].f)(temp + pp->vptr[ pmfv.i ].d));
```

Conversions between pointers to class members of a derived class and a 2nd or subsequent base class need to adjust member offsets. For a pointer to class data member, this means adjusting the physical offset by the `sizeof` the intervening base classes. For example, in a conversion between `Panda` and `Endangered`, the combined sizes of `Bear` and `Raccoon` must be either added or subtracted:

```
int Endangered::*p_Endangered;
int Panda::* p_Panda;

// p_Endangered == 1
p_Endangered = &Endangered::pe;


// ok: implicit conversion
// p_Panda ==> p_Endangered+sizeof(Bear)+sizeof(Raccoon)
p_Panda = p_Endangered;

// requires explicit cast
// p_Endangered ==> p_Panda-sizeof(Bear)-sizeof(Raccoon)
p_Endangered = (int Endangered::*)p_Panda;
```

For a pointer to a non-virtual class member function, the `d` offset will be set plus or minus the combined size of the intervening base classes. For example,

```
void (Endangered::*pmf_Endangered)(int);
void (Panda::*pmf_Panda)(int);

// pmf_Endangered == { 0, -1, &adjustPopulation }
pmf_Endangered = &Endangered::adjustPopulation;


// ok: implicit conversion;
// pmf_Endangered == {
//      // offset of Endangered part of Panda
//      sizeof(Bear)+sizeof(Raccoon),
//      -1,
//      &adjustPopulation }
pmf_Panda = pmf_Endangered;
```

A pointer to a virtual class member function will also need to set `d2` to offset into the relevant `vptr`.

The second offset `d2` is necessary because it is not possible to assume that the `vptr` is in a fixed position in every object. In particular, unless we make room for a `vptr` in every object, it is not possible to make the `vptr` the first word of an object. For example:

```
struct S {
        int s1;
        int s2;
};

class D : public S {
        virtual f();
        int d1;
};
```

The `S` part of a `D` may be used in a context where it is not known that it is a part of a `D`. Therefore, `D`'s `vptr` must be allocated after `s1` and `s2`.

Conclusions

Pointers to members are important because if they were left out of C++ they would leave a hole in the type system. This would force people to "cheat" and force implementors to use sub-optimal techniques to allow such cheating. The design and implementation of pointers to functions was determined by syntactic, semantic, and implementation details of other language features to such an extent that it could be claimed that they were discovered rather than invented. Pointers to members are most often used in ways that closely resembles the use of pointers to functions; hence the name "pointers to functions." The introduction of multiple inheritance complicated the originally very simple implementation of pointers to members, but also caused a unification of the concept of a pointer to member and an entry in a virtual function table.

Acknowledgements

The design of the pointer to member concept was a cooperative effort by Bjarne Stroustrup and Jonathan Shopiro with many useful comments by Doug McIlroy. The notion of static member functions is due to Jonathan Shopiro. Steve Dewhurst contributed greatly to the redesign of the pointer to member implementation to cope with multiple inheritance. Thanks to all who commented on earlier versions of this paper and/or found errors in the early implementations.

REFERENCES

1. "The Evolution of C++: 1985 to 1987", B. Stroustrup, *Proceedings, USENIX C++ Workshop*, 1987.

2. *The C++ Programming Language*, B. Stroustrup, Addison-Wesley, 1986.

3. "Multiple Inheritance for C++", B. Stroustrup, *Proceedings, EUUG Spring'87 Conference*, Helsinki, 1987.

EXCEPTION HANDLING WITHOUT LANGUAGE EXTENSIONS

William M. Miller, Software Development Technologies, Inc.

ABSTRACT

Exception handling is defined and its relevance to an object-oriented programming language such as C++ is explored. Design goals for exception handling methods are discussed, and an implementation of one such method not requiring extensions to the C++ language is described.[1]

1. Introduction

An *exception* is an event which derails the "normal" processing of a program. Such events may arise from within the program (exceeding the bounds of an array, for example), from services used by the program (no more memory is available for allocation), from data used by the program (unsuccessful conversion attempt), or from external causes (the user hit "break").

In small programs, recovery from the failure of an operation can usually be done most effectively at the point at which the problem was detected. As program size and complexity increase, however, issues of modularity make error recovery at the point of failure less and less appropriate. In a large program the method of handling an error will typically depend more on the context from which a general-purpose service routine was called than on the nature of the function itself. Thus, the major challenge of exception handling is the transfer of information and control from the code detecting the problem into a context suitable for resolving it.

In languages such as Ada [1] and PL/I [2], exception handling is provided by native features. C++, however, reflecting its origins in the "bare bones" approach of the C language, currently affords no such assistance to the programmer. Ironically, the other strengths of C++ make exception handling almost essential. As an object-oriented language it is intended to facilitate the development of software components which can be used without modification in a wide variety of environments, and clearly the methods which should be used for error recovery will vary in different applications. Furthermore, one of the staple techniques used in other languages which do not provide built-in exception

[1] The work described in this paper was the subject of a previous article by the author [3], who retains rights to the material published therein.

handling -- returning a completion status -- is inappropriate for functions which are invoked via overloaded operators.

Although the seriousness of this deficiency in C++ has been recognized and a proposal to extend the language to include exception handling has been outlined [4], at this writing the proposal is still in embryonic form and the date at which such a feature might become available is unknown. The remainder of this paper presents an approach which, though not without weaknesses, can be implemented to address the problem in the short term.

2. Design Considerations

The major function of an exception handling mechanism as described above is to pass information and control from the point at which a problem is detected to a context in which the correct recovery strategy may be applied. In more concrete terms, this means that a given function or block can establish a *handler* which will be invoked if an exception is subsequently *raised* in that block or in one of its direct or indirect dynamic descendants. The following design goals should be kept in mind in considering any approach to exception handling:

2.1. Flexible Recovery Actions

There are at least three possible actions a handler ought to be able to take when invoked by an exception:

- *abort* the operation currently in progress and transfer control to some location within the context of the block which established the handler (this option is the only one allowed in the current proposal for built-in exception handling)
- *retry* the operation, presumably after repairing the condition which led to the problem
- *abdicate* handling the exception, allowing it to propagate to a handler established earlier in the stack

2.2. Transparency

This criterion requires that intervening code between the handler and that raising the exception need take no special action to facilitate transfer of information and control to the handler. In particular, code should not have to check for completion values and do manual cleanup and propagation of error conditions.

It is this latter consideration which causes the most trouble for C++. The *setjmp* and *longjmp* functions are available to perform the *abort* action described in the preceding section; however, these functions were inherited from C and have no way of detecting that the stack frames being discarded may contain objects with

destructors. This failure to invoke destructor code makes *setjmp/longjmp* unsuitable by themselves for use in a C++ environment.

2.3. High Information Bandwidth

This requirement refers to the ability of the code detecting an exception to exchange information with the handler and is actually implicit in the first design goal. For example, the decision to *abdicate* from the processing of an exception would almost certainly be based on details concerning the exact nature and cause of the exception, else why would the handler be present at all? Similarly, the decision to *retry* an operation would usually involve communicating some modified information back to the exception-producing code, lest the same problem cause another exception.

2.4. Low Performance Overhead

Although good performance is an obvious requirement for any design, the particular emphasis in exception handling must be to reduce the overhead encountered when an exception is *not* raised. In the vast majority of cases, processing will be normal and exceptions will not be encountered, so the penalty for having the feature and not using it must be reduced as far as practical.

2.5. Portability

Any proposal must not rely on intimate knowledge of machine-specific information such as the format of stack frames; otherwise the availability of the approach will be limited to gurus and those with access to them.

3. An Interim Approach

Although the ideal solution would be to have exception handling implemented as an integral part of the language, it is possible to satisfy the listed design goals to a large degree within the current definition of C++. In brief, the approach described below utilizes a stack of all objects with destructors which grows and shrinks in parallel with the call stack and which is used to destroy objects prior to invoking *longjmp* to perform an *abort* operation. Exception handlers are implemented as a function containing the code of the handler and an object which associates the function with a given block.

The design is described below in four parts. The first part discusses a class which maintains the parallel stack and provides the necessary hooks for cleaning up prior to an *abort* operation. The second part covers issues related to transfer of information to and from exception handlers. The third describes the class used to establish an exception handler. The final section discusses the code necessary to raise an exception.

3.1. The *cleanup_obj* Class

As mentioned above, the major problem in C++ with respect to exception handling is the fact that *longjmp* is not capable of invoking the destructors of objects created in the stack frames it discards. This design addresses the issue via a class which implements the stack of objects with destructors (see Listing 1 below). The *cleanup_obj* class provides a base from which may be derived classes requiring cleanup in the event of an *abort*.

The stack is implemented as a doubly-linked list. The simpler singly-linked organization is not adequate because strict LIFO removal is not guaranteed; for instance, current implementations of cfront invoke both constructors and destructors of array members in increasing order of subscript. Also, this mechanism can be used for objects created with the *new* operator, which allows for arbitrary order of removal.

Two constructors are provided for the *cleanup_obj* class. The first takes no arguments and simply pushes the object unconditionally on the stack. The second takes an integer argument which, if non-zero, causes the object *not* to be pushed. This second constructor is included to facilitate the use of *cleanup_obj* with objects created via *new*. In some circumstances objects allocated in the heap are to be used as local temporaries and should be cleaned up in the event of an *abort*; other such objects may be intended for longer duration and should survive an *abort*. The second constructor allows this flexibility at the expense of a bit more execution time overhead.

Because C++ does not allow invoking destructors directly in an implementation-independent fashion, classes derived from *cleanup_obj* must not supply their own destructors. Instead, code which would ordinarily have occurred in a destructor should instead be placed into the *cleanup()* member function, allowing it to be invoked both during an *abort* and by the *cleanup_obj* destructor. In addition to the class-specific actions normally performed by the destructor, the derived *cleanup()* function must invoke *cleanup_obj::cleanup()* at some point in its processing in order to remove the object from the stack.

A further responsibility of the derived class, and particularly the *cleanup()* function, is to manage the storage used by the object itself. If the object was allocated in the heap via *new*, that storage will not be freed automatically during an *abort* operation because the destructor is not invoked. If a *cleanup_obj* class is to be used with *new*, the derived class constructor must set a flag in the object indicating whether the storage is in the heap or otherwise (testing the value of *this* for zero upon entry to distinguish these cases and then setting it explicitly). The *cleanup()* function is invoked with an argument which is non-zero to indicate that the object is being destroyed as the result of an *abort* operation, and if both an abort is in progress and the storage was allocated in the heap, the object should be deleted as the last operation before returning. Unfortunately, this storage management cannot be relegated to the base class because there is no implementation-independent way for the base class constructor to determine whether the allocation is static, automatic, or in free store.

There are two features of the new language definition (reflected in cfront version 2.0) which will improve *cleanup_obj*. The first is the class-specific *new*

operator, which will allow the base class to assume the burden of storage management described in the preceding paragraph. The second is multiple inheritance. It may be awkward or even impossible to have to derive classes from *cleanup_obj* because they actually need to be derived from a different base class. With multiple inheritance, that will no longer be a problem.

3.2. Exception Types

One of the design goals mentioned earlier was to allow unlimited bandwidth into and out of an exception handler. This feature is provided by the *exception_type* class and its derivatives (see Listing 2). In its simplest form, an *exception_type* object contains information about the exception to be communicated to the handler. The member *exc_num* can be used to identify the kind of exception which occurred (e.g., subscript out of bounds, heap full, etc.), while *msg_fmt* and the virtual member function *msg_text()* give access to a printable message describing the problem. In addition, *return_allowed* determines whether or not an exception handler can request a retry or continuation of the processing which raised the exception.

Classes derived from *exception_type* can provide much more information to the handler about the details of the exception, including a version of *msg_text()* which uses the additional information in constructing the printable message. Furthermore, members can be included to allow the exception handler to communicate information back to the point of the error for use in a retry of the failing operation. The *msg_fmt_args* class is provided so that classes requiring use of *sprintf()* to generate a message utilizing some or all of the additional information will have a buffer to use.

An alternative implementation of *exception_type* was considered before settling on the version reproduced in Listing 2. This alternative used one derived class for each kind of exception. This approach was rejected for two reasons. First, in view of the fact that there might be literally hundreds of kinds of exceptions, it seemed to be an unnecessary strain on the compiler's symbol table, potentially rendering the method unusable on small systems. Having separate classes only for the different kinds of information communicated to and from exception handlers seemed much less likely to overwhelm the capacities of small systems.

The second problem with a class per exception approach is that exception handlers will frequently be specialized only to be able to respond to a small range of exceptions, abdicating responsibility for all others to a more inclusive handler earlier in the stack. C++ intentionally does not allow discrimination between sibling classes of this kind, necessitating a "type" field of some sort; since explicit identification of the exception type is required anyway, the additional type security afforded by having different classes for each kind of exception is not worth the added complexity.

3.3. Exception Handlers

There are two parts to an exception handler under this design, a function which is invoked when an exception is raised, and a class which provides the linkage between the function and a specific block. The type of the exception handler function is declared by the type name *exc_hdl_func*, and the linkage class is *exception_handler* (see Listing 3).

The *exception_handler* class is derived from *cleanup_obj* because it also needs to track the call stack (and thus provides an example of how to build derived *cleanup_obj* objects). It implements a stack within a stack, allowing exceptions to be propagated hierarchically from handler to handler. It also contains a pointer to an *exc_hdl_func* which is to be invoked when an exception is raised and a *jmp_buf* to which control will be transferred if the exception handler chooses to abort the operation.

The *abort()* member function implements the operation of the same name. It simply walks the stack of objects to be cleaned up, invoking the *cleanup()* function for each, until it reaches the exception handler which invoked the *abort* operation. Note that any objects which were created after the *exception_handler* object will be cleaned up, even if they are in the same block: caution regarding block structure and order of execution is required. After all necessary cleanup operations are complete, *abort()* calls *longjmp()* to transfer control to the location indicated by the contained *jmp_buf*.

Exception handler functions receive two arguments, an *exception_type* object as described in the preceding section, and a pointer to the *exception_handler* object on whose behalf the handler function was invoked. There are three ways to terminate the execution of an exception handler function, corresponding to the three actions described in the design goals above:

- *abort*: invoke *ehp->abort()*. This call will never return.
- *retry*: return a non-zero value. This action is only permissible if *return_allowed* has been specified in the *exception_type*.
- *abdicate*: return zero.

Two actions are required to establish an exception handler in a given block. First, simply declaring an automatic object of type *exception_handler* places the linkage object into the stack, associating the block with the exception handler function specified as the initial value (if none is given as an initial value, or if the action to be taken changes at different locations in the code, the overloaded assignment operator allows changing the association at any subsequent point in the block). Second, the *abort_label* member must be initialized via an invocation of *setjmp()*. It is imperative that this action be completed before any code which might raise an exception.

3.4. Raising an Exception

The *raise_exception()* function (see Listing 4) is passed an *exception_type* (or one of its derivatives) which has been previously initialized to appropriate values for the particular exception. It simply traverses the stack of *exception_handler* objects, invoking each associated exception handler function in turn, until one either calls *abort()* or returns a non-zero value. The caller of *raise_exception()* should treat a return of control as a request for either retrying or continuing the operation, whichever is appropriate in terms of the semantics of the processing involved.

4. An Example

Listing 5 provides a complete example program demonstrating the use of the classes and routines of Listings 1-4. It defines the skeleton of a string class which cleans up the storage allocated for the string data in the event of an *abort* operation and which raises an exception in the event of an attempt to access a character with an offset larger than the current length of the string.

To begin, we derive an *exception_type* which allows two integer arguments to be passed into the exception handler and which allows the handler to return an integer value for use in a retry attempt (*exception_argII_retI*, conventionally named to indicate the types of its arguments and return value). The *msg_text()* member function is also upgraded to use the two integer arguments in creating the associated printable message.

Next, we define the outline of a string class by deriving it from *cleanup_obj*. The constructor allocates space for the string data, so the *cleanup()* member function, which replaces the normal destructor, deletes the storage. In addition, the constructor sets a flag specifying whether or not the allocation was via *new* so that *cleanup()* can determine whether to delete the storage for the object itself in the event of an *abort*.

The subscript operator checks to see that the desired subscript is less than the current length of the string. If it is not, it raises an exception by declaring an object of type *exception_argII_retI*, initializing it with the appropriate values to describe the problem and allow a return, and calling *raise_exception()*. If that function returns, it tries the subscript operation again, using the return value from the handler in place of the original subscript.

Next come two exception handler functions. The first is very simple and just prints the message associated with the exception and calls *abort()*. The second attempts to recover from the error by using the maximum allowable subscript as the new value. Since it is specific to subscript errors, it abdicates on all other kinds of exceptions, and if there is no allowable subscript value (i.e., the string length is zero), it aborts.

Finally, the main function establishes an exception handler using the *use_max* handler function and initializes the *abort_label* member using *setjmp()*. Note that if *setjmp()* returns a non-zero value, it is the result of an *abort* operation

(the value returned is that passed to *abort()*), so the *else* clause contains the normal processing after establishing the handler.

5. Conclusions

The design presented in this paper is an attempt to provide most of the benefits of exception handling during the time before a built-in implementation is available; it is not a claim that language extensions for exception handling are unneeded or undesirable. In particular, two of the design goals described above are not met fully. First, exception handling is not fully transparent to code which is not directly involved. Objects which require cleanup must be derived from the *cleanup_obj* class, and their destructors must be converted to *cleanup()* functions. Although restricting the changes only to small parts of the class declarations in this way is an improvement over ad hoc approaches to error handling and cleanup, an implementation integrated with the language would eliminate these "warts."

Second, there is some performance overhead involved in maintaining the doubly linked list for the stack of *cleanup_obj* objects. Whether or not this overhead is a problem will depend completely on the nature of the application. In the implementation used by the author, the overhead of linking and unlinking an object proved to be about half the time required to allocate and free a data item in the heap, which is probably one of the most common reasons cleanup functionality would be required. If blocks do a relatively large amount of processing compared with the time required to construct and destroy objects, the penalty is probably negligible. If, on the other hand, functions are typically small and each declares a number of objects requiring cleanup, the performance degradation would be quite noticeable. Again, an integrated implementation could keep track of objects requiring cleanup with substantially less overhead.

The approach described in this paper has, however, proved useful in the author's projects, and he presents it now in the hope that it will be helpful to others as well.

REFERENCES

[1] ANSI/MIL-STD-1815A-1983, *Reference Manual for the Ada Programming Language.*

[2] ANSI X3.53-1976, *American National Standard Programming Language PL/I.*

[3] Miller, William M., "Error Handling in C++," *Computer Language*, May, 1988.

[4] Stroustrup, Bjarne, "Possible Directions for C++," *Proc. USENIX C++ Workshop, 1987.*

```
// Derive from cleanup_obj to have objects cleaned up by
// exception_handler.abort()

class cleanup_obj {
   friend class exception_handler;      // so abort() can destroy objects
                                        // on the stack

   static cleanup_obj* head;
   cleanup_obj* prev;                   // points to newer object
   cleanup_obj* next;                   // points to older object

protected:

// With no arguments, object is always pushed onto the stack

   cleanup_obj():
         prev(NULL), next(head) {
      if (next != NULL)
            next->prev = this;          // back-link from old head
      head = this;
      }

// Push onto the stack only if object is not to be permanent

   cleanup_obj(int permanent) {
      if (permanent)
            next = prev = this;     // link to self (makes cleanup() safe)
      else {
         if ((next = head) != NULL)
               next->prev = this;
         head = this;
         prev = NULL;
         }
      }

   ~cleanup_obj() {
      cleanup(0);
      }

// Pop object from stack

   virtual void cleanup(int aborted) {
      if (prev != NULL)
            prev->next = next;
      else head = next;             // if nothing newer, this must be head
      if (next != NULL)
            next->prev = prev;
      }
   };
```

LISTING 1: The *cleanup_obj* Class

```
// Use exception_type for messages which have no arguments

struct exception_type {
   int exc_num;
   const char* msg_fmt;
   int return_allowed;

   exception_type(const int exn, const char* fmt, const int retable):
         exc_num(exn), msg_fmt(fmt), return_allowed(retable) {}

   virtual const char* msg_text() {
      return msg_fmt;
      }
   };

// msg_fmt_args is simply a base type for messages which have arguments,
// providing a buffer for use with sprintf().

const int MAX_MSG_LEN = 160;

struct msg_fmt_args: public exception_type {
   char buf[MAX_MSG_LEN];

   msg_fmt_args(const int exn, const char* fmt, const int retable): (exn,
         fmt, retable) {}
   };
```

LISTING 2: Exception Types

```
// Type for exception handler functions

typedef int exc_hdl_func(exception_type& excp,
      class exception_handler* ehp);

class exception_handler: public cleanup_obj {
   friend void raise_exception(exception_type& excp);
               // friend to allow traversal of stack of exception_handlers

   static exception_handler* top_of_stack;
   exception_handler* next;          // points to older handler
   exc_hdl_func* func;

public:
   jmp_buf abort_label;

   exception_handler(exc_hdl_func* exc_func = NULL):
         next(top_of_stack), func(exc_func) {
      top_of_stack = this;
      }

   void cleanup(int aborted) {
      top_of_stack = next;
      cleanup_obj::cleanup(aborted);
      }

   void abort(int ret_val);

   void operator=(exc_hdl_func* exc_func) {
      func = exc_func;
      }
   };

// Code for abort() member function (not inline because the current
// cfront implementation does not allow loops in inline functions)

void exception_handler::abort(int ret_val)
{
   while (cleanup_obj::head != NULL && cleanup_obj::head != this) {
      cleanup_obj::head->cleanup(1);
      }
   longjmp(abort_label, ret_val);
   }
```

LISTING 3: The *exception_handler* Class

```
void raise_exception(exception_type& excp)
{

// Walk the stack of exception handlers, invoking handler functions

   for (exception_handler* ehp = exception_handler::top_of_stack;
         ehp != NULL; ehp = ehp->next) {
      if (ehp->func != NULL && (*ehp->func)(excp, ehp)) {

// TRUE return => return to exception for retry or continue

         if (excp.return_allowed)
               return;
         fprintf(stderr, "Illegal return to exception: %s\n",
               excp.msg_text());
         exit(998);
         }
      }

   fprintf(stderr, "Unhandled exception: %s\n", excp.msg_text());
   exit(999);
   }
```

LISTING 4: *raise_exception*

```
// Define an exception type with two integer arguments, returning
// a string from the exception handler

struct exception_argII_retI: public msg_fmt_args {
   int int_arg1;
   int int_arg2;
   int ret_val;

   exception_argII_retI(const int exn, const char* fmt, const int retable,
         const int i1, const int i2): (exn, fmt, retable), int_arg1(i1),
         int_arg2(i2) {
      ret_val = 0;            // default return value
      }

   const char* msg_text() {
      sprintf(buf, msg_fmt, int_arg1, int_arg2);
      return buf;
      }
   };

// Define a string class (skeleton only) as a cleanup_obj

class string: public cleanup_obj {
   int in_heap;       // TRUE => allocated via "new"
   int max_len;       // maximum length string
   int cur_len;       // current length of string
   char* data;        // pointer to data of string

public:
   string(int maxl): max_len(maxl), cur_len(0) {
      if (this == 0) {
         this = (string*) new char[sizeof(string)];
         in_heap = 1;
         }
      else {
         this = this;      // needed to invoke base constructor
         in_heap = 0;
         }
      data = new char[max_len + 1];
      }

   void cleanup(int aborted) {      // in place of destructor
      delete data;
      cleanup_obj::cleanup(aborted);
      if (aborted && in_heap) {
         delete (char*) this;       // cast to avoid recursion
         }
      }

   char& operator[] (int sub);
   };
```

LISTING 5 (page 1 of 3)

```
// Define subscript operator that raises an exception if the
// subscript is out of range

#define SUBSCRIPT_RANGE_EXCP 1000

char& string::operator[] (int sub)  {
   while (sub >= cur_len) {

      exception_argII_retI excp(
            SUBSCRIPT_RANGE_EXCP,
            "Subscript (%d) too big (must be < %d)",
            1,       // Allow return
            sub, cur_len);
      raise_exception(excp);
      sub = excp.ret_val;
      }
   return *(data + sub);
   }

// An exception handler function which always prints the associated
// message on stderr and aborts

int do_abort(exception_type& excp, exception_handler* ehp) {
   fprintf(stderr, "%s\n", excp.msg_text());
   ehp->abort(1);       // arbitrary non-zero number
   return 0;            // never reached
   }

// An exception handler function (specific to subscript range error)
// which attempts to repair the error by using the maximum allowable

int use_max(exception_type& excp, exception_handler* ehp) {
   exception_argII_retI* subscr_excp = (exception_argII_retI*) &excp;

   if (excp.exc_num != SUBSCRIPT_RANGE_EXCP)
         return 0;      // abdicate
   if (subscr_excp->int_arg2 > 0) {
      subscr_excp->ret_val = subscr_excp->int_arg2 - 1;
      return 1;
      }
   do_abort(excp, ehp);
   return 0;      // never reached
   }
```

LISTING 5 (page 2 of 3)

```
int main() {
   exception_handler eh(use_max);   // establish handler linkage

   if (setjmp(eh.abort_label)) {
      return 1234;      // an exception was aborted
      }
   else {
      string abc(100);
      // ...
      abc[10] = 'x';    // generates exception
      }
   }
```

LISTING 5 (page 3 of 3): An Example

Solving the RPC problem in GNU C++

Michael D. Tiemann†

Experimental Systems Project
Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, TX 78759

ABSTRACT

The C++ programming language is being used in a number of projects which implement some form of distributed execution model. While each of these projects differ in their goals and implementations, all of them ultimately depend on some kind of remote procedure call (RPC), a facility which is not easily supported by C++. For this reason, many different extensions to C++ have been implemented, each one coping with this problem in its own, unique way. This paper presents a new construct, called a "wrapper," which provides a more general solution to the problem than previous work, and in some cases, provides greater efficiency as well.

Wrappers are a highly experimental feature, implemented in the GNU C++ compiler; this paper represents the start of their evolution. GNU C++ is a highly optimizing native code C++ compiler for the Sun3 and the VAX running BSD 4.[23]. The compiler, as well as a C++ source level debugger, GDB+, a linker, documentation, and library support, is available as free software, under the terms of the GNU General Public License.

1. Introduction

The Experimental Systems (ES-Kit) research at the Microelectronics and Computer Technology Corporation (MCC) is focused on development of rapid implementation technologies applicable to low cost prototyping of parallel computing systems. The architecture of these systems will be based on hardware modules, linked together via bus and/or message communication channels. These modules will provide system-level functionality such as disk, memory, processor, etc., allowing a variety of architectural configurations to be assembled and measured without great expense. The operating system for a machine comprised of these modules must support parallel applications, and it must provide a convenient interface to UNIX. The software language platform must support rapid reconfigurability to match the hardware, software module reusability, and great enough efficiency that high-performance designs can be reasonably proven against more traditional systems.

Such requirements immediately qualified the C++ programming language. Using the object oriented programming model to separate program functionality into logical blocks (of both code and data), generic operations can be implemented

† On leave at INRIA. B.P. 105, 78153 Le Chesnay Cedex, France.

and interfaced in a number of different ways, all in a consistent manner. Reusability is one of the keys to rapid prototyping. By allowing the user to define the semantics relating to such operations as type conversion and even the operators in expressions, code can be written that looks natural, while in fact its operations may be quite complex. The abstract data typing facilities of C++ make it possible to design from a very high level, and then refine the implementation by substituting functionality and/or behavior where desired. Yet C++ can still be compiled into very efficient code, usually as efficient as normal C code would be.

For all of its strengths, C++ has some notable shortcomings as well. Just as a user on a single processor system would like control over the type abstractions of their application, users of parallel systems need to gain control over the programming languages semantics as control and data moves across logical and physical machine boundaries. C++ fails to provide any mechanism for this requirement, and few general purpose languages do. The absence of a good way to specify RPCs in C++ has led some groups away from the language ([Cap87]), and others to extend it in differing, incompatible ways ([Gri87],[Cal87],[Gau87],[Lin87]).

The ES-Kit project, too, has developed some extensions to the C++ programming language which give programmers control at these boundary conditions, and preserve, in a very substantial way, all the efficiency one could expect using a language like C. The baseline compiler for these extensions is the GNU C++ compiler, a highly optimizing, native code C++ compiler, largely compatible with C++ V1.2 from AT&T.[1] The compiler also has a C++ source level debugger, GDB+, and a linker which supports incremental linking with correct C++ semantics.

This paper will begin with a presentation of the ES-Kit software environment, as a motivation for the implementation goals which resulted. A section on related work gives this work context, and suggests that the approach, while well suited to C++, is relatively language-independent. The implementation of wrappers is then presented, along with their semantics and a suggested syntax. The implications (and limitations) of wrappers are then discussed, and future areas of research are outlined. The paper finishes with a discussion of other uses to which wrappers can be put.

2. Requirements and Goals

Remote procedure calls form the basis of the ES-Kit parallel execution model. Unlike the serial environment, where procedure calls are mainly used to enhance maintainability of software through modularity, the ES-Kit programming model uses them to create parallelism. When a (remote) procedure call is made, control forks, with one thread acting on the procedure call, while the other returns to the caller. It is at the option of the caller whether to block the caller thread until the call has completed, or to continue. It is absolutely essential that the programming language provide the user with a convenient way of manage this execution model, or better yet, forgetting about it, just as a procedure-call oriented environment makes the use of its call stack simple and painless, and something which can almost always be ignored.

[1] All C++ 2.0 features are implemented except for multiple inheritance.

The basic problem in implementing a system which abstracts procedure calls in C++ is that, like C, C++ does not provide any operations on function calls except to execute them. It is possible to say sizeof (int), and the C++ compiler will return the system's integer size in bytes, but one cannot ask for the length of an argument list, or reliably get the number of arguments passed, their types, their positions and dispositions on the stack, etc. Thus, it is extremely difficult for the systems programmer to extend an abstraction when one hardly exists.

For reasons that were not immediately obvious, it was decided that the compiler would handle this problem in as general a fashion as possible, subject to the following constraints:

(1) any C++ member function call can be an RPC,

(2) the client source code should not reflect whether the member function call is local or remote,

(3) the solution must be portable (preferably implementable within the limitations of a C virtual machine),

(4) there should be minimal (if any) change to the GNU C++ compiler,

(5) it easily supports existing RPC environments (such as Sun RPC),

(6) it is easy to implement blocking vs. non-blocking semantics,

(7) it is easy to describe how objects should be passed (migrated) for the call.

3. Related Work

The parallel execution model of the ES-Kit environment is derived from Halstead's work on *futures*[Hal85]. One of the main differences between the ES-Kit environment[Led88] and Halstead's Multilisp environment is that C++ is statically typed, hence futures can have specific future types, leading to somewhat improved efficiency and safety. This realization was also made by Liskov in her presentation of *promises*[Lis88]. The ES-Kit future is like Liskov's *promise*, though we have not implemented other features of her work, such as *call-streams*.

The CLAM system[Mil86] provides an RPC facility which relies on a *bundler*. Bundlers are attached to parameters when passing explicit pointers does not have the desired semantics. For example, if a routine wanted to send a string as an argument to a remote procedure, passing the address of the string on the local machine does little good. A bundler can format the string into a parameter that is suitable for interhostal transit. Bundlers must be explicitly placed with each parameter in need of bundling.

The Sun RPC library[Sun88] is a collection of procedures which provide remote argument passing for certain builtin types, and a stub generator (rpcgen) while builds the RPC interface. rpcgen accepts a language that looks like restricted C declarator syntax, with a dash of Pascal-style variant records. If the user wishes to pass structures or other types which are not among the builtin types, s/he can declare these types in the XDR language, and argument passing functions will be automatically generated. The user must change the program interface to use rpcgen by only passing references (pointers in C) as arguments to possibly

remote functions. Sun's RPC protocol also requires that the procedure call complete before control is returned to the user.

On the SmallTalk front, Paul McCullough[McC87] has implemented something he calls "Transparent Forwarding." There is a striking similarity between the structure of his work and the work being presented here. In fact, there is almost a direct correspondence between the three classes he needed to add to SmallTalk and the three pieces which I will present as being necessary components of an interesting wrapper design. Perhaps this similarity is a direct result of the similarity of the initial goals, and specifically, the transparency requirement.

A GNU C++ wrapper should not be confused with Zeta-Lisp wrappers. Zeta-Lisp wrappers were a more primitive idea which evolved into *whoppers*, a mechanism very similar to GNU C++ wrappers. It is hoped that the reader will excuse the author from not adopting the *whopper* nomenclature.[Gum88]

An alternative which was not investigated was extending C++ to incorporate *reflection*. Reflection is a very powerful technique which, in essence, allows the system to encapsulate itself and then perform computation on that encapsulation. The inverse operation, reification, allows the system to then "become" that system which the reflection captured. Reflection and reification are mentioned here because they are interesting concepts, and because the wrapping of a function call is, in a very limited way, a reflective operation.

More detailed comparisons with related work will be expanded inline.

4. Implementation of Wrappers

In the ES-Kit kernel, parallelism is achieved by using "futures." When an application makes a RPC, the ES-Kit kernel intercepts the call, creates a new process which will carry out the computation, and returns to the caller a "future" (typically a more refined type, such as a future_int, a future_char_star, etc.). A future is essentially a reference to the result of the function call which will be available "some time in the future." As long as the caller does not attempt to evaluate the future, it will not block. When the caller does try to evaluate it, if the future has not yet completed, the kernel puts the caller to sleep, and wakes it up when the future is ready. If the future had completed by the time the caller attempted to evaluate it, the caller gets the value of the future, and proceeds normally. This model of parallelism is natural, largely transparent, and centers around being able to translate what look like normal function calls into requests for computation to be scheduled by the kernel.

As soon as a preliminary version of the GNU C++ compiler was available, it became evident that in order to make life bearable for application programmers, these requests for computation must be as easy to write as a normal function call. To make it easier to port "normal" C++ programs into the ES-Kit environment, the more these requests could resemble normal function calls, the better.

4.1. Introduction to Wrappers

The problem of implementing a transparent, user-definable RPC mechanism is handled by adding a construct called a *wrapper* to the compiler. A wrapper is a new syntactic construct that allows the programmer to specify an alternate

implementation of member function calls. An RPC handler can then just be seen as a special case of this. The basic idea behind a wrapper is the notion that in order to execute a member function call, it is necessary only to know the object for that call (which will become the this pointer), the member function for that object being called, and the arguments to that function. A wrapper abstracts procedure calls by having the compiler translate member function calls into encapsulations of member function calls, and passing these encapsulations to the wrapper.

First, an example of their syntax will be presented, followed by a description of their semantics.

```
struct foo
{
        foo ();                 // foo constructor
        ~foo ();        // foo destructor

        virtual ()foo (int (foo::*)(...), ...):    // a virtual wrapper

        int g (int);            // foo member function g
        virtual int h (int, int); // foo member function h
};
```

A wrapper has a very distinct syntactic form from operators, member functions, constructors and destructors. Initially, the idea was to pun wrappers with an overloaded operator (actually the combination of two overloaded operators, yielding operator->() ()). Bjarne Stroustrup suggested starting from a clean syntactic point, to avoid the kinds of ambiguity problems that such punning can pose. He proposed the syntactic form seen here, of placing empty parenthesis just in front of the declarator. This syntactic form permits one to specify many different ways of encapsulating (or executing) member function calls.

As can be seen from the declaration of the wrapper (which in this case happens to be virtual), the wrapper has, as its arguments, all the information it needs in order to wrap calls for either foo::g or foo::h. Wrappers are member functions, and hence calls to the wrapper have a this pointer. They take a pointer to member function argument which is initialized by the compiler to the value of the (possibly virtual) member function being wrapped. The rest of its argument list, which is left untyped for now, serves as the destination for the arguments of wrapped function.

In these examples of syntax, the wrapper "wraps" all function calls, but the user can specify whether to wrap the virtual or non-virtual function, and whether to wrap using a virtual or non-virtual wrapper:

```
main ()
{
        foo *a;

        a->g (100);             // wrap 'foo::g' with virtual wr.
        a->h (101, 102);        // wrap virtual 'h' with virtual wr.

        a->foo::h (103, 104); // wrap 'foo::h' with virtual wr.

        a->foo::() g (200);     // wrap 'foo::g' with foo's wr.
        a->foo::() h (201, 202);        // wrap virtual 'h' with foo's wr.
```

```
        a->foo::() foo::h (203, 204); // wrap 'foo::h' with foo's wr.
}
```

Now that ideas behind wrappers have been sketched, it is time to look at them a little more deeply.

Wrappers are intended to encapsulate, and not necessarily substitute for, calls to member functions. As a result, when a member function call is being compiled, the compiler performs all conversions necessary on the arguments to that member function. Once this has been done, but before the code for the call is emitted, the compiler then computes the address of the member function that is to be wrapped, and makes that address an additional argument at the head of the argument list. The wrapper is then called with the resulting argument list, again converting any arguments from this list to match the types specified by the wrapper.

4.2. Wrapper Issues

If all member functions of a class have the same type signature (numbers and types of arguments, and return values), then writing wrappers for them is a very simple task. Problems arise when one wishes to write a single wrapper to handle member functions with distinct type signatures. In such a situation, the different argument lists can be matched by using the C++ '...' notation, leaving the types and number of arguments to the wrapper unchecked. This makes putting function calls into wrappers very easy, but makes getting them back out rather difficult.

If wrappers were added to the language without any additional support, it might leave more loose ends dangling than it tied up:

(1) How does one keep a specific function call from being wrapped?

(2) How does one keep a particular member function from being wrapped?

(3) What happens when passing a derived member function to a base wrapper?

(4) How does one handle type conversion in an argument list full of '...'?

(5) Should constructors be wrapped?

(6) Should wrappers be recursive? (can a call to a wrapper be wrapped?)

(7) Should function calls within wrappers be wrapped? (where is the bottom?)

(8) How does one wrap a function call via a pointer-to-function (which may be missing type information)?

(9) How should return values be handled? (overload based on return types?)

Such a list looks long, but when looking at recent work on this problem (in LISP and SmallTalk, not just C++!), one sees that it looks familiar. After presenting our solutions to these problems for C++, a comparison will be made with McCullough's work on SmallTalk. The two are strikingly similar, both yielding extensions which are easily implemented, have low impact of the efficiency of the system as a whole, and provide great expressive power.

4.3. Not Wrapping

The first two issues bring up the question of escape hatches. No matter how wonderful a particular language feature is, there always appears to be some case where it is more desirable to get around it. Sometimes one wishes to defeat the feature at compile-time (such as calling a virtual function directly, rather than by the virtual function table), other times on wishes to defeat the feature at run-time (which is usually done with if-then-else control logic).

Distributed programs tend to have run-time behavior which is hard to predict at compile time. In the context of RPC, the main reason not to go through an RPC wrapper is because it is known (somehow ahead of time) that the call will be local, and should therefore not go through the extra levels type conversions. Because this locality information may be available at compile time or only at run time, the means of getting the information to the compiler should be general enough to produce an efficient solution in either case.

An attractive solution is to provide a *wrapper predicate*. A wrapper predicate is a binary operator, which takes an object and a member function pointer, and returns zero if the function should not be wrapped, and non-zero otherwise. An example of the syntax is given in section 4.4. The syntax is defined in section 5.1.

A default wrapper predicate always returns the value 1 (indicating that the function should be wrapped). Wrapper predicates are inherited, and may therefore be required to handle member functions which are not members of the class that the wrapper predicate belongs to. In that case, the wrapper predicate should provide a "default" case (which normally would return a non-zero result).

Wrapper predicates, when they are necessary, can be defined to be inline, and using standard constant-folding techniques, compile-time decisions can be made based on their otherwise run-time semantics. This is not an unreasonable expectation of a C++ compiler, since such constant folding is already implemented in the C++ compilers from AT&T and GNU.

Wrapper predicates provide both run-time and compile-time escape hatches for wrapper users. They also seriously affect encapsulation (in the same way that declaring a private function public in a derived class definition does), and should probably be used only when necessary.

4.4. Making Wrappers Generic

Typically, the user would like all RPCs to be handled in a uniform fashion (much as normal procedure calls are now). To that end, it is desirable to specify one wrapper which will uniformly handle all procedure calls. In a strongly-typed world, this is not always possible. The implicit conversion of a pointer to an object to a pointer to a base class of that object is anti-symmetric with respect to "pointers to member functions." That is to say, we can only reliably convert "pointers to member functions" from a "pointer to base class member function" to a "pointer to derived class member function." It is therefore not type-safe to pass a pointer to a derived member function to a wrapper in a base class, as can be seen in the following example:

```
struct B { virtual f (); ... };
struct D : public B { virtual f (); virtual g (); ... };
```

```
int foo (B*, int (B::*)());
int bar (D*, int (D::*)());

int surprise ()
{
        B bb; D dd;

        foo (&bb, &B::f);       // ok
        bar (&bb, &B::f);       // error: B* does not convert to D*

        // ok! f is in base class
        foo (&dd, &D::f);

        // error! int (D::*)() does not convert to int (B::*)()
        foo (&dd, &D::g);
}
```

The real problem with passing a pointer to a derived member function to a wrapper expecting a pointer to a base member function is that type information is lost. Fortunately, this problem can be solved with minimal overhead. If this and the pointer to member function argument (call it pmf) are composed within a wrapper, the compiler can know that combination to be type-correct. This is a direct result of how the wrapper is called. An assignment to this and/or pmf, or an implicit conversion of pmf (including the implicit conversion that occurs when pmf is composed with another object, or further passed as an argument) means that typewise, all bets are off. The compiler could insert code to test whether the composition is indeed valid, relying on a run-time encoding scheme based on information held in the virtual function table, or it could just simply disallow such things from happening. It can also prevent assignments to this and the pointer to member function argument within wrappers.

Passing pointers to member functions against the inheritance hierarchy grain is the first (and by far the easiest) problem to handle. The second problem to handle is making a single wrapper wrap all member functions, regardless of their argument lists. Wrappers are supposed to encapsulate a very simple, very powerful idea, and it would be nice to specify the implementation of that idea as concisely as possible. If a different wrapper must be declared for every different function, the whole purpose of wrappers is defeated.

All argument lists can be matched by declaring a wrapper with an untyped argument list, i.e., '...'. However, this argument list may need to maintain type information until the arguments can get to the ultimate target of the procedure call. To accomplish this, we allow wrappers to specify a *typing function*. Types are not values in C++, so it is not possible to specify the typing function directly. However, the compiler can generate such a function which is generally useful in the two most important cases. The default typing function is the mapping of types according to standard C promotion rules. All C and C++ compilers have such a function built into them. A user-defined typing function can be synthesized by specifying the result type that all arguments should be converted to if the value appears within the untyped part of a function's argument list. Such a result type is called a *synthetic type*.

A synthetic type should be designed in such a way that it can ravel and unravel any kind of type that may be passed as an argument to (or from) a wrapper. The compiler enforces such rules in a lazy fashion, i.e., when a type conversion is impossible, the user will get an error message. It does not tell the user whether his synthetic type is *a priori* complete.

Because C++ is primarily a statically typed language, there is not a great deal of support for encapsulating type information in a representation that can be stored, retrieved, and used to reconstruct a well-typed value at runtime. This restriction also currently applied to types which appear to be run-time dependent, but can actually be computed statically from the program's control graph. Although not well supported by C++, most distributed systems implemented in C++ (at least all that this author has seen) implement some kind of type which performs this magic by hand, usually with some extension to the compiler. Fortunately, one such group taking a look at such extensions is Dr. Stroustrup's. His work on parametric types will be very influential on (and will hopefully obviate the need for) synthetic types. In the mean time, here is an example of a synthetic type:

```
struct DT               // a synthetic type
{
        DT (int);               // encode an int
        operator int ();        // decode an int

        DT (double);            // encode a double
        operator double ();     // decode a double

        DT (void);              // encode a void type!
};

struct Base
{
        // wrapper predicate (from section 4.3):
        int ()?Base (int (Base::*)(...));

        // wrapper w/ synthetic type:
        DT (DT...)Base (int (Base::*)(...). ...);
}
```

It is important to note the distinction between wrappers and stub generation. Wrappers (and wrapper predicates and synthetic types) do not themselves comprise a stub generator. However, they can be used to greatly simplify the interface to one.

4.5. What Should Be Wrapped?

Until one has implemented a C++ compiler, it is difficult to appreciate what a typed value is in C++. Object construction is a process of type elevation. Starting with an untyped chunk of memory, a base constructor renders it typed with a base type. Such an instantiation may involve the initialization an object's virtual function table pointer, and initialization of some of its members. After the objects base types have been initialized, each type in the lattice between the base types and the

final type of the object will be initialized. It is not until this process of initialization has been completed that the object can say "I am." Because of this, it makes no sense to have a wrapper wrap calls to constructors, since the object has not been completed, just as it makes no sense to stop initializing once an object has an operator= in its scope, and start using assignment semantics for the rest of the operation.

Suppose a user wished to define her/his own wrapper for purposes of either augmenting and/or reimplementing wrappers derived from base classes. Should that wrapper be wrapped by those from the base class, or should the wrapping stop at the first wrapper? This is indeed a very hard question, since it essentially forces the distinction between wrappers as an initialization of function-call objects, in which case a recursive interpretation might be nice, and wrappers as an alternate function-call implementation (or operator), in which case the first applicable wrapper applies, and nothing more.

The wrapper extension is a large one, and one which should be evaluated carefully. To keep things manageable, the latter interpretation is the one which has been implemented. If justifiable needs for the former develop, it would not be hard handle them with a new, appropriate syntactic form.

4.6. Dealing With Return Values

Return values are a necessary evil in C++. Every machine (and almost every compiler) handles return values in its own way. To make matters worse, return values may come back in a register, or a set of registers, or sit atop the stack, or not be returned at all, but may "fill in" an address handed to them by the caller; these variations may all exist within the same compiler!

The C++ programming language does not allow overloading based upon return type, and furthermore does not allow typeless return values (as it does allow for its argument types). While these restrictions may seem unnecessary (and even an annoying obstacle) for the task at hand, it is well to adhere to them nonetheless. Portability reasons alone make it impossible to do otherwise: because a compiler is required to do nothing more than create a coherent call/return interface, it is not possible to predict ahead of time how many kinds of return values a wrapper will need to implement.

A wrapper, therefore, is limited to accepting member functions which return a certain type, requiring that new wrappers be written for functions that return new types. Because of this, recursive wrappers are not necessarily desirable, since wrappers of base types cannot wrap functions which return objects of derived types. There is, however, no restriction that a wrapper must return the same type as that of the function it is wrapping. Note that a wrapper need not return the same type as the function that it wraps. Also note that the following two wrappers are distinguishable as having different argument lists, and does not constitute (to first approximation) overloading based on return type:

```
int    X::()X(int    (X::*)(...), ...);
double X::()X(double (X::*)(...), ...);
```

5. Syntax and Semantics of Wrappers

5.1. Syntax

The syntax of wrappers is somewhat baroque, in keeping with C's (and consequently C++'s) notorious declarator syntax. A wrapper for class T has the declarator name ()T. Its return value, argument list, virtual attributes, etc., are declared just as they would be for normal member functions. The first argument in the wrapper must be of type pointer to member function type, or type void *.

A wrapper predicate for type T has the declarator name ()?T. It takes only one argument, which must be of type pointer to member function type. It return type is integer.

A synthetic type T2 is associated with a wrapper for type T by placing it in its wrapper parenthesis followed by '...', i.e., (T2...)T. The type T2 must be a user-defined type, and it must be declared before use. Naturally, for this type to be of any use, it must have at least one constructor.

5.2. Semantics

A wrapper is applied as follows:

Given a class B with a wrapper W, a class D with a member function F, and given that D derives from B, P is a pointer to type D, and F is not a constructor or a wrapper, then W may wrap the call to P->F (...) under the following conditions:

(1) there is no wrapper predicate P in any class between B and D in the type lattice, or

(2) the wrapper predicate P returns a non-zero result given arguments O and F.

Under no other conditions would an attempt be made to wrap the call to F.

Wrapping takes place as follows: all arguments to the original function are converted according to the argument list of F (as per normal C++ (and ANSI C) semantics). If the wrapper or the wrapper predicate do not satisfy normal C++ visibility rules, it is an error. If D derives from multiple base classes B1 and B2, and both provide a wrapper, an ambiguity error is reported; similarly for wrapper predicates. If there are no visibility or ambiguity problems, a function call is wrapped by prepending a pointer to the member function being wrapped to the argument list of the function, and passing the resulting argument string to the wrapper. The wrapper treats the argument list as a normal function call would, with the exception of its treatment of arguments which the wrapper converts to a synthetic type.

If a synthetic type is specified for a wrapper, then

(1) it must be unique, and

(2) it must convert any argument passed to it, and

(3) all conversions must satisfy normal C++ visibility rules, and

(4) all parameters which are not explicitly typed by the wrapper are converted to the type of the synthetic type, and

(5) argument lists which contain synthetic types as a result of converting arguments are terminated with a synthetic type encoding the void type.

It should be stressed again that synthetic types are a provisional measure until a determination can be made as to how to (better) use parametric types, if they are added to C++.

5.3. Implementation notes

The representation of pointers to member functions must be done in a host-independent manner. By filling the virtual function table with all member functions available to a given class, and distributing a class's virtual function table to each node which could host an instance of that class, a member function's address can be represented by its index in the (extended) virtual function table. This index is, of course, host-independent.

Computing virtual function tables which permit explicit non-virtual function calls is somewhat tricky. This is because an object of derived type must be able to make explicit calls to member functions from either the derived or base classes. Phillipe Gautron solved this problem with a simple algorithm [Gau87], and this algorithm is incorporated in the GNU C++ compiler.

It is possible for a function to be called with a variable number of arguments. Unfortunately, it is not also possible to build an argument list of variable size. That is, one can make two calls to the same function, one call with five arguments and another call with seven arguments, but each of the calls has a fixed number of arguments (five and seven, respectively). For this reason, if a wrapper is to call a function with the arguments it receives, and if it does not know (at compile time) how many arguments with which to call this function, it is constrained to build an argument buffer, and let the receiver of that buffer format a proper call stack.

A solution to this problem which would be the extension of varargs. The varargs construct provides C programmer with the ability to process arguments from an argument list, where the arguments have unknown size and type. These arguments are accessed by a pseudo-function va_arg which takes a pointer to the variable argument declaration and the type of the argument to get. Functions which use varargs usually have some means of detecting when all arguments have been processed. If one had pseudo-functions va_push and va_call, then one could use the normal call stack (or register window parameter convention, or whatever) to pass a variable number of arguments. This would be most useful when passing an unknown number of arguments, all of the same type. Such an extension is probably better left to the compiler. No extension of this kind has been implemented in GNU C++ as of this time.

The GNU C++ compiler, for historical reasons, also prepends the length of the argument list to the parameter list, so that pure value-passed argument lists can be copied into message buffers efficiently. This argument is treated as a visible "invisible argument," hence a wrapper specified in GNU C++ look like

```
class X { protected: int ()X (int arglen, int (X::*)(...), ...); ... };
```

When this hack is no longer needed, it will be removed.

5.4. Comparison with a SmallTalk model

The solution to the RPC problem using wrappers bears a striking resemblance to work done by Paul McCullough at Xerox PARC to add similar (transparent) features to SmallTalk.

His model starts out with a proxy class which implements only one method, the universal method **doesNotUnderstand:**. This would correspond to the ES-Kit approach of implementing a future which provides a wrapper. Transparency is achieved because in both cases, all messages that involve the proxy (future) object are trapped by that object.

To provide better performance (and provide greater flexibility) McCullough has something called a **PolicyMaker**, which decides whether and what should be sent where. A wrapper predicate provides almost the same functionality.

Finally, when an RPC is to be made in McCullough's extended SmallTalk, objects are sent via a **TransporterRoom** object, which takes care of communication protocols between machines, as well at the linearization of messages and object. This is clearly analogous to the possible behavior of synthetic types.

Other changes he needed to make, for example, to have his proxy objects respond to *some* messages (like a **PrintYou!** method from a debugger), or to have them interact with SmallTalk's == operator, are language dependent implementation details. In fact, he had to reimplement the primitive == operator (which was hard-coded for efficiency reasons) so that == could be used to test for equality of remote objects via proxy. Because C++ is a compiled language, overloading operator== has no impact on efficiency, and consequently no extensions for this need to be made.

6. Other Uses For Wrappers

It was claimed at the outset of the paper that wrappers were a general extension to the C++ programming language. Besides RPC, to what other use, then, can they be put? This question is answered by looking at other systems which make the function-call abstraction a first-class item, and then making the obvious connections.

6.1. before: and after: methods

It is occasionally desirable to specify function prologue code, or function epilogue code, or both, for a class of functions. For example, a program profiler might start each function being profiled with code to call a profiling routine, and code at the end of the function to report that function has finished.

Instrumentation and debugging information is very often specified as things which must be done before and/or after a function is executed. The Zeta-Lisp system (and other Common Lisp systems, no doubt), provide before: and after:

hooks.[2] Wrappers provide the programmer with a means of executing hooks without changing the code of the function being executed. For example, here is a wrapper which profiles how often certain functions are executed:

```
inline void ()X(void (X::*pmf)(int, int), int i, int j)
{
        start_call (pmf);       // ::start_call or (*this->start_call)
        (*pmf)(i, j);
        finish_call (pmf);      // ::finish_call or (*this->finish_call)
}
```

This example shows that wrappers can handle cases as general as any handled by before: and after: methods, since the wrapper can execute its own, class-specific, or instance-specific before: and after: methods.

One advantage of having before: and after: methods is that they are distinct from the function call, so that the problems of building the called functions argument list within the wrapper need not be faced.

One could use a wrapper predicate as a before: hook, and always return 0, causing the wrapper predicate's code to be executed without going through a wrapper.

6.2. Memoization

Memoization is a technique which can save computation. For example, if one knows that a particular function is functional (it contains and modifies no state), then one can be assured that repeated calls to that function with the same parameters will yield the same results. This can be very useful when a particular task must be performed at some unknown point in a program, and its result used subsequently.

Memoization is implemented in the GNU C++ compiler. The instruction recognizer assigns an integer instruction code value to an instruction based on the (tree) pattern of that instruction. The instruction code is needed in several different places within the compiler, but none of those places can be sure that the results have yet been computed. If the compiler were to eagerly compute these codes (as soon as the instruction was generated), its work might be wasted, since the instruction may later be optimized: a three-address add instruction may become a two-address add instruction, and later even just a move, or be optimized away completely! It is therefore undesirable to try to recognize an instruction until its code value is needed. A function called recog_memoized recognizes the instruction, and *memoizes* the result. The result remains valid unless the instruction is changed (i.e., by the optimizer). Given this implementation, an instruction is recognized only as many times as it needs to be. The small overhead of memoization is more than offset by the savings of not having to recognize the same instruction more than once.

Simulators are also a good example of an application which can benefit from memoization. Simulators tend to have large "functional" units; computing the

[2] A *hook* is a slot in a program where one can deposit a function to be executed.

value for one such unit may be expensive, but that value may not change if the stimuli for that unit do not change (or vary more than a certain amount).

Wrappers provide a natural means to memoize functions. When a user has isolated a function (or set of functions) which appear to be good candidates for memoization, s/he can define a wrapper, and have the wrapper implement the lookup/execute machinery of memoization. The memoization can be enabled or disabled without interfering with the original function (being memoized) in the least. An example of a memoized factorial function is distributed with the GNU C++ library.

Wrappers can also be used to look for such functions: a memoization table can be constructed, and at the end of the run, statistics on how stimuli variation affected the "memoized" functions can be printed. C++ is already used in many simulation systems; adding memoizing wrappers to these existing platforms will preserve encapsulation, while possibly improving performance.

6.3. Synchronization

Remote procedure calls are useful for implementing formalisms in distributed systems other than merely making procedure calls. For example, in Grass [Gra86], a synchronization mechanism called *mediators* is presented. In the section of the paper describing possible implementations, RPCs are used so that multiple clients can exchange messages with a single mediator. As the paper states "In the perception of the client process, a remote procedure call appears to be no different than a simple local procedure call."

A provider of resources can implement a synchronization mechanism inside of a wrapper, and have all calls which depend on these resources go through this wrapper. All such resources can then be managed in a consistent manner by a single wrapper, simplifying maintenance. When a new resource allocation procedure is added, the provider need only concern herself/himself with the interface to the resource, and not with the interface to the synchronization mechanism. The wrapper automatically takes care of that.

7. Possibilities for future work

7.1. Handling Return Types

The handling of return types is currently very unclean. While a well-designed synthetic type can allow wrappers to handle any argument type which may appear in the argument list of a wrapper, a single wrapper cannot handle a task as simple as wrapping both a function which return an int and a function which returns a double.

This problem can be solved by making a reference to the return value available as one of the parameters of the wrapper. The compiler could choose whether to pass the wrapper a reference to the object being initialized by the function call, or a reference to a temporary, which could then be copied to presumably register-resident object. Such a choice would make initialization by wrappers essentially as efficient as returns from normal functions. This has the benefit of allowing the implementor of the synthetic type to take advantage of building the return value

where it will ultimately end up, rather than in a temporary buffer which will have to be copied. C++ is good about letting class designers be smart about initializing arguments, but not about coordinating return values.

7.2. Types as Values

When a call to a function f gets wrapped, arguments are first converted to the types which f expects, then to the types that the wrapper expects. The wrapper may cause f to be executed, in which case the arguments that came into the wrapper must eventually be re-converted to what f expects. All this converting can be inefficient, especially if it is not desirable.

An *inline wrapper* may convert arguments directly to the type the wrapper expects (bypassing f's conversions completely). These arguments can then be passed the function being wrapper (or any other function) using the type information of the function being called. An inline wrapper permits parameters of a wrapped call to undergo only two conversions instead of three.

John Rose presents a similar idea [Ros88], in a much more general framework. The greatest problem with wrappers is the amount of type information one must give up in order to encapsulate, within the language, something as general as user-defined function calls. If there were some reasonable way for the user to compute more directly with types, a lot of the complexity of designing a wrapper, a wrapper predicate, and synthetic types would go away.

It also appears that Bjarne Stroustrup's work on a implementation of parametric types for C++ may help accomplish the same goal. Any extensions made with respect to wrappers must be especially careful with regard to possible implementations of parametric types. For example, it is even possible that if parametric types are expressive enough, wrappers could become an operator like '&', a purely polymorphic operator which is applied by the compiler in certain cases. Rules for expressions of this type (type wrapper-of-function?) can then be implemented by parametric type rules.

7.3. Wrappers for Constructors

The SOS project [Sha87] provides an execution environment with dynamic linking. The dynamic linking extensions which they have added to C++ make it possible to migrate objects from one machine to another and/or from one address space to another. It is implemented by having all member function calls (including calls to constructors!) go through an extended virtual function table. When a new object instance is created, a call is made to find (dynamically loading if necessary) the extended virtual function table the object will need. The object is allocated, and the constructor is called with the object, the virtual function table (as an extra hidden parameter), and whatever other arguments the call to that constructor specifies.

Since the call to the constructor is made via the virtual function table, and the virtual function table is not *a priori* linked with the application, the code for the constructor does not need to be linked until an object of that type is actually allocated. At that time, all member functions that object needs are linked in.

If it were possible to wrap calls to constructors, then the special function calls needed to find the virtual function table, pass the virtual function table pointer to the constructor, and the other vagaries of the system could be handled without the need to make more specific compiler hacks.

8. Conclusion

Wrappers are an attempt to add flexibility to C++ where it is desperately needed, while incurring minimal (possibly zero) runtime overhead. Wrappers provide a very nice encapsulation of procedure-oriented computation, making it easier to write self-scheduling parallel programs, given the right environment. They can also be used to profile functions in a user-definable way, which can be useful for finding optimization opportunities, such as memoization.

The current implementation of wrappers suffers from the lack of certain primitives within the C++ language, notably the lack of an ability to construct and manipulate parameter lists as first-class objects. It is hoped that the extensions presented in this paper will help to further clarify more precisely what primitives are lacking, and provide a context in which to implement them usefully.

This paper has presented a number of possible extensions to the C++ programming language. While some of these extensions are implemented in the GNU C++ compiler, many are not, and some perhaps should not be. A language which changes every day is no more useful than a language which never changes — a sensible compromise must be sought which allows the language to evolve while remaining stable enough to be usable.

Wrappers are not a new idea, at least not any more so than object-oriented programming is new. Both have been around for a while, in various guises. A comparison between a C++ implementation and a SmallTalk implementation of transparent RPC showed the ideas underlying wrappers useful in both cases, and even showed that their solutions were structurally very similar.

This paper presents implementation issues and strategies which make wrappers a reasonable extension to the C++ programming language, and other issues which show that there is still work to be done. Wrappers have been implemented in GNU C++; however, their implementation should be expected to change if related language features, such as parametric types, simplify or generalize their use.

Acknowledgments

This work was funded in part by MCC, the MCC Experimental Systems Project, DARPA (contract number MDA972-88-C-0013). The Institut National de Recherche en Informatique et en Automatique (INRIA) also hosted me during the time much of this paper was being written. This paper reflects the opinions of the author, and does not necessarily represent the views of the US Government, MCC, or the staff of the ES-Kit project.

Bill Leddy was the principle designer of the ES-Kit kernel; his needs precipitated most of these ideas. Wayne Allen, Gumby Wallace, and Jon Shopiro listened well to early ideas concerning wrappers, and offered many useful suggestions. Bjarne Stroustrup coined the term "wrapper," and provided me with a syntax, and suggested that wrappers might be worthwhile. Doug Lea and John Rose

contributed ideas and inspirations which have been woven into this paper. Phillipe Gautron pointed out the need to encode non-virtual functions in special ways in virtual function tables, and also showed me his algorithm for doing that. Marc Shapiro got out the red pencil when I needed a reviewer. Most of all, I would like to thank Richard Stallman and contributors to the GNU project, for sharing their software with me.

Bibliography

[Cal87] Call, Lisa A., Cohrs, David L., and Miller, Barton P. "CLAM: an Open System for Graphical User Interfaces." OOPSLA 87 Proceedings. October, 1987.

[Cap87] Caplinger, Michael. "An Information System Based on Distributed Objects." OOPSLA 87 Proceedings. October, 1987.

[Gau87] Gautron, P., and Shapiro, M., "Two extensions to C++: A Dynamic Link Editor and Inner Data." Proceedings and Additional Papers from the First USENIX C++ Workshop. Santa Fe, New Mexico, November 9-10, 1987.

[Gra86] Grass, J.E., and Campbell, R.H., "Mediators: A Synchronization Mechanism." Proceedings from the 6th International Conference on Distributed Computing Systems. May 1986.

[Gri87] Grimshaw, Andrew S., and Liu, Jane W. S. "Mentant: An Object-Oriented Macro Data Flow System." OOPSLA 87 Proceedings. October, 1987.

[Gum88] Gumby Wallace, Private communication.

[Hal85] Halstead, R., "Multilisp: A language for concurrent symbolic computation." ACM Transactions on Programming Languages and Systems 4, 4 (October 1985).

[Led88] Leddy, B., "ES-Kit Kernel Release 1 Design Notes." MCC Technical Report number ACA-ESP-141-88.

[Lin87] Linton, Mark. "The Design of the Allegro Programming Environment." Proceedings and Additional Papers from the First USENIX C++ Workshop. Santa Fe, New Mexico, November 9-10, 1987.

[Lis88] Liskov, B., and Shrira, L., "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems." Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation. June, 1988.

[Mae87] Maes, P., "Concepts and Experiments in Computational Reflection." OOPSLA 87 Proceedings. October, 1987.

[McC87] McCullough, P., "Transparent Forwarding: First Steps." OOPSLA 87 Proceedings. October, 1987.

[Ros88] Rose, J., "Refined Types: Highly Differentiated Type Systems and Their Use in the Design of Intermediate Languages." Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation. June, 1988.

[Smi88] Smith, Robert J. II, "Experimental System Building Blocks." MCC Technical Report number ACA-ESP-102-88.

[Sta88] Stallman, R., "Internals of GNU CC." (Last updated version 1.25). Free Software Foundation. Cambridge, Massachusetts, 1988.

[Sun88] Sun Microsystems, "Network Programming." Part Number 800-1779-10. Revision A, May 9, 1988.

[Tie88] Tiemann, M., "User's Guide to GNU C++." MCC Technical Report number ACA-ESP-099-88.

USENIX Association Services and Benefits

The USENIX Association is a not-for-profit organization of individuals and institutions with an interest in UNIX and UNIX-like systems and the C programming language. It is dedicated to fostering the development and communication of research and technological information and ideas pertaining to UNIX and UNIX-related systems. The Association sponsors workshops and semiannual technical meetings, produces and distributes a bimonthly newsletter, *;login:*; publishes a quarterly technical journal, *Computing Systems*; and serves as coordinator of a software exchange via its "Software Distribution Tapes."

The Association was formed in 1975 and incorporated in 1980 to meet the needs of UNIX users and system maintainers who met periodically to discuss problems and exchange ideas concerning UNIX. It is governed by a Board of Directors elected biennially.

The USENIX Association offers several services to its members:

- Mailing of the newsletter *;login:*;
- Mailing of the technical journal *Computing Systems*;
- Offering of various UNIX publications and technical information for purchase;
- Presentation of technical meetings twice a year and single-topic workshops periodically;
- A discount on the meeting registration fee;
- The right to order 4.3BSD UNIX Manuals;
- The right to vote on matters affecting the Association, its bylaws, and in the election of its directors and officers.

STUDENT MEMBERSHIP $15

Open to any full-time student at an accredited educational institution. A copy of your student I.D. card must be provided.

INDIVIDUAL MEMBERSHIP $40

Open to any individual or institution. Individual Members may vote; however, they do not automatically receive the Distribution Tapes or other services requiring UNIX license verification.

USENIX Association Membership Application

Membership is by Calendar Year

Please type or print

[ ] New [ ] Renewal

Name: _____

Address: _____

_____

_____

_____

Phone: _____

uucp network address: *uunet!* _____

Individual, Corporate, and Supporting categories are all open to either institutions or individuals. Membership fees are:

[ ] $ 40 Individual

[ ] $ 15 Student (full-time)
With copy of student I.D. card

[ ] $ 275 Corporate

[ ] $125 Educational Institution

[ ] $1000 Supporting

[ ] Check enclosed: $ _____
Payments must be in US dollars payable on a US bank

[ ] Purchase order enclosed; invoice required

[ ] Check if you do NOT want your name and address made available to other members.

[ ] Check if you do NOT want your name and address made available for commercial mailings.

Please complete and return this form with your purchase order or payment to:

USENIX Association
P. O. Box 2299
Berkeley, CA 94710

For Office Use

Inst: .....

Mem#: ..... Check#: .....

Lic: ..... Rf: .....

Date: ..... Db: .....