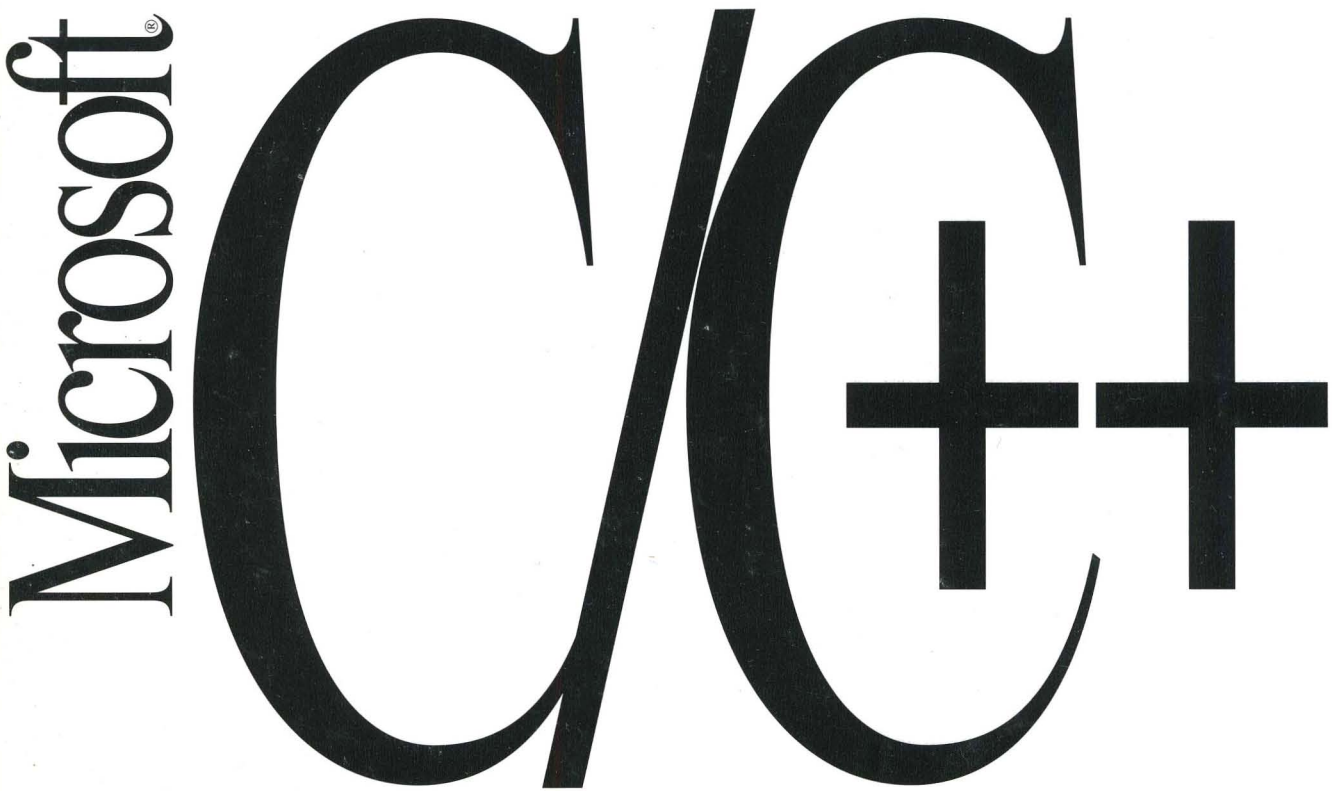
C++ Language Reference



Microsoft® C/C++

Version 7.0

C++ Language Reference

For MS-DOS® and Windows™ Operating Systems

Microsoft Corporation

Document No. LN24772-91

10 9 8 7 6 5 4 3 2 1

Contents Overview

Contents

Figures and Tables

Figures

Tables

Introduction

This manual explains the C++ programming language as it is implemented in Microsoft C++ version 7.0. Microsoft C++ is based on *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup (the ANSI base document for C++). For information about the Microsoft C compiler, see the *C Language Reference* manual.

If you are new to C++, you might learn more quickly by starting with the *C++ Tutorial* manual.

Note Microsoft documentation uses the term "DOS" to refer to both the MS-DOS® and IBM Personal Computer DOS operating systems. The name of a specific operating system is used to note features unique to that system.

Scope and Organization of This Manual

C++, like C, is a language that is heavily reliant on a rich set of library functions to provide the following:

- Portable operating-system interface (file and screen I/O)
- String and buffer manipulation
- Floating-point math transformations
- Character classification information
- Other supporting functionality

For information about the run-time library functions, see the *Run-Time Library Reference* manual. For information about the Microsoft Foundation class library or the iostream classes, see the *Class Libraries Reference* manual.

This manual is intended for programmers who have already learned the fundamentals of C++ programming; it is not intended as a learning guide. For information about learning C++, see the *C++ Tutorial.*

This manual is organized as follows:

Chapter 1, "Lexical Conventions," introduces the fundamental elements of a C++ program, as they are meaningful to the compiler. These elements, called "lexical

elements," are used to construct statements, definitions, declarations, and so on, which are used to construct complete programs.

Chapter 2, "Basic Concepts," explains concepts such as scope, linkage, program startup and termination, storage classes, and types. These concepts are key to understanding C++. Terminology used in this book is also introduced in this chapter.

Chapter 3, "Standard Conversions," describes the type conversions the compiler performs between built-in, or "fundamental," types. It also explains how the compiler performs conversions among pointer, reference, and pointer-to-member types.

Chapter 4, "Expressions," describes C++ expressions—sequences of operators and operands that are used for computing values, designating objects or functions, or generating other side effects.

Chapter 5, "Statements," explains the C++ program elements that control how, and in what order, programs are executed. Among the statements covered are expression statements, compound statements, selection statements, iteration statements, jump statements, declaration statements, and null statements.

Chapter 6, "Declarations," is one of three chapters devoted to how complete declarations are used to form declaration statements. This chapter describes such topics as storage-class specifiers, function definitions, initializations, enumerations, **class**, **struct**, and **union** declarations, and **typedef** declarations. Related information can be found in Chapter 7, "Declarators," and Appendix B, "Microsoft-Specific Modifiers."

Chapter 7, "Declarators," explains the portion of a declaration statement that names an object, type, or function.

Chapter 8, "Classes," introduces C++ classes. C++ treats an object declared with the **class**, **struct**, or **union** keyword as a class type. This chapter explains how to use these class types.

Chapter 9, "Derived Classes," covers the details of inheritance—a process by which you can define a new type as having all the attributes of an existing type, plus any new attributes you add.

Chapter 10, "Member-Access Control," explains how you can control access to class members. Use of access-control specifiers can help produce more robust code because you can limit the number of ways an object's state can be changed.

Chapter 11, "Special Member Functions," describes special functions unique to class types. These special functions perform initialization (constructor functions), cleanup (destructor functions), and conversions. This chapter also describes the **new** and **delete** operators, which are used for dynamic memory allocation.

Chapter 12, "Overloading," explains a C++ feature that allows you to define a group of functions with the same name but different arguments. Which function in the group is called depends on the argument list in the actual function call. In addition, this chapter covers overloaded operators, a mechanism for defining your own behavior for C++ operators.

Chapter 13, "Preprocessing," describes the C++ preprocessor, as well as the pragmas recognized by Microsoft C++.

Appendix A, "Phases of Translation," explains in what order a C++ program is translated from source code to an executable file.

Appendix B, "Microsoft-Specific Modifiers," describes the modifiers specific to Microsoft C++. These modifiers control memory addressing, calling conventions, and so on.

Appendix C, "Grammar Summary," is a summary of the C++ grammar with the Microsoft extensions. Portions of this grammar are shown throughout this manual in "Syntax" sections.

Document Conventions

This book uses the following typographic conventions:

Example	Description
STDIO.H	Uppercase letters indicate filenames, segment names, registers, and terms used at the operating-system command level.
char, **_setcolor**, **__far**	Bold type indicates C and C++ keywords, operators, language-specific characters, and library routines. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown.
	Many functions and constants begin with either a single or double underscore. These are part of the name and are mandatory. For example, to have the **__cplusplus** manifest constant be recognized by the compiler, you must enter the leading double underscore.
expression	Words in italic indicate placeholders for information you must supply, such as a filename.
grammar-element$_{opt}$	The $_{opt}$ subscript indicates that this element of the grammar is optional and can be omitted.

[[*option*]]	Items inside double square brackets are optional.
#pragma pack {1 | 2}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of these items unless double square brackets ([[ ]]) surround the braces.
`#include <io.h>`	This font is used for examples, user input, program output, and error messages in text.
CL [[*option...*]] *file...*	Three dots (an ellipsis) following an item indicate that more items having the same form may appear.

`while()`
`{`
`.`
`.`
`.`
`}`
	A column or row of three dots tells you that part of an example program has been intentionally omitted.

CTRL+ENTER	Small capital letters are used to indicate the names of keys on the keyboard. When you see a plus sign (+) between two key names, you should hold down the first key while pressing the second. The carriage-return key, sometimes marked as a bent arrow on the keyboard, is called ENTER.
"argument"	Quotation marks enclose a new term the first time it is defined in text.
`"C string"`	Some language constructs, such as strings, require quotation marks. Quotation marks required by the language have the form `" "` and `' '` rather than " " and ' '.
Color Graphics Adapter (CGA)	The first time an acronym is used, it is usually spelled out.

Microsoft Specific

This manual documents the C++ language, as it is implemented in Microsoft C/C++ version 7.0. As a result, some of the features of C++ that are implementation dependent or undefined in the ANSI base document are defined by the Microsoft implementation. You can find these features by looking for the "Microsoft Specific" heading in the left margin.

32-Bit Specific

Your version of Microsoft C/C++ may have the capability of generating 32-bit flat-model code. Such compilations may differ slightly from compilations targeting 16-bit computers.

When a particular feature is specific only to 32-bit compilations, it is marked by the "32-Bit Specific" heading in the left margin. In text, such compilations are referred to as "32-bit target compilations."

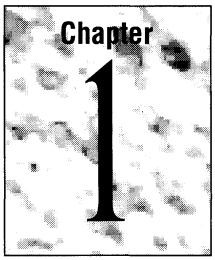
Special Terminology

In this manual, the term "argument" refers to the entity that is passed to a function. In some cases, it is modified by "actual" or "formal," which mean the argument specified in the function call and the argument specified in the function header, respectively.

The term "variable" refers to a simple C-type data object. The term "object" refers to both C++ objects and variables; it is an inclusive term.

For more information on terminology used in this manual, see "Terms" in Chapter 2, on page 25.

Lexical Conventions



This chapter introduces the fundamental elements of a C++ program, as they are meaningful to the compiler. These elements, called "lexical elements," are used to construct statements, definitions, declarations, and so on, which are used to construct complete programs. These elements are:

- Tokens
- Comments
- Identifiers
- C++ keywords
- Punctuators
- Operators
- Literals

Although the C++ operators are summarized in this chapter, a complete discussion of operators is deferred until Chapter 4, "Expressions."

C++ programs, like C programs, consist of one or more files. Each of these files is translated in the following conceptual order (the actual order follows the "as if" rule: translation must occur as if these steps had been followed):

1. Lexical tokenizing. In this translation phase, character mapping and trigraph processing, line splicing, and tokenization are performed.
2. Preprocessing. This translation phase brings in ancillary source files referenced by **#include** directives, handles "stringizing" and "charizing" directives, and performs token pasting and macro expansion (see Chapter 13, "Preprocessing," for more information about preprocessor behavior). The result of the preprocessing phase is a sequence of "tokens," which, taken together, defines a "translation unit."

   Preprocessor directives always begin with the number-sign (**#**) character (that is, the first non-white-space character on the line must be a number sign). Only one preprocessor directive can appear on a given line. For example:

```
#include <iostream.h>  // Include text of iostream.h in
                       //  translation unit.
#define NDEBUG         // Define NDEBUG (NDEBUG contains empty
                       //  text string).
```

3. Code generation. This translation phase uses the tokens generated in the preprocessing phase to generate object code.

   During this phase, syntactic and semantic checking of the source code is performed.

See Appendix A, "Phases of Translation," for more specific information about how a source program is translated.

Note The C++ preprocessor is a strict superset of the ANSI C preprocessor. It differs in its support for the single-line comment, its definition of the **__cplusplus** constant, and in its support of the C++ operators:

- `.*`
- `->*`
- `::`

(For more information about these operators, see "Operators" on page 10 and Chapter 4, "Expressions"; for more information about comments, see "Comments" on page 3.)

1.1 Tokens

A token is the smallest element of a C++ program that is meaningful to the compiler. The C++ parser recognizes these kinds of tokens: identifiers, keywords, literals, operators, and other separators. A stream of these tokens makes up a translation unit.

Tokens are most commonly separated by "white space." White space can be one or more:

- Blanks
- Horizontal or vertical tabs
- New lines
- Formfeeds
- Comments

Syntax

token:
keyword
identifier
constant
operator
punctuator

preprocessing-token:
header-name
identifier
pp-number
character-constant
string-literal
operator
punctuator
each non-white-space character that cannot be one of the above

The parser separates tokens out of the input stream by creating the longest token possible using the input characters. Consider the following code fragment:

```
a = i+++j;
```

The intention of the programmer who wrote the code might have been one of the following:

- Preincrement `j`, add the values of `i` and `j`, and assign the sum to `a` (where the tokens are `i`, `+`, and `++j`). For more information about prefix incrementing, see "Increment and Decrement Operators" in Chapter 4 on page 94.

  This interpretation is equivalent to the expression `a = i + (++j)`.

- Add the values of `i` and `j`, assign the sum to `a`, then postincrement `i` (where the tokens are `i`, `++`, `+`, and `j`). For more information about postincrementing, see "Postfix Increment and Decrement Operators" in Chapter 4 on page 90.

  This interpretation is equivalent to the expression `a = (i++) + j`.

Because the parser creates the longest token possible from the input stream, it chooses the second interpretation, making the tokens `i++`, `+`, and `j`.

1.2 Comments

A comment is text that the compiler ignores but is useful for programmers. Comments are normally used to annotate code for future reference. The compiler treats them as white space. Occasionally, comments are used to render certain lines of code inactive for test purposes; however, the **#if/#endif** preprocessor directives work better for this.

A C++ comment is written in one of the following ways:

- The `/*` (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the `*/` characters. This syntax is the same as ANSI C.
- The `//` (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. Therefore, it is commonly called a "single-line comment."

The comment characters `(/*`, `*/`, and `//)` have no special meaning within a character constant, string literal, or comment. Comments using the first syntax, therefore, cannot be nested. Consider this example:

```
/* Intent:  Comment out this block of code.
   Problem: Nested comments on each line of code are illegal.
FileName = String( "hello.dat" ); /* Initialize file string */
cout << "File: " << FileName << "\n"; /* Print status message */
*/
```

The preceding code will not compile because the compiler scans the input stream from the first `/*` to the first `*/` and considers it a comment. In this case, the first `*/` occurs at the end of the `Initialize file string` comment. The last `*/`, then, is no longer paired with an opening /*.

Note that the single-line form (`//`) of a comment followed by the line-continuation token (\) can have surprising effects. Consider this code:

```
#include <stdio.h>

int main()
{
    printf( "This is a number %d", // \
             5 );

    return 0;
}
```

After preprocessing, the preceding code appears as follows:

```
#include <stdio.h>

int main()
{
    printf( "This is a number %d", // 5 );

    return 0;
}
```

Because the single-line comment causes all further text on the same logical line to be considered a comment, the preceding program generates error messages.

1.3 Identifiers

An identifier is a sequence of characters used to denote one of the following:

- Object or variable name
- Class, structure, or union tag
- Enumerated type
- Member of a class, structure, union, or enumeration
- Function or class-member function
- **typedef** name
- Label name
- Macro name
- Macro parameter

Syntax

identifier:
nondigit
identifier nondigit
identifier digit

nondigit: one of
_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

digit: one of
0 1 2 3 4 5 6 7 8 9

Microsoft C++ identifiers are sequences of characters that must form a name shorter than 247 characters (actually, only the first 247 characters are significant). This restriction is complicated by the fact that names for user-defined types are "decorated" by the compiler to preserve type information; the resultant name, including the type information, cannot be longer than 247 characters. Factors that can influence the length of a decorated identifier are:

- Whether the identifier denotes an object of user-defined type
- Whether the identifier denotes a function
- The number of arguments to a function
- Whether the identifier names an object of user-defined type

The first character of an identifier must be an alphabetic character, either uppercase or lowercase, or an underscore (_). Because C++ identifiers are case sensitive, `fileName` is different from `FileName`.

Constraints

Identifiers cannot be exactly the same spelling and case as keywords (see "C++ Keywords" on page 6 for more information). (Identifiers that contain keywords are legal. For example, `Pint` is a legal identifier, even though it contains **int**, which is a keyword.)

Use of two sequential underscore characters (__) at the beginning of an identifier, or a single leading underscore followed by a capital letter, is reserved for C++ implementations in all scopes. Use of one leading underscore followed by a lowercase letter should be avoided for names with file scope because of possible conflicts with current or future reserved identifiers.

1.4 C++ Keywords

Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program.

The second section of keywords below are particular to Microsoft C++; these keywords are disabled when the /Za (ANSI-conformance) option is used during compilation.

Syntax

keyword: one of

asm[7]	**float**	**signed**
auto	**for**	**sizeof**
break	**friend**	**static**
case	**goto**	**struct**
catch[8]	**if**	**switch**
char	**inline**	**template**[8]
class	**int**	**this**
const	**long**	**throw**[8]
continue	**new**	**try**[8]
default	**operator**	**typedef**
delete	**private**	**union**
do	**protected**	**unsigned**
double	**public**	**virtual**
else	**register**	**void**
enum	**return**	**volatile**
extern	**short**	**while**

__asm[1,2]	**__finally**[1,6]	**__segment**[1,4]
__based[1,3]	**__fortran**[1,4]	**__segname**[1]
__cdecl[1]	**__huge**[1,4]	**__self**[1]
__emit[1,5]	**__interrupt**[1]	**__stdcall**[1]
__except[1,6]	**__loadds**[1,4]	**__syscall**[1]
__export[1]	**__near**[1]	**__try**[1,6]
__far[1]	**__pascal**[1,4]	
__fastcall[1,4]	**__saveregs**[1,4]	

[1]Microsoft-specific keyword.

[2]Replaces C++ **asm** syntax.

[3]The **__based** keyword has limited uses for 32-bit target compilations. In such compilations, **__based** is supported syntactically and for disambiguation purposes, but only the declaration of objects based on a pointer is supported semantically. Types that are based on a pointer are considered 32-bit displacements to a 32-bit base.

[4]Supported syntactically and for disambiguation purposes, but not meaningful in 32-bit compilations.

[5]**__emit** is not, strictly speaking, a keyword; rather it is a pseudoinstruction for the inline assembler. **__emit** is not supported for 32-bit compilations.

[6]Microsoft structured exception-handling keywords, **__try, __except**, and **__finally** are meaningful only for 32-bit targets.

[7]Reserved for compatibility with other C++ implementations, but not implemented. Use **__asm**.

[8]Not implemented in Microsoft C/C++ version 7.0.

Note The Microsoft extended keywords listed above are prefaced with a double underscore for ANSI compliance. For backward compatibility, however, the single-underscore versions of these keywords are supported unless the /Za (ANSI compliance) compilation option is specified.

Note that the keywords **near**, **far**, **huge**, **cdecl**, **fortran**, **pascal**, and **interrupt** are available with no leading underscores unless the /Za (ANSI compliance) compilation option is specified.

In addition to the C++ keywords shown above, Microsoft C++ defines the names in Table 1.1 as macros. Some of these macros can be tested using the **#ifdef** or **#ifdefined** preprocessor directive.

Table 1.1 Microsoft C/C++ Predefined Identifiers

Identifier	Compatibility	Value
__cplusplus	C++	The value of this macro is not significant. If it is defined, the program is compiled as C++. This macro is not defined for translation units compiled as C.
__DATE__	ANSI C, C++	The date of compilation of the source file. The date is a character string of the form `"Mmm dd yyyy"`. The quotes are included to form a proper C++ string.
__FILE__	ANSI C, C++	The name of the current source file. __FILE__ expands to a string surrounded by double quotes.
__LINE__	ANSI C, C++	The line number in the current source file. The line number is a decimal number.
__STDC__	ANSI C	Defined equal to 1 only if /Za (ANSI-conformance) compilation option used; otherwise undefined.
__TIME__	ANSI C, C++	The time of compilation of the source file. The time is a character string of the form `"hh:mm:ss"`. The quotes are included to form a proper C++ string.
__TIMESTAMP__	Microsoft	The date and time of translation of the current source file. The timestamp is a character string of the form `"Ddd Mmm dd hh:mm:ss"`. The quotes are included to form a proper C++ string.
_MSC_VER	Microsoft	Defines the compiler version as a string literal in the form `ddd`. For Microsoft C/C++ version 7.0, the string is `"700"`.
_MSDOS	Microsoft	Always defined. Identifies target operating system as MS-DOS.
_M_I86	Microsoft	Always defined. Identifies target machine as a member of the 8086 family.
_M_I8086	Microsoft	Defined when compilation targets 8086 and 8088 processors (default or /G0 compiler option).

Table 1.1 *(continued)*

Identifier	Compatibility	Value
_M_I286	Microsoft	Defined when compilation targets 80286 processor (/G2 compiler option).
_M_I386	Microsoft	Defined when compilation targets 80386 processor (/G3 flat-model compilation—not available on compilers that target 16-bit applications).
_M_I86*m*M	Microsoft	Always defined. Identifies memory model, where *m* is either **S** (small or tiny model), **M** (Medium model), **C** (compact model), **L** (large model), or **H** (huge model). If huge model is used, both **_M_I86LM** and **_M_I86HM** are defined. Small model is the default. For more information about memory models, see Appendix B, "Microsoft-Specific Modifiers."
__DLL	Microsoft	Defined for run-time library as a DLL (/MD compiler option).
NO_EXT_KEYS	Microsoft	No longer emitted by the compiler. This macro was defined in previous versions of Microsoft C for compilations that used the /Za (ANSI-conformance) option. In Microsoft C/C++ version 7.0, the **__STDC__** macro is used instead.
_CHAR_UNSIGNED	Microsoft	Defined only when the /J compiler option is given to make **char** unsigned by default.

1.5 Punctuators

Punctuators in C++ have syntactic and semantic meaning to the compiler, but do not, of themselves, specify an operation that yields a value. Some punctuators, either alone or in combination, can also be C++ operators or be significant to the preprocessor.

Syntax

punctuator: one of

```
! % ^ & * ( ) - + = { } | ~
[ ] \ ; ' : " < > ? , . / #
```

The punctuators [ ], ( ), and { } must appear in pairs after translation phase 4. (For more information, see Appendix A, "Phases of Translation.")

1.6 Operators

Operators specify an evaluation to be performed on one of the following:

- One operand (unary operator)
- Two operands (binary operator)
- Three operands (ternary operator)

The C++ language includes all C operators and adds several new operators. The following syntax lists those operators unique to C++ first, and it then lists the operators shared between C and C++.

Syntax

operator: one of

C++ Operators

.*	: :	**delete**
–>*	**new**	

C/C++ Operators

[	%	*=
]	<<	/=
(	>>	%=
)	<	+=
.	>	–=
–>	<=	<<=
++	>=	>>=
––	==	&=
&	!=	^=
*	^	|=
+	|	'
–	&&	#
~	||	##
!	?	:> (Microsoft C and C++ specific)
sizeof	:	
/	=	

Operators follow a strict precedence. This precedence defines the evaluation order of expressions containing these operators. Operators associate with either the expression on their left or the expression on their right; this is called "associativity." Table 1.2 shows the precedence and associativity of C++ operators (from highest to lowest precedence).

Table 1.2 C++ Operator Precedence, Syntax, and Associativity

Operator	Name or Meaning	Syntax	Associativity
==	Equality	*equality-expression* == *relational-expression*	Left to right
::	Scope resolution	*class-name* :: *name*	None
::	Global	:: *name*	None
[ ]	Array subscript	*postfix-expression* [*expression*$_{opt}$]	Left to right
( )	Function call	*postfix-expression* (*expression-list*$_{opt}$)	Left to right
( )	Conversion	*simple-type-name* (expression-list$_{opt}$)	None
.	Member selection (object)	*postfix-expression* . *name*	Left to right
->	Member selection (pointer)	*postfix-expression* -> *name*	Left to right
++	Postfix increment	*postfix-expression* ++	None
--	Postfix decrement	*postfix-expression* --	None
new	Allocate object	::$_{opt}$ **new** *placement*$_{opt}$ *new-type-name new-initializer*$_{opt}$	None
		::$_{opt}$ **new** *placement*$_{opt}$ (*type-name*) *new-initializer*$_{opt}$	None
delete	Deallocate object	::$_{opt}$ **delete** *cast-expression*	None
delete[ ]		::$_{opt}$ **delete** [ ] *cast-expression*	None
++	Prefix increment	++ *unary-expression*	None
--	Prefix decrement	-- *unary-expression*	None
*	Dereference	* *cast-expression*	None
&	Address-of	& *cast-expression*	None
+	Unary plus	+ *cast-expression*	None
-	Arithmetic negation (unary)	- *cast-expression*	None

Table 1.2 *(continued)*

Operator	Name or Meaning	Syntax	Associativity
!	Logical NOT	! *cast-expression*	None
~	Bitwise complement	~ *cast-expression*	None
:>	Base operator	*base-expression* :> *expression*	None
sizeof	Size of object	**sizeof** *unary-expression*	None
sizeof ( )	Size of type	**sizeof**(*type-name*)	None
(*type*)	Type cast (conversion)	(*type-name*) *cast-expression*	Right to left
.*	Apply pointer to class member (objects)	*pm-expression* .* *cast-expression*	Left to right
–>*	Dereference pointer to class member	*pm-expression* –>* *cast-expression*	Left to right
*	Multiplication	*multiplicative-expression* * *pm-expression*	Left to right
/	Division	*multiplicative-expression* / *pm-expression*	Left to right
%	Remainder (modulus)	*multiplicative-expression* % *pm-expression*	Left to right
+	Addition	*additive-expression* + *multiplicative-expression*	Left to right
–	Subtraction	*additive-expression* – *multiplicative-expression*	Left to right
<<	Left shift	*shift-expression* << *additive-expression*	Left to right
>>	Right shift	*shift-expression* >> *additive-expression*	Left to right
<	Less than	*relational-expression* < *shift-expression*	Left to right
>	Greater than	*relational-expression* > *shift-expression*	Left to right

Table 1.2 (*continued*)

Operator	Name or Meaning	Syntax	Associativity
<=	Less than or equal to	*relational-expression* <= *shift-expression*	Left to right
>=	Greater than or equal to	*relational-expression* >= *shift-expression*	Left to right
!=	Inequality	*equality-expression* != *relational-expression*	Left to right
&	Bitwise AND	*and-expression* & *equality-expression*	Left to right
^	Bitwise exclusive OR	*exclusive-or-expression* ^ *and-expression*	Left to right
|	Bitwise OR	*inclusive-or-expression* | *exclusive-or-expression*	Left to right
&&	Logical AND	*logical-and-expression* && *inclusive-or-expression*	Left to right
||	Logical OR	*logical-or-expression* || *logical-and-expression*	Left to right
e1?*e2*:*e3*	Conditional	*logical-or-expression* ? *expression* : *conditional-expression*	Right to left
=	Assignment	*unary-expression* = *assignment-expression*	Right to left
*=	Multiplication assignment	*unary-expression* *= *assignment-expression*	Right to left
/=	Division assignment	*unary-expression* /= *assignment-expression*	Right to left
%=	Modulus assignment	*unary-expression* %= *assignment-expression*	Right to left
+=	Addition assignment	*unary-expression* += *assignment-expression*	Right to left
-=	Subtraction assignment	*unary-expression* -= *assignment-expression*	Right to left
<<=	Left-shift assignment	*unary-expression* <<= *assignment-expression*	Right to left

Table 1.2 *(continued)*

Operator	Name or Meaning	Syntax	Associativity
>>=	Right-shift assignment	*unary-expression* >>= *assignment-expression*	Right to left
&=	Bitwise AND assignment	*unary-expression* &= *assignment-expression*	Right to left
|=	Bitwise inclusive OR assignment	*unary-expression* |= *assignment-expression*	Right to left
^=	Bitwise exclusive OR assignment	*unary-expression* ^= *assignment-expression*	Right to left
,	Comma	*expression*, *assignment-expression*	Left to right

The [ ], ( ), and **? :** operators (array subscript, function call, and conditional, respectively) can be used only as pairs. However, these operators can be separated by expressions (see Chapter 4, "Expressions," for more information).

The # and ## operators can occur only in **#define** preprocessor directives.

1.7 Literals

Invariant program elements are called "literals" or "constants." The terms "literal" and "constant" are used interchangeably here. Literals fall into four major categories: integer, character, floating-point, and string literals.

Syntax

literal:
integer-constant
character-constant
floating-constant
string-literal

Integer Constants

Integer constants are constant data elements that have no fractional parts or exponents. They always begin with a digit. Integer constants can be specified in decimal, octal, or hexadecimal form. They can specify signed or unsigned types and long or short types.

Syntax

integer-constant:
decimal-constant integer-suffix$_{opt}$
octal-constant integer-suffix$_{opt}$
hexadecimal-constant integer-suffix$_{opt}$
'*c-char-sequence*'

decimal-constant:
nonzero-digit
decimal-constant digit

octal-constant:
0
octal-constant octal-digit

hexadecimal-constant:
0x *hexadecimal-digit*
0X *hexadecimal-digit*
hexadecimal-constant hexadecimal-digit

nonzero-digit: one of
1 2 3 4 5 6 7 8 9

octal-digit: one of
0 1 2 3 4 5 6 7

hexadecimal-digit: one of
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

integer-suffix:
unsigned-suffix long-suffix$_{opt}$
long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
u U

long-suffix: one of
l L

To specify integer constants using octal or hexadecimal notation, use a prefix that denotes the base. To specify an integer constant of a given integral type, use a suffix that denotes the type.

To specify a decimal constant, begin the specification with a nonzero digit. For example:

```
int i = 157;   // Decimal constant
int j = 0198;  // Not a decimal number; erroneous octal constant
int k = 0365;  // Leading zero specifies octal constant, not decimal
```

To specify an octal constant, begin the specification with 0 , followed by a sequence of digits in the range 0 through 7. The digits 8 and 9 are errors in the specification of an octal constant. For example:

```
int i = 0377;   // Octal constant
int j = 0397;   // Error: 9 is not an octal digit
```

To specify a hexadecimal constant, begin the specification with `0x` or `0X` (the case of the "x" does not matter), followed by a sequence of digits in the range `0` through `9` and `a` (or `A`) through `f` (or `F`). Hexadecimal digits `a` or A through `f` or F) represent values in the range 10 through 15. For example:

```
int i = 0x3fff; // Hexadecimal constant
int j = 0X3FFF; // Equal to i
```

To specify an unsigned type, use either the **u** or **U** suffix. To specify a long type, use either the **l** or **L** suffix. For example:

```
unsigned uVal = 328u;              // Unsigned value
long lVal = 0x7FFFFFL;             // Long value specified
                                   //  as hex constant
unsigned long ulVal = 0776745ul;   // Unsigned long value
```

Character Constants

Character constants are one or more members of the "source character set," the character set in which a program is written, surrounded by single quotation marks (`'`). They are used to represent characters in the "execution character set," the character set on the machine where the program executes.

Microsoft Specific For Microsoft C++, the source and execution character sets are both ASCII. ♦

There are three kinds of character constants:

- Normal character constants
- Multicharacter constants
- Wide character constants

Note Use wide character constants in place of multicharacter constants to ensure portability and upward compatibility.

Character constants are specified as one or more characters enclosed in single quotation marks. For example:

```
char ch = 'x';          // Specify normal character constant.
int mbch = 'ab';        // Specify system-dependent
                        //  multicharacter constant.
wchar_t wcch = L'ab';   // Specify wide character constant.
```

Note that `mbch` is of type **int**. If it were declared as type **char**, the second byte would not be retained. The number of meaningful characters in a multicharacter constant is equal to the expression **sizeof(int)**. For 16-bit targets (/G0, /G1, and /G2 compilation options), this is 2; for 32-bit targets (flat-model compilation), this is 4. Specifying too many characters for a multicharacter constant generates an error message.

Syntax

character-constant:
  '*c-char-sequence*'
  L'*c-char-sequence*'

c-char-sequence:
  c-char
  c-char-sequence c-char

c-char:
  any member of the source character set except the single quote ('), backslash (\), or newline character
  escape-sequence

escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence

simple-escape-sequence: one of
  **\' \" \? **
  \a \b \f \n \r \t \v

octal-escape-sequence:
  \octal-digit
  \octal-digit octal-digit
  \octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:
  \x hexadecimal-digit
  hexadecimal-escape-sequence hexadecimal-digit

Microsoft C++ supports normal multicharacter, and wide character constants. Use wide character constants to specify members of the extended execution character

set (for example, to support an international application). Normal character constants have type **char**, multicharacter constants have type **int**, and wide character constants have type **wchar_t**. (The type **wchar_t** is defined in the standard include files STDDEF.H, STDLIB.H, and STRING.H. The wide-character functions, however, are prototyped only in STDLIB.H.)

The only difference in specification between normal and wide character constants is that wide character constants are preceded by the letter `L`. For example:

```
char schar = 'x';               // Normal character constant
wchar_t wchar = L'\x81\x19';    // Wide character constant
```

Table 1.3 shows reserved or nongraphic characters that are system dependent or not allowed within character constants. These characters should be represented with escape sequences.

Table 1.3 C++ Reserved or Nongraphic Characters

Character	ASCII Representation	ASCII Value	Escape Sequence
Newline	NL (LF)	10 or 0x0a	\n
Horizontal tab	HT	9	\t
Vertical tab	VT	11 or 0x0b	\v
Backspace	BS	8	\b
Carriage return	CR	13 or 0x0d	\r
Formfeed	FF	12 or 0x0c	\f
Alert	BEL	7	\a
Backslash	\	92 or 0x5c	\\
Question mark	?	63 or 0x3f	\?
Single quotation mark	'	39 or 0x27	\'
Double quotation mark	"	34 or 0x22	\"
Octal number	*ooo*	---	*ooo*
Hexadecimal Number	*hhh*	---	\x*hhh*
Null character	NUL	0	\0

Important If the character following the backslash does not specify a legal escape sequence, the result is implementation defined. In Microsoft C++, the character following the backslash is taken literally, as though the escape were not present, and a level 1 warning ("unrecognized character escape sequence") is issued.

Octal escape sequences, specified in the form *\ooo*, consist of a backslash and one, two, or three octal characters. Hexadecimal escape sequences, specified in the form \x*hhh*, consist of the characters `\x` followed by a sequence of hexadecimal digits. Unlike octal escape constants, there is no limit on the number of hexadecimal digits in an escape sequence.

Octal escape sequences are terminated by the first character that is not an octal digit, or when three characters are seen. For example:

```
wchar_t och = L'\076a';  // Sequence terminates at a
char    ch = '\233';     // Sequence terminates after 3 characters
```

Similarly, hexadecimal escape sequences terminate at the first character that is not a hexadecimal digit. Because hexadecimal digits include the letters `a` through `f` (and `A` through `F`), make sure the escape sequence terminates at the intended digit.

Because the single quotation mark (') encloses character constants, use the escape sequence `\'` to represent enclosed single quotation marks. The double quotation mark (") can be represented without an escape sequence. The backslash character (\) is a line-continuation character when placed at the end of a line. If you want a backslash character to appear within a character constant, you must type two backslashes in a row (`\\`). (See Appendix A, "Phases of Translation," for more information about line continuation.)

Floating-Point Constants

Floating-point constants specify values that must have a fractional part. These values contain decimal points (.) and may contain exponents.

Syntax

floating-constant:
  fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
  digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
  digit-sequence$_{opt}$. *digit-sequence*
  digit-sequence .

exponent-part:
  e *sign*$_{opt}$ *digit-sequence*
  E *sign*$_{opt}$ *digit-sequence*

sign: one of
+ –

digit-sequence:
digit
digit-sequence digit

floating-suffix: one of
f l F L

Floating-point constants have a "mantissa," which specifies the value of the number, an "exponent," which specifies the magnitude of the number, and an optional suffix that specifies the constant's type. The mantissa is specified as a sequence of digits followed by a period, followed by an optional sequence of digits representing the fractional part of the number. For example:

```
18.46
38.
```

The exponent, if present, specifies the magnitude of the number as a power of 10, as shown in the following example:

```
18.46e0     // 18.46
18.46e1     // 184.6
```

If an exponent is present, the trailing decimal point is unnecessary in whole numbers such as `18E0`.

Floating-point constants default to type **double**. By using the suffixes **f** or **l** (or **F** or **L**—the suffix is not case sensitive), the constant can be specified as **float** or **long double**, respectively.

String Literals

A string literal consists of zero or more characters from the source character set surrounded by double quotation marks ("). A string literal represents a sequence of characters, which, taken together, forms a null-terminated string. While some C++ class libraries, including the Microsoft libraries, supply sophisticated string-handling functionality, the strings defined in the language are relatively simple.

Syntax

string-literal:
"*s-char-sequence*$_{opt}$"
L"*s-char-sequence*$_{opt}$"

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except double quotation marks ("), backslash (\), or newline
escape-sequence

C++ strings have these types:

- Array of **char**[*n*], where *n* is the length of the string (in characters) plus 1 for the terminating '\0' that marks the end of the string
- Array of **wchar_t**, for wide-character strings

The result of modifying a string constant is undefined. For example:

```
char *szStr = "1234";
szStr[2] = 'A';     // Results undefined
```

Microsoft Specific

In some cases, identical string literals may be "folded" to save space in the executable file. In string-literal folding, the compiler causes all references to a particular string literal to point to the same location in memory, instead of having each reference point to a separate instance of the string literal:

```
#include <iostream.h>
#include <string.h>
//  Define two pointers that refer to identical
//  string literals.
char *sz1 = "A String";
char *sz2 = "A  String";

void main()
{
    // Reverse sz1
    for( int i = 0, j = strlen( sz1 ) - 1; i < j; ++i, --j )
    {
       char chTmp = sz1[i];
       sz1[i] = sz1[j];
       sz1[j] = chTmp;
    }

    // Display the result of the program.
    cout << "sz1 = " << sz1 << endl;
    cout << "sz2 = " << sz2 << endl;
}
```

If the literals are not folded, the output of the program is:

```
sz1 = gnirtS A
sz2 = A String
```

However, if the strings are folded, the output of the program is:

```
sz1 = gnirtS A
sz2 = gnirtS A♦
```

When specifying string literals, adjacent strings are concatenated. Therefore, this declaration:

```
char szStr[] = "12" "34";
```

is identical to this declaration:

```
char szStr[] = "1234";
```

This concatenation of adjacent strings makes it easy to specify long strings across multiple lines:

```
cout << "Four score and seven years "
        "ago, our forefathers brought forth "
        "upon this continent a new nation.";
```

In the preceding example, the entire string "Four score and seven years ago, our forefathers brought forth upon this continent a new nation." is spliced together. This string might also have been specified using line splicing as follows:

```
cout << "Four score and seven years \
ago, our forefathers brought forth \
upon this continent a new nation.";
```

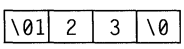
After all adjacent strings in the constant have been concatenated, the **NULL** character, `'\0'`, is appended to provide an end-of-string marker for C string-handling functions.

When the first character of the first string is an escape character, string concatenation can yield surprising results. Consider the following two declarations:

```
char szStr1[] = "\01" "23";
char szStr2[] = "\0123";
```

While it is natural to assume that `szStr1` and `szStr2` contain the same values, the values they actually contain are shown in Figure 1.1.

"\01" "23"



"\0123"

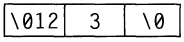


Figure 1.1 Escapes and String Concatenation

Microsoft Specific

The maximum length of a string literal is 2,048 bytes. This limit applies both to strings of type **char[]** and **wchar_t[]**.♦

Determine the size of string objects by counting the number of characters and adding 1 for the terminating `'\0'` .

Because the double quotation mark `(")` encloses strings, use the escape sequence (`\"`) to represent enclosed double quotation marks. The single quotation mark (`'`) can be represented without an escape sequence. The backslash character (`\`) is a line-continuation character when placed at the end of a line. If you want a backslash character to appear within a string, you must type two backslashes (`\\`). (For more information about line continuation, see Appendix A, "Phases of Translation.")

To specify a string of type wide character (**wchar_t[]**), precede the opening double quotation mark with the character **L**. For example:

```
wchar_t wszStr[] = L"1a1g";
```

All normal escape codes listed in the "Character Constants" example on page 16 are valid in string constants. For example:

```
cout << "First line\nSecond line";
cout << "Error! Take corrective action\a";
```

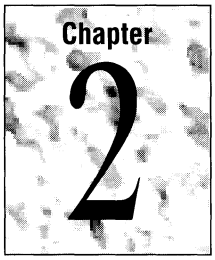
Because the escape code terminates at the first character that is not a hexadecimal digit, specification of string constants with embedded hexadecimal escape codes can cause unexpected results. The following example is intended to create a string literal containing ASCII 5, followed by the characters `five`:

```
"\x05five"
```

The actual result is a hexadecimal 5F, which is the ASCII code for an underscore, followed by the characters `ive`. The following example produces the desired results:

```
"\005five"     // Use octal constant.
"\x05" "five"  // Use string splicing.
```

Basic Concepts



This chapter explains some concepts that are key to the understanding of C++. While many of these concepts are familiar to C programmers, there are some subtle differences that can cause unexpected program results. Some of the topics covered in this chapter are:

- Terminology. Terms used later in this book are introduced and defined.
- Scope. The scope of a C++ object or function is different from C; the scoping rules are defined. See "Scope" on page 28.
- Linkage. Linkage rules are described as is the definition of a "program." See "Program and Linkage" on page 33.
- Program startup, termination, and the **main** function. The sequence of startup and termination is discussed, as is defining the purpose and behavior of the **main** function. See "Startup and Termination" on page 38.
- Storage classes. The treatment of **auto** and **static** objects in C++, including initialization and destruction, is discussed. See "Storage Classes" on page 46.
- Types. The behavior of C++ fundamental (built-in) types is explained. The discussion includes derived types, type names, name spaces, and limits for each type. See "Types" on page 49.

2.1 Terms

The following short definitions explain C++ terminology used in this chapter and in the rest of the book, as well.

Table 2.1 C++ Terminology

Term	Meaning
Declaration	A declaration introduces names and their types into a program without necessarily defining an associated object or function. However, many declarations do serve as definitions.
Definition	A definition provides information that allows the compiler to allocate memory for objects or generate code for functions.
Dereference	Dereferencing converts a pointer value to an r-value.
Lifetime	The lifetime of an object is the period during which an object exists, including its creation and destruction.
Linkage	Names can have external linkage, internal linkage, or no linkage. Within a program (a set of translation units), only names with external linkage denote the same object or function. Within a translation unit, names with either internal or external linkage denote the same object or function (except when functions are overloaded). (For more information on translation units, see Appendix A, "Phases of Translation.") Names with no linkage denote unique objects or functions.
Name	A name denotes an object, function, set of overloaded functions, enumerator, type, class member, template, value, or label. C++ programs use names to refer to their associated language element. Names can be type names or identifiers.
Object	An object is an instance (a data item) of a user-defined type (a class type). The difference between an object and a variable is that variables retain state information, whereas objects may also have behavior. This manual draws a distinction between objects and variables: "object" means instance of a user-defined type, whereas "variable" means instance of a fundamental type. In cases where either object or variable is applicable, the term "object" is used as the inclusive term, meaning "object or variable."
Scope	Names can be used only within specific regions of program text. These regions are called the scope of the name.
Storage class	The storage class of a named object determines its lifetime, initialization, and, in certain cases, its linkage.
Type	Names have associated types that determine the meaning of the value or values stored in an object or returned by a function.
Variable	A variable is a data item of a fundamental type (for example, **int**, **float**, or **double**). Variables store state information but define no behavior for how that information is handled. See the list item "Object" above for information about how the terms "variable" and "object" are used in this manual.

2.2 Declarations and Definitions

Declarations explain to the compiler that a program element or name exists; definitions specify to the compiler what code or data the name describes. A name must be declared before it can be used.

Declarations

A declaration introduces one or more names into a program. Declarations also serve as definitions, except in the following cases:

- The declaration is a function prototype (a function declaration with no function body).
- The declaration contains the **extern** specifier but no initializer (objects and variables) or function body (functions). This signifies that the definition is not necessarily in the current translation unit and gives the name external linkage.
- The declaration is of a static data member inside a class declaration.

Because static class data members are discrete variables shared by all objects of the class, they must be defined and initialized outside the class declaration. (For more information about classes and class members, see Chapter 8, "Classes.")

- The declaration is a class name declaration with no following definition.
- The declaration is a **typedef** statement.

Examples of declarations that are also definitions are:

```
// Declare and define int variables i and j.
int i;
int j = 10;

// Declare enumeration suits.
enum suits { Spades = 1, Clubs, Hearts, Diamonds };

// Declare class CheckBox.
class CheckBox : public Control
{
public:
            Boolean IsChecked();
    virtual int     ChangeState() = 0;
};
```

Some declarations that are not definitions are:

```
extern int i;
char *strchr( const char *Str, const char Target );
```

Definitions

A definition is a unique specification of an object or variable, function, class, or enumerator. Because definitions must be unique, a program can contain only one definition for a given program element. Note that because declarations can occur more than once in a program, classes, structures, enumerated types, and so on can be declared for each compilation unit. The constraint on this multiple declaration is that all declarations must be identical.

There can be a many-to-one correspondence between declarations and definitions. There are two cases in which a program element can be declared and not defined:

- A function is declared but never referenced with a function call or with an expression that takes the function's address.
- A class is used only in a way that does not require its definition be known. However, the class must be declared. The following code illustrates such a case:

```
class WindowCounter;     // Forward reference; no definition

class Window
{
    static WindowCounter windowCounter;  // Definition of
                                         //  WindowCounter
                                         //  not required.
};
```

2.3 Scope

C++ names can be used only in certain regions of a program. This area is called the "scope" of the name. Scope determines the "lifetime" of a name that does not denote a static object. Scope also determines the visibility of a name, when class constructors and destructors are called, and when variables local to the scope are initialized. (For more information, see "Constructors" and "Destructors" in Chapter 11, on pages 300 and 305, respectively.) There are five kinds of scope:

- Local scope. A name declared within a block is accessible only within that block and blocks enclosed by it, and only after the point of declaration. The names of formal arguments to a function in the scope of the outermost block of the function have local scope, as if they had been declared inside the block enclosing the function body. Consider the following code fragment:

  ```
  {
      int i;
  }
  ```

  Because the declaration of `i` is in a block enclosed by curly braces, `i` has local scope.

- Function scope. Labels are the only names that have function scope. They can be used anywhere within a function but are not accessible outside that function.
- File scope. Any name declared outside all blocks or classes has file scope. It is accessible anywhere in the translation unit after its declaration. Names with file scope that do not declare static objects are often called "global" names.
- Class scope. Names of class members have class scope. Class member functions can be accessed only by using the member-selection operators (. or –>) or pointer-to-member operators (.* or –>*) on an object or pointer to an object of that class; nonstatic class member data is considered local to the object of that class. Consider the following class declaration:

```
class Point
{
    int x;
    int y;
};
```

  The class members `x` and `y` are considered to be in the scope of class `Point`.
- Prototype scope. Names declared in a function prototype are visible only until the end of the prototype. The following prototype declares two names (`szDest` and `szSource`); these names go out of scope at the end of the prototype:

```
char *strcpy( char *szDest, const char *szSource );
```

Point of Declaration

A name is considered declared immediately after its declarator but prior to its (optional) initializer. (For more information on declarators, see Chapter 7.) An enumerator is considered declared immediately after the identifier that names it but prior to its (optional) initializer.

Consider this example:

```
double dVar = 7.0;

int main()
{
    double dVar = dVar;
}
```

If the point of declaration were *after* the initialization, then `dVar` would be initialized to 7.0, the value of the global variable `dVar`. However, since that is not the case, `dVar` is initialized to an undefined value.

Enumerators follow the same rule. However, enumerators are exported to the enclosing scope of the enumeration. In the following example, the enumerators `Spades`, `Clubs`, `Hearts`, and `Diamonds` are declared. Because the enumerators are

exported to the enclosing scope, they are considered to have global scope. The identifiers in the following example are already defined in global scope.

Consider the following code:

```
const int Spades = 1, Clubs = 2, Hearts = 3, Diamonds = 4;

enum Suits
{
    Spades = Spades,
    Clubs,
    Hearts,
    Diamonds
};
```

Because the identifiers in the preceding code are already defined in global scope, an error message is generated.

Note Using the same name to refer to more than one program element—for example, an enumerator and an object—is considered poor programming practice and should be avoided. In the preceding example, this practice causes an error.

Hiding Names

A name can be hidden by declaring that name in an enclosed block. In Figure 2.1, `i` is redeclared within the inner block, thereby hiding the variable associated with `i` in the outer block scope.

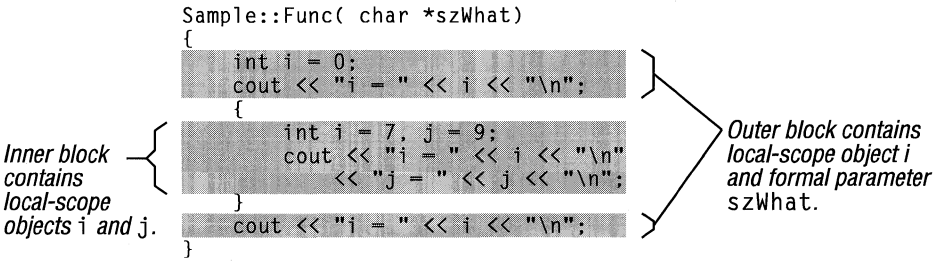


Figure 2.1 Block Scope and Name Hiding

The output from the program shown in Figure 2.1 is:

```
i = 0
i = 7
j = 9
i = 0
```

Note The argument `szWhat` is considered to be in the scope of the function. Therefore, it is treated as if it had been declared in the outermost block of the function.

Hiding Names with File Scope

Names with file scope can be hidden by explicitly declaring the same name in block scope. However, these names can be accessed using the scope-resolution operator (**::**). For example:

```
#include <iostream.h>

int i = 7;      // i has file scope--declared
                // outside all blocks

int main( int argc, char *argv[] )
{
    int i = 5;  // i has block scope--hides
                // the i with file scope

    cout << "Block-scoped i has the value: " << i << "\n";
    cout << "File-scoped i has the value: " << ::i << "\n";

    return 0;
}
```

The result of the preceding code is:

```
Block-scoped i has the value: 5
File-scoped i has the value: 7
```

Hiding Class Names

Class names can be hidden by declaring a function, object or variable, or enumerator in the same scope. However, the class name can still be accessed when prefixed by the keyword **class**.

```
// Declare class Account at file scope.
class Account
{
public:
    Account( double InitialBalance )
        { balance = InitialBalance; }
    double GetBalance()
        { return balance; }
    double Deposit( double HowMuch )
        { return balance += HowMuch; }
    double Withdraw( double HowMuch )
        { return balance -= HowMuch; }
private:
    double balance;
};

double Account = 15.37;            // Hides class name Account

int main( int argc, char *argv[] )
{
    class Account Checking( Account );

    cout << "Opening account with balance of: "
         << Checking.GetBalance() << "\n";
    cout << "Depositing $10.57, for a remaining balance of: "
         << Checking.Deposit( 10.57 ) << "\n";
    cout << "Withdrawing $27.16, for a remaining balance of: "
         << Checking.Withdraw( 27.16 ) << "\n";

    return 0;
}
```

Note that any place the class name (`Account`) is called for, the keyword **class** must be used to differentiate it from the function-scoped variable `Account`. This rule does not apply when the class name occurs on the left side of the scope-resolution operator (**::**). Names on the left side of the scope-resolution operator are always considered class names. The following example demonstrates how to declare a pointer to an object of type `Account` using the **class** keyword:

```
class Account *Checking = new class Account( Account );
```

Note that the `Account` in the initializer (in parentheses) in the preceding statement has function scope; it is of type **double**.

(For more information about pointers, see "Derived Types" on page 52. For information about declaration and initialization of class objects, see Chapter 8, "Classes." For information about using the **new** and **delete** free-store operators, see Chapter 11, "Special Member Functions.")

Scope of Formal Arguments to Functions

Formal arguments (arguments specified in function headers) to functions are considered to be in the scope of the outermost block of the function body.

2.4 Program and Linkage

A program consists of one or more translation units linked together. Execution (conceptually) begins in the translation unit that contains the function **main**. (For more information about translation units, see Appendix A, "Phases of Translation." For more information about the **main** function, see "Program Startup—the main Function" on page 38.)

Types of Linkage

The way the names of objects and functions are shared between translation units is called "linkage." These names can have:

- Internal linkage, in which case they refer only to program elements inside their own translation units; they are not shared with other translation units.

  The same name in another translation unit may refer to a different object or a different class. Names with internal linkage are sometimes referred to as being "local" to their translation units. An example declaration of a name with internal linkage is:

  ```
  static int i;   // The static keyword ensures internal linkage.
  ```

- External linkage, in which case they can refer to program elements in any translation unit in the program—the program element is shared among the translation units.

  The same name in another translation unit is guaranteed to refer to the same object or class. Names with external linkage are sometimes referred to as being "global."

  An example declaration of a name with external linkage is:

  ```
  extern int i;
  ```

- No linkage, in which case they refer to unique entities. The same name in another scope may not refer to the same object. (Note, however, that you can pass a pointer to an object with no linkage. This makes the object accessible in other translation units.)

Linkage in Names with File Scope

The following linkage rules apply to names (other than **typedef** and enumerator names) with file scope:

- If a name is explicitly declared as **static**, it has internal linkage and identifies a program element inside its own translation unit.
- If a function name with file scope is explicitly declared as **inline**, it has external linkage (Microsoft specific).
- If a name is declared as **const** but not as **extern**, the name still has external linkage (Microsoft specific).
- A class has internal linkage if it meets these criteria (Microsoft specific):
  - Uses no C++ functionality (for example, member-access control, member functions, constructors, destructors, and so on).
  - Not used in the declaration of another name that has external linkage. This constraint means that objects of class type that are passed to functions with external linkage cause the class to have external linkage.
- Enumerator names and **typedef** names have no linkage.
- All other names with file scope have external linkage.

Linkage in Names with Class Scope

The following linkage rules apply to names with file scope:

- Static class members have external linkage.
- Class member functions have external linkage.
- Functions declared as **friend** functions must have external linkage. Declaring a static function as a **friend** is an error (Microsoft specific).
- Enumerators and **typedef** names do not have external linkage.

Linkage in Names with Block Scope

The following linkage rules apply to names with block scope (local names):

- Names declared as **extern** have external linkage unless they were previously declared as **static**.
- All other names with block scope have no linkage.

Names with No Linkage

The only names that have no linkage are:

- Function parameters.
- Block-scoped names not declared as **extern** or **static**.
- Enumerators.
- Names declared in a **typedef** statement. An exception is when the **typedef** statement is used to provide a name for an unnamed class type. The name may then have external linkage if the class has external linkage. The following example shows a situation in which a **typedef** name has external linkage:

```
typedef struct
{
    short x;
    short y;
} POINT;
extern int MoveTo( POINT pt );
```

  The **typedef** name, `POINT` becomes the class name for the unnamed **structure**. It is then used in the declaration of a function with external linkage.

Because **typedef** names have no linkage, their definitions can differ between translation units. Because the compilations take place discretely, there is no way for the compiler to detect these differences. As a result, errors of this kind are not detected until link time. Consider the following case:

```
// Translation unit 1
typedef int INT

INT myInt;
...

// Translation unit 2
typedef short INT

extern INT myInt;
...
```

The preceding code generates an "unresolved external" error at link time.

C++ functions can be defined only in file or class scope. The following example illustrates how to define functions and shows an erroneous function definition:

```
#include <iostream.h>

void ShowChar( char ch );     // Declare function ShowChar.

void ShowChar( char ch )      // Define function ShowChar.
{                             // Function has file scope.
    cout << ch;
}

struct Char                   // Define class Char.
{
    char Show();              // Declare Show function.
    char Get();               // Declare Get function.
    char ch;
};

char Char::Show()             // Define Show function
{                             //  with class scope.
    cout << ch;
    return ch;
}

void GoodFuncDef( char ch )   // Define GoodFuncDef
{                             //  with file scope.
    int BadFuncDef( int i )   // Erroneous attempt to
    {                         //  nest functions.
        return i * 7;
    }
    for( int i = 0; i < BadFuncDef( 2 ); ++i )
        cout << ch;
    cout << "\n";
}
```

Linkage to Non-C++ Functions

C functions and data can be accessed only if they are previously declared as having C linkage. However, they must be defined in a separately compiled translation unit.

Syntax

linkage-specification:
extern *string-literal*{*declaration-list*$_{opt}$}
extern *string-literal declaration*

declaration-list:
declaration
declaration-list declaration

Microsoft C++ supports the strings "C" and "C++" in the *string-literal* field. The following example shows alternative ways to declare names that have C linkage:

```
// Declare printf with C linkage.
extern "C" int printf( const char *fmt, ... );

//  Cause everything in the header file "cinclude.h"
//   to have C linkage.
extern "C"
{
#include <cinclude.h>
}

//  Declare the two functions ShowChar and GetChar
//   with C linkage.
extern "C"
{
    char ShowChar( char ch );
    char GetChar( void )
}

//  Define the two functions ShowChar and GetChar
//   with C linkage.
extern "C" char ShowChar( char ch )
{
    putchar( ch );
    return ch;
}

extern "C" char GetChar( void )
{
    char ch;

    ch = getchar()
    return ch;
}

// Declare a global variable, errno, with C linkage.
extern "C" int errno;
```

2.5 Startup and Termination

Program startup and termination is facilitated by using two functions: **main** and **exit**. Additionally, other startup and termination code may be executed.

Program Startup—the main Function

A special function called **main** is the entry point to all C++ programs. This function is not predefined by the compiler; rather, it must be supplied in the program text. The declaration syntax for **main** is as follows:

int main();

or, optionally:

int main(int *argc*[[**, char** **argv*[ ] [[, **char** **envp*[ ]]]]] **);**

Alternatively, the **main** function can be declared as returning **void** (no return value). If you declare **main** as returning **void**, you cannot return an exit code to the parent process or operating system using a **return** statement; to return an exit code when **main** is declared as **void**, you must use the **exit** function.

Argument Definitions

The arguments in the prototype

int main(int *argc*[[**, char** **argv*[ ] [[, **char** **envp*[ ]]]]] **);**

allow convenient command-line parsing of arguments and, optionally, access to environment variables. The argument definitions are as follows:

argc An integer that contains the count of arguments that follow in *argv*. The *argc* parameter is always greater than or equal to 1.

argv An array of null-terminated strings representing command-line arguments entered by the user of the program. By convention, *argv*[**0**] is the command with which the program is invoked, *argv*[**1**] is the first command-line argument, and so on, until *argv*[*argc*], which is always **NULL**.

The first command-line argument is always *argv*[**1**] and the last one is *argv*[*argc* – 1].

envp (Microsoft specific.) The *envp* array, which is a common extension in many UNIX systems, is used in Microsoft C++. It is an array of strings representing the variables set in the user's environment. This array is terminated by a **NULL** entry.

The following example shows how to use the *argc*, *argv*, and *envp* arguments to **main**:

```
/* Program to type out the environment variables.
 * If the /n command-line option is specified,
 * the listing of environment variables is line-
 * numbered.
 */

#include <iostream.h>
#include <string.h>

#define NL "\n"

int main( int argc, char *argv[], char *envp[] )
{
    int iNumberLines = 0;     // Default is no line numbers.

    // If more than .EXE filename supplied, and
    //  user supplies  /n or /N, flag option for line numbers.
    if( argc == 2 && stricmp( argv[1], "/n" ) == 0 )
        iNumberLines = 1;

    // Walk through list of strings until a NULL is encountered.
    for( int i = 0; envp[i] != NULL; ++i )
    {
        if( iNumberLines )
            cout << i;
        cout << ": " << envp[i] << NL;
    }
    return 0;
}
```

Note The `return 0` statement at the end of the program is necessary because **main** is a function declared as returning an **int** value. If no **return** statement is present, or if the **return** statement does not specify a value, an error message is generated.

Wildcard Expansion (Microsoft Specific) You can use wildcards—the question mark (?) and asterisk (*)—to specify filename and path arguments on the command line.

Command-line arguments are handled by a routine called **_setargv**. By default, **_setargv** does not expand wildcards into separate strings in the `argv` string array. However, by linking with the SETARGV.OBJ file, you can replace the default **_setargv** routine with a version that handles wildcards.

To include SETARGV.OBJ, either add it to your project in PWB, or specify it on the CL command line. In either case, the /NOE (no extended dictionary search)

must be supplied to the linker to avoid multiple-definition errors for the **_setargv** function. A sample CL command follows:

```
CL PROG.C \C7\LIB\SETARGV.OBJ /LINK /NOE;
```

The result of the preceding command is that wildcard filenames are expanded in the same manner as MS-DOS commands. (See your MS-DOS user's guide if you are unfamiliar with these characters.) Enclosing a command-line argument in quotation marks (" ") suppresses the wildcard expansion. Within quoted arguments, you can represent quotation marks literally by preceding the double quotation mark with a backslash (\).

Note If you write your own **_setargv** function, it must be specified as **extern "C"** for the linker to recognize it.

If no matches are found for the wildcard argument, the argument is passed literally.

Parsing Command-Line Arguments (Microsoft Specific)

Microsoft C startup code uses the following rules when interpreting arguments given on the MS-DOS command line:

- Arguments are delimited by white space, which is either a space or a tab.
- The caret character (^) is not recognized as an escape character or delimiter. The character is handled completely by the command-line parser in the operating system before being passed to the `argv` array in the program.
- A string surrounded by double quotation marks ("*string*") is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
- A double quotation mark preceded by a backslash (\") is interpreted as a literal double quotation mark character (").
- Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
- If an even number of backslashes is followed by a double quotation mark, then one backslash is placed in the `argv` array for every pair of backslashes, and the double quotation mark is interpreted as a string delimiter.
- If an odd number of backslashes is followed by a double quotation mark, then one backslash is placed in the `argv` array for every pair of backslashes, and the double quotation mark is "escaped" by the remaining backslash, causing a literal double quotation mark (") to be placed in `argv`.

The following program demonstrates how command-line arguments are passed:

```
#include <iostream.h>

int main( int argc,       // Number of strings in array argv
          char *argv[],   // Array of command-line argument strings
          char *envp[] )  // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    cout << "\nCommand-line arguments:\n";
    for( count = 0; count < argc; count++ )
         cout << "  argv[" << count << "]   "
                << argv[count] << "\n";

    return 0;
}
```

Table 2.2 shows example input and expected output, demonstrating the rules in the preceding list.

Table 2.2 Results of Parsing Command Lines

Command-Line Input	argv[1]	argv[2]	argv[3]
"abc"de	abc	d	e
a\\\b d"e f"g h	a\\\b	de fg	h
a\\\"b c d	a\"b	c	d
a\\\\"b c" d e	a\\b c	d	e

Customizing Command-Line Processing (Microsoft Specific) If your program does not take command-line arguments, you can save a small amount of space by suppressing use of the library routine that performs command-line processing. This routine is called **_setargv** and is described in "Wildcard Expansion" on page 39. To suppress its use, define a routine that does nothing in the file containing the **main** function, and name it **_setargv.** The call to **_setargv** is then satisfied by your definition of **_setargv**, and the library version is not loaded.

Similarly, if you never access the environment table through the *envp* argument, you can provide your own empty routine to be used in place of **_setenvp**, the environment-processing routine. Just as with the **_setargv** function, **_setenvp** must be declared as **extern "C"**.

Your program might make calls to the **spawn** or **exec** family of routines in the C run-time library. If this is the case, you should not suppress the environment-processing routine, since this routine is used to pass an environment from the parent process to the child process.

main Function Restrictions

Several restrictions apply to the **main** function that do not apply to any other C++ functions. The **main** function:

- Cannot be overloaded (See Chapter 12, "Overloading").
- Cannot be declared as **inline**.
- Cannot be declared as **static**.
- Cannot have its address taken.
- Cannot be called.

Program Termination

There are several ways to exit a program:

- Call the **exit** function.
- Call the **abort** function.
- Fail an **assert** test.
- Execute a **return** statement from **main**.

exit Function

The **exit** function, declared in the standard include file STDLIB.H, terminates a C++ program.

The value supplied as an argument to **exit** is returned to the operating system as the program's return code or exit code. By convention, a return code of zero means that the program completed successfully.

Note You can use the constants **EXIT_FAILURE** and **EXIT_SUCCESS**, defined in STDLIB.H, to indicate success or failure of your program.

Issuing a **return** statement from the **main** function is equivalent to calling the **exit** function with the return value as its argument.

abort Function

The **abort** function, also declared in the standard include file STDLIB.H, terminates a C++ program. The difference between **exit** and **abort** is that **exit** allows the C run-time termination processing to take place, and **abort** causes immediate program termination.

assert Macro

The **assert** macro allows programmers to insert conditional failure code inline to assist in debugging. Should the programmer's "assertion" prove false, the location of the assertion is printed on the standard output device and the program terminates.

See the *Run-Time Library Reference* manual for more information about using **exit**, **abort**, and **assert** to terminate program execution.

return Statement

Issuing a **return** statement from **main** is functionally equivalent to calling the **exit** function. Consider the following example:

```
int main()
{
    exit( 3 );
    return 3;
}
```

The **exit** and **return** statements in the preceding example are functionally identical. However, C++ requires that functions that have return types other than **void** return a value. The **return** statement allows you to return a value from **main**.

Additional Startup Considerations

In C++, object construction and destruction can involve executing user code. Therefore, it is important to understand which initializations happen before entry to **main**, and which destructors are invoked after exit from **main**. (For detailed information about construction and destruction of objects, see "Constructors" and "Destructors" in Chapter 11, on pages 300 and 305, respectively.)

The following initializations take place prior to entry to **main**:

- Default initialization of static data to zero. All static data without explicit initializers are set to zero prior to executing any other code, including run-time initialization. Static data members must still be explicitly defined.

- Initialization of global static objects in a translation unit. This may occur either before entry to **main** or before the first use of any function or object in the object's translation unit.

Microsoft Specific

In Microsoft C/C++, global static objects are initialized before entry to **main**. ♦

Global static objects that are mutually interdependent but in different translation units may cause incorrect behavior.

Additional Termination Considerations

You can terminate a C++ program by using **exit**, **return**, or **abort**. You can add exit processing using the **atexit** function. These are discussed in the following sections.

Using exit or return

When you call **exit** or execute a **return** statement from **main**, static objects are destroyed in the reverse order of their initialization. This example shows how such initialization and cleanup works:

```
#include <stdio.h>

class ShowData
{
public:
    // Constructor opens a file.
    ShowData( const char *szDev )
    {
        OutputDev = fopen( szDev, "w" );
    }

    // Destructor closes the file.
    ~ShowData() { fclose( OutputDev ); }

    // Disp method shows a string on the output device.
    void Disp( char *szData )
    {
        fputs( szData, OutputDev );
    }
private:
    FILE *OutputDev;
};

//  Define a static object of type ShowData. The output device
//   selected is "CON" -- the standard output device.
ShowData sd1 = "CON";
```

```
//  Define another static object of type ShowData. The output
//   is directed to a file called "HELLO.DAT"
ShowData sd2 = "hello.dat";

int main()
{
    sd1.Disp( "hello to default device\n" );
    sd2.Disp( "hello to file hello.dat\n" );

    return 0;
}
```

In the preceding example, the static objects `sd1` and `sd2` are created and initialized before entry to `main`. After this program terminates using the `return` statement, first `sd2`, then `sd1` is destroyed. The destructor for the `ShowData` class closes the files associated with these static objects. (For more information about initialization, constructors, and destructors, see Chapter 11, "Special Member Functions.")

Note Another way to write this code is to declare the `ShowData` objects with block scope, allowing them to be destroyed when they go out of scope:

```
int main()
{
    ShowData sd1, sd2( "hello.dat" );

    sd1.Disp( "hello to default device\n" );
    sd2.Disp( "hello to file hello.dat\n" );

    return 0;
}
```

Using atexit

The **atexit** function allows you to specify an exit-processing function that executes prior to program termination. No global static objects initialized prior to the call to **atexit** are destroyed prior to execution of the exit-processing function.

Using abort

Calling the **abort** function causes immediate termination. It bypasses the normal destruction process for initialized global static objects. It also bypasses any special processing that was specified using the **atexit** function.

2.6 Storage Classes

Storage classes govern the lifetime, linkage, and treatment of objects and variables in C++. A given object can have only one storage class. The following sections discuss the C++ storage classes for data: automatic, static, register, and external.

Automatic

Objects and variables with automatic storage are local to a given instance of a block. In recursive or multithreaded code, automatic objects and variables are guaranteed to have different storage in different instances of a block. Microsoft C++ stores automatic objects and variables on the program's stack.

Objects and variables defined within a block have **auto** storage unless otherwise specified using the **extern** or **static** keyword. Automatic objects and variables can be specified using the **auto** keyword, but explicit use of **auto** is unnecessary. Automatic objects and variables have no linkage.

Automatic objects and variables persist only until the end of the block in which they are declared.

To optimize the generated code, the compiler may put automatic variables in registers. However, the program always behaves as if automatic variables are allocated on the stack.

Static

Objects and variables declared as **static** retain their values for the duration of the program's execution. In recursive code, a static object or variable is guaranteed to have the same state in different instances of a block of code.

Objects and variables defined outside all blocks have static lifetime and external linkage by default. A global object or variable that is explicitly declared as **static** has internal linkage.

Static objects and variables persist for the duration of the program's execution.

Register

Variables declared as **register** are allocated to a CPU register if there is a suitable register available. These variables behave as automatic variables, except that the compiler gives them preference over automatic variables when allocating register storage.

Only function arguments and local variables can be declared with the register storage class.

Like automatic variables, register variables persist only until the end of the block in which they are declared.

External

Objects and variables declared as **extern** declare an object that is defined in another translation unit or in an enclosing scope as having external linkage.

Declaration of **const** variables with the **extern** storage class forces the variable to have external linkage. An initialization of an **extern const** variable is allowed in the defining translation unit. Initializations in translation units other than the defining translation unit produce undefined results.

The following code shows two **extern** declarations, `DefinedElsewhere` (which refers to a name defined in a different translation unit) and `DefinedHere` (which refers to a name defined in an enclosing scope):

```
extern int DefinedElsewhere;        // Defined in another translation
                                    //  unit.

int main()
{
    int DefinedHere;
    {
       extern int DefinedHere;      // Refers to DefinedHere in
                                    //  the enclosing scope.
    }
    return 0;
}
```

Initialization of Objects

A local automatic object or variable is initialized every time the flow of control reaches its definition. A local static object or variable is initialized the first time the flow of control reaches its definition. Consider the following example, which defines a class that logs initialization and destruction of objects, then defines three objects, `I1`, `I2`, and `I3`:

```
#include <iostream.h>
#include <string.h>

// Define a class that logs initializations and destructions.
class InitDemo
{
public:
    InitDemo( const char *szWhat );
    ~InitDemo();
private:
    char *szObjName;
};

// Constructor for class InitDemo
InitDemo::InitDemo( const char *szWhat )
{
    if( szWhat != 0 && strlen( szWhat ) > 0 )
    {
        // Allocate storage for szObjName, then copy
        //  initializer szWhat into szObjName.
        szObjName = new char[ strlen( szWhat ) + 1 ];
        strcpy( szObjName, szWhat );
            cout << "Initializing: " << szObjName << "\n";
    }
    else
        szObjName = 0;
}

// Destructor for InitDemo
InitDemo::~InitDemo()
{
    if( szObjName != 0 )
    {
        cout << "Destroying: " << szObjName << "\n";
        delete szObjName;
    }
}

// Enter main function
int main()
{
    InitDemo I1( "Auto I1" );
    {
        cout << "In block.\n";
        InitDemo I2( "Auto I2" );
        static InitDemo I3( "Static I3" );
    }
    cout << "Exited block.\n";

    return 0;
}
```

The preceding code demonstrates how and when the objects `I1`, `I2`, and `I3` are initialized and when they are destroyed. The output from the program is:

```
Initializing: Auto I1
In block.
Initializing: Auto I2
Initializing: Static I3
Destroying: Auto I2
Exited block.
Destroying: Auto I1
Destroying: Static I3
```

There are several points to note about the program.

First, `I1` and `I2` are automatically destroyed when the flow of control exits the block in which they are defined.

Second, in C++, it is not necessary to declare objects or variables at the beginning of a block. Furthermore, these objects are initialized only when the flow of control reaches their definitions. (`I2` and `I3` are examples of such definitions.) The output shows exactly when they are initialized.

Finally, static local variables such as `I3` retain their values for the entire duration of the program but are destroyed as the program terminates.

2.7 Types

C++ supports three kinds of object types:

- Fundamental types, which are built into the language (such as **int**, **float**, or **double**). Instances of these fundamental types are often called "variables." These are discussed in "Fundamental Types" on page 50.
- Derived types, which are new types derived from built-in types. These are discussed in "Derived Types" on page 52.
- Class types, which are new types created by combining existing types. These are discussed in Chapter 8, "Classes."

Fundamental Types

Fundamental types in C++ are divided into three categories: "integral," "floating," "void," and "segment." Integral types are capable of handling whole numbers, while floating types are capable of specifying values that may have fractional parts.

The **void** type describes an empty set of values. No variable of type **void** may be specified—it is primarily used in declaring functions that return no values or in declaring "generic" pointers to untyped or arbitrarily typed data. Any expression can be explicitly converted or cast to type **void**. However, such expressions are restricted to the following uses:

- An expression statement. (See Chapter 4, "Expressions," for more information.)
- The left operand of the comma operator. (See "The Comma Operator" in Chapter 4, on page 116 for more information.)
- The second or third operand of the conditional operator (**? :**). (See "Expressions with the Conditional Operator" in Chapter 4, on page 117 for more information.)

The **__segment** type is used only when specifying the segment for a based object or pointer.

Table 2.3 explains the restrictions on type sizes. These restrictions are independent of the Microsoft implementation.

Table 2.3 Fundamental Types of the C++ Language

Category	Type	Contents
Integral	**char**	Type **char** is an integral type that usually contains members of the execution character set—in Microsoft C++, this is ASCII. Variables of type **char** may be declared as **signed char** or **unsigned char**; in either case, they are the same size as a variable declared simply as type **char**. The C++ compiler treats variables of type **char**, **signed char**, and **unsigned char** as having different types. Variables of type **char** are treated as type **signed char** by default, unless the /J compilation option is used, in which case they are treated as type **unsigned char**.
	short	Type **short int** (or simply **short**) is an integral type that is larger than or equal to the size of type **char**, and shorter than or equal to the size of type **int**. Objects of type **short** may be declared as **signed short** or **unsigned short**; in either case, they are the same size as an object declared simply as type **short**. The C++ compiler treats objects of type **short** and **signed short** as different from **unsigned short** .

Table 2.3 *(continued)*

Category	Type	Contents
	int	Type **int** is an integral type that is larger than or equal to the size of type **short int** and shorter than or equal to the size of type **long**.
		Objects of type **int** may be declared as **signed int** or **unsigned int**; in either case, they are the same size as an object declared simply as type **int**. The C++ compiler treats objects of type **int** and **signed int** as different from **unsigned int.**
	long	Type **long** (or **long int**) is an integral type that is larger than or equal to the size of type **int**.
		Objects of type **long** may be declared as **signed long** or **unsigned long**; in either case, they are the same size as objects declared simply as type **long**. The C++ compiler treats objects of type **long** and **signed long** as different from **unsigned long**.
Floating	**float**	Type **float** is the smallest floating type.
	double	Type **double** is a floating type that is larger than or equal to type **float** but shorter than or equal to the size of type **long double**.
	long double	Type **long double** is a floating type that is larger than or equal to type **double**.

In Microsoft C++, variables of various fundamental types require different amounts of storage, depending on whether the program is compiled for a 16- or 32-bit target. Table 2.4 shows these differences.

Table 2.4 Sizes of Fundamental Types

Type	16-bit Target Compilation	32-bit Target Compilation
char, unsigned char, signed char	1 byte	1 byte
short, short int, signed short, unsigned short	2 bytes	2 bytes
int, unsigned int, signed int	2 bytes	4 bytes
long, long int, unsigned long, signed long	4 bytes	4 bytes
float	4 bytes	4 bytes
double	8 bytes	8 bytes
long double	10 bytes	10 bytes

For more information about type conversion, see Chapter 3, "Standard Conversions."

Derived Types

Derived types are new types that can be used in a program. They can conceptually be divided into types that are directly derived and types that are composed of other types. Both types are discussed in this section.

Directly Derived Types

New types derived directly from existing types are types that point to, refer to, or (in the case of functions) transform type data to return a new type. These types are discussed in the sections that follow.

Arrays of Variables or Objects Arrays of variables or objects can contain a specified number of a particular type. For example, an array derived from integers is an array of type **int**. The following code sample declares and defines an array of 10 **int** variables and an array of 5 objects of class `SampleClass`:

```
int         ArrayOfInt[10];
SampleClass aSampleClass[5];
```

Functions Functions take zero or more arguments of given types and return objects of a specified type (or return nothing, if the function has a **void return** type).

Pointers of a Given Type Pointers to variables or objects select an object in memory. The object can be global, local (or stack-frame), or dynamically allocated. Pointers to functions of a given type allow a program to defer selection of the function used on a particular object or objects until run time. The following example shows declaration and definition of a pointer to a variable of type **char**:

```
char *szPathStr = new char[_MAX_PATH];
```

References to Objects References to objects provide a convenient way to access objects by reference but use the same syntax required to access objects by value. The following example demonstrates how to use references as arguments to functions, and as return types of functions:

```
BigClassType &func( BigClassType &objname )
{
    objname.DoSomething();     // Note that member-of operator(.)
                               //  is used.
    objname.SomeData = 7;      // Data passed by non-const
                               //  reference is modifiable.
    return objname;
}
```

The important points about passing objects to a function by reference are:

- Public member data can be read or modified. See Chapter 10, "Member-Access Control," for information about access specifiers such as **public**.
- The syntax for accessing members of **class**, **struct**, and **union** objects is the same as if they were passed by value: the member-of operator (.).
- The objects are not copied prior to the function call; their addresses are passed. This can reduce the overhead of the function call.

Additionally, functions that return a reference need only return the address of the object to which they refer, instead of a copy of the whole object.

Although the preceding example describes references only in the context of communication with functions, references are not constrained to this use. Consider, for example, a case where a function needs to be an l-value—a common requirement for overloaded operators:

```
class Vector
{
public:
    Point &operator[]( int nSubscript ); // Function returns a
                                         //  reference type
    ...
};
```

The preceding declaration specifies a user-defined subscript operator for class `Vector`. In an assignment statement, two possible conditions occur:

```
Vector v1;
int    i;
Point p;
v1[7] = p;   // Vector used as an l-value
p = v1[7];   // Vector used as an r-value
```

The latter case, where `v1[7]` is used as an r-value, can be implemented without use of references. However, the former case, where `v1[7]` is used as an l-value, cannot be implemented easily without functions that are of reference type. Conceptually, the last two statements in the preceding example translate to the following code:

```
v1.operator[]( 7 ) = 3;     // Vector used as an l-value
i = v1.operator[]( 7 );     // Vector used as an r-value
```

When viewed in this way, it is easier to see that the first statement must be an l-value to be semantically correct on the left side of the assignment statement.

For more information about overloading, and about overloaded operators in particular, see "Overloaded Operators" in Chapter 12, on page 351.

Another use for references is in declaring a **const** reference to a variable or object. A reference declared as **const** retains the efficiency of passing an argument by reference, while preventing the called function from modifying the original object. Consider the following code:

```
// IntValue is a const reference.
void PrintInt( const int &IntValue )
{
    printf( "%d\n", IntValue );
}
```

Reference initialization is different from assignment to a variable of reference type. Consider the following code:

```
int i = 7;
int j = 5;

// Reference initialization
int &ri = i;  // Initialize ri to refer to i.
int &rj = j;  // Initialize rj to refer to j.

// Assignment
ri = 3;       // i now equal to 3.
rj = 12;      // j now equal to 12.
ri = rj;      // i now equals j (12).
```

Constants See "Literals" in Chapter 1, on page 14 for more information about the various kinds of constants allowed in C++.

Pointers to Class Members These pointers define a type that points to a class member of a particular type. Such a pointer can be used by any object of the class type or any object of a type derived from the class type.

Use of pointers to class members enhances the type safety of the C++ language. Several new operators and constructs are used with pointers to members, as shown in Table 2.5.

Table 2.5 Operators and Constructs Used with Pointers to Members

Operator or Construct	Syntax	Use
::*	*type*::**ptr-name*	Declaration of pointer to member. The *type* specifies the class name, and *ptr-name* specifies the name of the pointer to member. Pointers to members may be initialized. For example: `MyType::*pMyType = &MyType::i;`
.*	*obj-name*.**ptr-name*	Dereference a pointer to a member using an object. For example: `int j = Object.*pMyType;`
->*	*obj-ptr*->**ptr-name*	Dereference a pointer to a member using a pointer to an object. For example: `int j = pObject->*pMyType;`

Consider this example that defines a class `AType` and the derived type `pDAT`, which points to the member `I1`:

```
#include <iostream.h>

// Define class AType.
class AType
{
public:
    int I1;
    Show() { cout << I1 << "\n"; }
};

// Define a derived type pDAT that points to I1 members of
// objects of type AType.
int AType::*pDAT = &AType::I1;
```

```
int main()
{
    AType aType;              // Define an object of type AType.
    AType *paType = &aType;   // Define a pointer to that object.

    int i;

    aType.*pDAT = 7777;       // Assign to aType::I1 using .* operator.
    aType.Show();

    i = paType->*pDAT;        // Dereference a pointer using .->
operator.
    cout << i << "\n";


    return 0;
}
```

The pointer to member `pDAT` is a new type derived from class `AType`. It is more strongly typed than a "plain" pointer because it points only to **int** members of class `AType` (in this case, `I1`). Pointers to static members are plain pointers rather than pointers to class members. Consider the following example:

```
class HasStaticMember
{
public:
    static int SMember;
};
int HasStaticMember::SMember = 0;

int *pSMember = &HasStaticMember::SMember;
```

Note that the type of the pointer is "pointer to **int**," and not "pointer to `HasStaticMember::int`."

Pointers to members can refer to member functions as well as member data. Consider the following code:

```
#include <stdio.h>

// Declare a base class, A, with a virtual function, Identify.
// (Note that in this context, struct is the same as class.)
struct A
{
    virtual void Identify() = 0; // No definition for class A.
};
```

```
// Declare a pointer to the Identify member function.
void (A::*pIdentify)() = &A::Identify;

// Declare class B derived from class A.
struct B : public A
{
    void Identify();
};

// Declare class C derived from class A.
struct C : public A
{
    void Identify();
};

// Define Identify functions for classes B and C.
void B::Identify()
{
    printf( "Identification is B::Identify\n" );
}

void C::Identify()
{
    printf( "Identification is C::Identify\n" );
}

int main()
{
    B  BObject;          // Declare objects of type B
    C  CObject;          //  and type C.
    A *pA;               // Declare pointer to type A.

    pA = &BObject;       // Make pA point to an object of type B.
    (pA->*pIdentify)();  // Call Identify function through pointer
                         //  to member pIdentify.
    pA = &CObject;       // Make pA point to an object of type C.
    (pA->*pIdentify)();  // Call Identify function through pointer
                         //  to member pIdentify.

    return 0;
}
```

The output from this program is:

```
Identification is B::Identify
Identification is C::Identify
```

The function is called through a pointer to type `A`. However, because the function is a virtual function, the correct function for the object to which `pA` refers is called.

Composed Derivative Types

The following sections discuss composed derivative types. Information about aggregate types and initialization of aggregate types can be found in "Initializing Aggregates" in Chapter 7, on page 219.

Classes Classes are a composite group of member objects, functions to manipulate these members, and (optionally) access-control specifications to member objects and functions.

By grouping composite groups of objects and functions in classes, C++ allows programmers to create derivative types that define not only data but also the behavior of objects.

Class members default to private access and private inheritance. Classes are covered in Chapter 8, "Classes"; access control is covered in Chapter 10, "Member-Access Control."

Structures C++ structures are the same as classes, except that all member data and functions default to public access, and inheritance defaults to public inheritance.

For more information about access control, see Chapter 10, "Member-Access Control."

Unions Unions allow programmers to define types capable of containing different kinds of variables in the same memory space. The following code shows how you can use a union to store several different types of variables:

```
//  Declare a union that can hold data of types char, int, long,
//   float, double, or char *.
union ToPrint
{
    char   chVar;
    int    iVar;
    long   lVar;
    float  fVar;
    double dVar;
    char  *szVar;
};
```

```
// Declare an enumerated type that describes what type to print.
enum PrintType { CHAR_T, INT_T, LONG_T,
                 FLOAT_T, DOUBLE_T, STRING_T };

void Print( ToPrint Var, PrintType Type )
{
    switch( Type )
    {
    case CHAR_T:
        printf( "%c", Var.chVar );
        break;
    case INT_T:
        printf( "%d", Var.iVar );
        break;
    case LONG_T:
        printf( "%ld", Var.lVar );
        break;
    case FLOAT_T:
        printf( "%d", Var.fVar );
        break;
    case DOUBLE_T:
        printf( "%f", Var.dVar );
        break;
    case STRING_T:
        printf( "%s", Var.szVar );
        break;
    }
}
```

Type Names

Synonyms for both fundamental and derived types can be defined using the **typedef** keyword. The following code illustrates the use of **typedef**:

```
typedef unsigned char BYTE;   // 8-bit unsigned entity.
typedef BYTE *        PBYTE;  // Pointer to BYTE.

BYTE Ch;                      // Declare a variable of type BYTE.
PBYTE pbCh;                   // Declare a pointer to a BYTE
                              //  variable.
```

The preceding example shows uniform declaration syntax for the fundamental type **unsigned char** and its derivative type **unsigned char ***. The **typedef** construct is also helpful in simplifying declarations. The following example declares a type name (PVFN) representing a pointer to a function that returns type **void**. The advantage of this declaration is that, later in the program, an array of these pointers is declared very simply.

```
#include <iostream.h>
#include <stdlib.h>

// Prototype two functions.
void func1();
void func2();

//  Define PVFN to represent a pointer to a function that
//   returns type void.
typedef void (*PVFN)();

...

// Declare an array of pointers to functions.
PVFN pvfn[] = { func1, func2 };

// Invoke one of the functions.
(*pvfn[1])();
```

2.8 L-Values and R-Values

Expressions in C++ can evaluate to "l-values" or "r-values." L-values are expressions that evaluate to a type other than **void** and that designate a variable.

L-values appear on the left side of an assignment statement (hence the "l" in l-value). Variables that would normally be l-values can be made nonmodifiable by using the **const** keyword; these may not appear on the left of an assignment statement.

Reference types are always l-values.

Some examples of correct and incorrect usages are:

```
i = 7;              // Correct. A variable name, i, is an l-value.
7 = i;              // Error. A constant, 7, is an r-value.
j * 4 = 7;          // Error. The expression j * 4 yields an r-value.
*p = i;             // Correct. A dereferenced pointer is an l-value.
const int ci = 7; // Declare a const variable.
ci = 9;             // ci is a nonmodifiable l-value, so the
                    //  assignment causes an error message to
                    //  be generated.
((i < 3) ? i :      // Correct. Conditional operator (? :)
          j) = 7; //  returns an l-value.
```

Note The previous example illustrates correct and incorrect usage when operators are not overloaded. By overloading operators, you can make an expression such as `j * 4` an l-value.

2.9 Name Spaces

"Name space" refers to the place the compiler keeps symbols used to refer to various program elements. The place a symbol is kept influences whether two program symbols will conflict. C has two name spaces, but C++ maintains only one name space.

The two C name spaces are:

- Variable, function, **typedef**, and enumerator name space
- Structure, enumeration, and union tag name space

In C++, all these names share a single name space.

2.10 Numerical Limits

The two standard include files, LIMITS.H and FLOAT.H, define the "numerical limits," or minimum and maximum values, a variable of a given type can hold. These minimums and maximums are guaranteed to be portable to any C++ compiler that uses the same data representation as ANSI C. The LIMITS.H include file defines the numerical limits for integral types, and FLOAT.H defines the numeric limits for floating types.

Integral Limits

The limits (constant names, meanings, and values) for integral types are defined in the standard include file LIMITS.H. They are shown in Table 2.6.

Table 2.6 Limits for Integral Types

Constant	Meaning	Value
CHAR_BIT	Number of bits in the smallest variable that is not a bit field.	8
SCHAR_MIN	Minimum value for a variable of type **signed char**.	–127
SCHAR_MAX	Maximum value for a variable of type **signed char**.	127
UCHAR_MAX	Maximum value for a variable of type **unsigned char**.	255 (0xff)
CHAR_MIN	Minimum value for a variable of type **char**.	Same as –127; 0 if /J option used.
CHAR_MAX	Maximum value for a variable of type **char**.	Same as 127; 255 if /J option used.
MB_LEN_MAX	Maximum number of bytes in a multicharacter constant.	2
SHRT_MIN	Minimum value for a variable of type **short**.	–32767
SHRT_MAX	Maximum value for a variable of type **short**.	32767
USHRT_MAX	Maximum value for a variable of type **unsigned short**.	65535 (0xffff)
INT_MIN	Minimum value for a variable of type **int**[1].	–32767
INT_MAX	Maximum value for a variable of type **int**[2].	32767
UINT_MAX	Maximum value for a variable of type **unsigned int**[3].	65535 (0xffff)
LONG_MIN	Minimum value for a variable of type **long**.	–2147483647
LONG_MAX	Maximum value for a variable of type **long**.	2147483647
ULONG_MAX	Maximum value for a variable of type **unsigned long**.	4294967295 (0xffffffff)

[1]The value for **INT_MIN** is –2147483648 for 32-bit target compilations

[2]The value for **INT_MAX** is 2147483647 for 32-bit target compilations

[3]The value for **UINT_MAX** is 4294967295 (0xffffffff) for 32-bit target compilations

Floating Limits

The limits (constant names, meanings, and values) for floating types are defined in the standard include file FLOAT.H. They are:

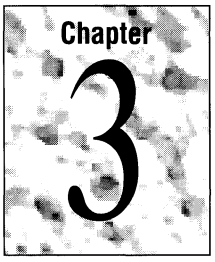
Table 2.7 Limits for Floating Types

Constant	Meaning	Value
FLT_DIG **DBL_DIG** **LDBL_DIG**	Number of digits, *q*, such that a floating-point number with *q* decimal digits can be rounded into a floating-point representation and back, without loss of precision.	7 15 19
FLT_EPSILON **DBL_EPSILON** **LDBL_EPSILON**	Smallest positive number *x*, such that $x \neq 1.0 + x$	1.192092896e-07F 2.2204460492503131e-016 5.4210108624275221706e-020
FLT_MANT_DIG **DBL_MANT_DIG** **LDBL_MANT_DIG**	Number of digits in the radix specified by **FLT_RADIX** in the floating-point significand. In Microsoft C/C++, the radix is 2; hence these values specify bits.	24 53 64
FLT_MAX **DBL_MAX** **LDBL_MAX**	Maximum representable floating-point number.	3.402823466e+38F 1.7976931348623158e+308 1.189731495357231765e+4932L
FLT_MAX_10_EXP **DBL_MAX_10_EXP** **LDBL_MAX_10_EXP**	Maximum integer such that 10 raised to that number is a representable floating-point number.	38 308 4932
FLT_MAX_EXP **DBL_MAX_EXP** **LDBL_MAX_EXP**	Maximum integer such that **FLT_RADIX** raised to that number is a representable floating-point number.	128 1024 16384
FLT_MIN **DBL_MIN** **LDBL_MIN**	Minimum positive normalized floating-point number.	1.175494351e-38F 2.2250738585072014e-308 3.3621031431120935063e-4932L

Table 2.7 *(continued)*

Constant	Meaning	Value
FLT_MIN_10_EXP **DBL_MIN_10_EXP** **LDBL_MIN_10_EXP**	Minimum negative integer such that 10 raised to that number is a representable floating-point number.	–37 –307 –4931
FLT_MIN_EXP **DBL_MIN_EXP** **LDBL_MIN_EXP**	Minimum negative integer such that **FLT_RADIX** raised to that number is a representable floating-point number.	–125 –1021 –16381
FLT_RADIX **DBL_RADIX** **LDBL_RADIX**	Radix of exponent representation.	2 2 2
FLT_ROUNDS **DBL_ROUNDS** **LDBL_ROUNDS**	Rounding mode for floating-point addition.	1 (near) 1 (near) 1 (near)

Standard Conversions



The C++ language defines conversions between its fundamental types. It also defines conversions for pointer, reference, and pointer-to-member derived types. These conversions are called "standard conversions." (For more information about types, standard types, and derived types, see "Types" in Chapter 2, on page 49.)

This chapter discusses the following standard conversions:

- Integral promotions
- Integral conversions
- Floating conversions
- Floating and integral conversions
- Arithmetic conversions
- Pointer conversions
- Reference conversions
- Pointer-to-member conversions

Note User-defined types can specify their own conversions. Conversion of user-defined types is covered in "Constructors" and "Conversions" in Chapter 11, on pages 300 and 312, respectively.

The following code causes conversions (in this example, integral promotions):

```
long  lnum1, lnum2;
int   inum;

// inum promoted to type long prior to assignment.
lnum1 = inum;

// inum promoted to type long prior to
//  multiplication.
lnum2 = inum * lnum2;
```

Note The result of a conversion is an l-value only if it produces a reference type. For example, a user-defined conversion declared as

```
MyType &operator int()
```

returns a reference and is an l-value. However, a conversion declared as

```
MyType operator int()
```

returns an object and is not an l-value.

3.1 Integral Promotions

Objects of an integral type can be converted to another wider integral type (that is, a type that can represent a larger set of values). This widening type of conversion is called "integral promotion." Integral promotion allows the following to be used in an expression wherever another integral type can be used:

- Objects, literals, and constants of type **char** and **short int**
- Enumeration types
- **int** bit fields
- Enumerators

C++ promotions are "value-preserving." That is, the value after the promotion is guaranteed to be the same as the value before the promotion. In value-preserving promotions, objects of shorter integral types (such as bit fields or objects of type **char**) are promoted to type **int** if **int** can represent the full range of the original type. If **int** cannot represent the full range of values, then the object is promoted to type **unsigned int**. While this strategy is the same as that used by ANSI C, value-preserving conversions do not preserve the "signedness" of the object.

Value-preserving promotions and promotions that preserve signedness normally produce the same results. However, they can produce different results if the promoted object is one of the following:

- An operand of /, %, /=, %=, <, <=, >, or >=

  These operators rely on sign for determining the result. Therefore, value-preserving and sign-preserving promotions produce different results when applied to these operands.

- The left operand of >> or >>=

  These operators treat signed and unsigned quantities differently when performing a shift operation. For signed quantities, shifting a quantity right causes the sign bit to be propagated into the vacated bit positions. For unsigned quantities, the vacated bit positions are zero-filled.

- An argument to an overloaded function or operand of an overloaded operator that depends on the sign of that operand for argument matching. (See "Overloaded Operators" in Chapter 12, on page 351 for more about defining overloaded operators.)

3.2 Integral Conversions

Integral conversions are performed between integral types. The integral types are **char**, **int**, and **long** (and the **short**, **signed**, and **unsigned** versions of these types).

Converting Signed to Unsigned

Objects of signed integral types can be converted to corresponding unsigned types. When these conversions occur, the actual bit pattern does not change; however, the interpretation of the data changes. Consider this code:

```
#include <iostream.h>

int main()
{
    short  i = -3;
    unsigned short u;

    cout << (u = i) << "\n";

    return 0;
}
```

The following output results:

```
65533
```

In the preceding example, a **signed short**, `i`, is defined and initialized to a negative number. The expression `(u = i)` causes `i` to be converted to an **unsigned short** prior to the assignment to `u`.

Converting Unsigned to Signed

Objects of unsigned integral types can be converted to corresponding signed types. However, such a conversion can cause misinterpretation of data if the value of the

unsigned object is outside the range representable by the signed type, as demonstrated in the following example:

```
#include <iostream.h>

void main()
{
 short  i;
 unsigned short u = 65533;

 cout << (i = u) << "\n";
}
```

The following output results:

```
-3
```

In the preceding example, `u` is an **unsigned short** integral object that must be converted to a signed quantity to evaluate the expression `(i = u)`. Because its value cannot be properly represented in a **signed short**, the data is misinterpreted as shown.

Standard Conversion

Objects of integral types can be converted to shorter signed or unsigned integral types. However, this can result in loss of data if the value of the original object is outside the range representable by the shorter type. Such a conversion is called "standard conversion."

Note The compiler issues a high-level warning when a conversion to a shorter type takes place.

3.3 Floating Conversions

An object of a floating type can be safely converted to a more precise floating type—that is, the conversion causes no loss of significance. For example, conversions from **float** to **double** or from **double** to **long double** are safe, and the value is unchanged.

An object of a floating type can also be converted to a less precise type, if it is in a range representable by that type. (See "Floating Limits" in Chapter 2, on page 63 for the ranges of floating types.) If the original value cannot be represented precisely, it can be converted to either the next higher or the next lower representable value. If no such value exists, the result is undefined. Consider the following example:

```
cout << (float)1E300 << endl;
```

The maximum value representable by type **float** is 3.402823466E38—a much smaller number than 1E300. Therefore, the number is converted to infinity, and the result is 1.#INF.

3.4 Floating and Integral Conversions

Certain expressions can cause objects of floating type to be converted to integral types, or vice versa.

Floating to Integral

When an object of floating type is converted to an integral type, the fractional part is truncated. No rounding takes place in the conversion process. Truncation means that a number like 1.3 is converted to 1, and –1.3 is converted to –1.

Integral to Floating

When an object of integral type is converted to a floating type and the original value cannot be represented exactly, the result is either the next higher or next lower representable value.

3.5 Arithmetic Conversions

Many binary operators (discussed in "Expressions with Binary Operators" in Chapter 4, on page 102) cause conversions of operands and yield results the same way. The way these operators cause conversions is called "usual arithmetic conversions." Arithmetic conversions of operands of different types are performed as shown in Table 3.1.

Table 3.1 Conditions for Type Conversion

Conditions Met	Conversion
Either operand is of type **long double**.	Other operand is converted to type **long double**.
Preceding condition not met and either operand is of type **double**.	Other operand is converted to type **double**.
Preceding conditions not met and either operand is of type **float**.	Other operand is converted to type **float**.

Table 3.1 *(continued)*

Conditions Met	Conversion
Preceding conditions not met (none of the operands are of floating types).	Integral promotions are performed on the operands as follows: ▪ If either operand is of type **unsigned long**, the other operand is converted to type **unsigned long**. ▪ If preceding condition not met, and if either operand is of type **long** and the other of type **unsigned int**, the operand of type **unsigned int** is converted to type **long** (in 16-bit compilations) or both operands are converted to type **unsigned long** (in 32-bit compilations). ▪ If the preceding two conditions are not met, and either operand is of type **long**, the other operand is converted to type **long**. ▪ If the preceding three conditions are not met, and either operand is of type **unsigned int**, the other operand is converted to type **unsigned int**. ▪ If none of the preceding conditions are met, both operands are converted to type **int**.

The following code illustrates the conversion rules described in Table 3.1:

```
float   fVal;
double  dVal;
int   iVal;
unsigned long ulVal;

dVal = iVal * ulVal; // iVal converted to unsigned long;
                     // result of multiplication converted to double.

dVal = ulVal + fVal; // ulVal converted to float;
                     // result of addition converted to double.
```

The first statement in the preceding example shows multiplication of two integral types, `iVal` and `ulVal`. The condition met is that neither operand is of floating type and one operand is of type **unsigned int**. Therefore, the other operand, `iVal`, is converted to type **unsigned int**. The result is assigned to `dVal`. The condition met is that one operand is of type **double**; therefore, the **unsigned int** result of the multiplication is converted to type **double**.

The second statement in the preceding example shows addition of a **float** and an integral type, `fVal` and `ulVal`. The `ulVal` variable is converted to type **float** (third condition in Table 3.1). The result of the addition is converted to type **double** (second condition in Table 3.1) and assigned to `dVal`.

3.6 Pointer Conversions

Pointers can be converted during assignment, initialization, comparison, and other expressions. These conversions are described in the following sections.

Null Pointers

An integral constant expression that evaluates to zero, or such an expression cast to type **void** *, is converted to a pointer called the "null pointer." This pointer is guaranteed to compare unequal to a pointer to any valid object or function (except for pointers to based objects, which can have the same offset and still point to different objects).

Pointers to Type void

Pointers to type **void** can be converted to pointers to any other type, but only with an explicit type cast. (See "Expressions with Explicit Type Conversions" in Chapter 4, on page 119 for more information about type casts). A pointer to any type can be converted implicitly to a pointer to type **void**.

A pointer to an incomplete object of a type can be converted to a pointer to **void** and back. The result of such a conversion is equal to the value of the original pointer. An incomplete object is an object that is declared, but for which insufficient information is available to determine its size.

Pointers to Objects

A pointer to any object that is not **const** or **volatile** can be converted to a pointer of type **void** *.

Pointers to Functions

A pointer to a function can be converted to type **void** *, if type **void** * is large enough to hold that pointer.

Microsoft Specific In medium-model 16-bit target compilations, the data pointer size is 2 bytes, and the code pointer size is 4 bytes. Therefore, a pointer to type **void** is too small to hold a pointer to a function. ♦

Pointers to Classes

There are two cases in which a pointer to a class can be converted to a pointer to a base class.

The first case is when the specified base class is accessible and the conversion is unambiguous. (See "Multiple Base Classes" in Chapter 9, on page 267 for more information about ambiguous base-class references.)

Whether a base class is accessible depends on the kind of inheritance used in derivation. Consider the inheritance situation illustrated in Figure 3.1.

Figure 3.1 Inheritance Graph for Illustration of Base-Class Accessibility

Table 3.2 shows the base-class accessibility for the situation illustrated in Figure 3.1.

Table 3.2 Base-Class Accessibility

Type of Function	Derivation	Conversion from B* to A* Legal?
External (not class-scoped) function	Private	No
	Protected	No
	Public	Yes
B member function (in B scope)	Private	Yes
	Protected	Yes
	Public	Yes
C member function (in C scope)	Private	No
	Protected	Yes
	Public	Yes

The second case in which a pointer to a class can be converted to a pointer to a base class is when you use an explicit type conversion. (See "Expressions with Explicit Type Conversions" in Chapter 4, on page 119 for more information about explicit type conversions.

The result of such a conversion is a pointer to the "subobject," the portion of the object that is completely described by the base class.

The following code defines two classes, `A` and `B`, where `B` is derived from `A`. (For more information on inheritance, see Chapter 9, "Derived Classes.") It then defines `bObject`, an object of type `B`, and two pointers (`pA` and `pB`) that point to the object.

```
class A
{
public:
    int AComponent;
    int AMemberFunc();
};

class B : public A
{
public:
    int BComponent;
    int BMemberFunc();
};

B bObject;
A *pA = &bObject;
B *pB = &bObject;

pA->AMemberFunc();   // OK in class A
pB->AMemberFunc();   // OK: inherited from class A
pA->BMemberFunc();   // Error: not in class A
```

The pointer `pA` is of type `A *`, which can be interpreted as meaning "pointer to an object of type `A`." Members of `bObject` (such as `BComponent` and `BMemberFunc`) are unique to type `B` and are therefore inaccessible through `pA`. The `pA` pointer allows access only to those characteristics (member functions and data) of the object that are defined in class `A`.

Expressions

Any expression with an array type can be converted to a pointer of the same type. The result of the conversion is a pointer to the first array element. The following example demonstrates such a conversion:

```
char szPath[_MAX_PATH]; // Array of type char.
char *pszPath = szPath; // Equals &szPath[0].
```

An expression that results in a function returning a particular type is converted to a pointer to a function returning that type, except when:

- The expression is used as an operand to the address-of operator (**&**).
- The expression is used as an operand to the function-call operator.

Pointers Modified by Microsoft Keywords

The Microsoft keywords **__near**, **__far**, **__huge**, and **__based** modify types to specify the addressing desired. The following standard conversions between pointers modified by these keywords are performed (provided the types obey the conversion rules discussed elsewhere in this chapter):

- A near pointer can be promoted to a far pointer.
- Any pointer can be converted to a huge pointer by first converting the pointer to a far pointer, then converting it to a huge pointer.
- A huge pointer can be converted to a far pointer. Because far addressing has different implications than huge addressing, the compiler issues a warning.
- A pointer based on a near address can be converted to a near pointer.
- A pointer based on any segment other than **void** can be converted to a far pointer.

Microsoft C++ supplies no standard conversions to any form of based pointer or from any form of address that contains or implies segment information (far, huge, or based) to a near pointer.

C++ does not supply a standard conversion from a **const** or **volatile** type to a type that is not **const** or **volatile**. However, any sort of conversion can be specified using explicit type casts (including conversions that are unsafe).

Note C++ pointers to members, with the exception of pointers to static members, are different from normal pointers and do not have the same standard conversions. Pointers to static members are normal pointers and have the same conversions as normal pointers. (See "Pointers to Class Members" in Chapter 2, on page 55 for more information.)

3.7 Reference Conversions

A reference to a class can be converted to a reference to a base class in the following cases:

- The specified base class is accessible (as defined in "Pointers to Classes" on page 72).
- The conversion is unambiguous. (See "Multiple Base Classes" in Chapter 9, on page 267 for more information about ambiguous base-class references.)

The result of the conversion is a pointer to the subobject that represents the base class.

For more information about references, see "References to Objects" in Chapter 2, on page 53.

3.8 Pointer-to-Member Conversions

Pointers to class members can be converted during assignment, initialization, comparison, and other expressions. These conversions are discussed in the next two sections.

Integral Constant Expressions

An integral constant expression that evaluates to zero is converted to a pointer called the "null pointer." This pointer is guaranteed to compare unequal to a pointer to any valid object or function (except for pointers to based objects, which can have the same offset and still point to different objects).

The following code illustrates the definition of a pointer to member `i` in class `A`. The pointer, `pai`, is initialized to 0, which is the null pointer.

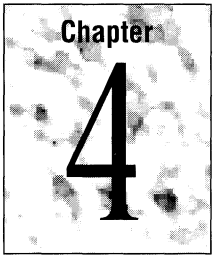
```
class A
{
public:
 int i;
};

int A::*pai = 0;
```

Pointers to Base-Class Members

A pointer to a member of a base class can be converted to a pointer to a member of a class derived from it, when the following conditions are met:

- The inverse conversion, from derived class to base class pointer, is accessible.
- The derived class does not inherit virtually from the base class.

Expressions



This section describes C++ expressions. Expressions are sequences of operators and operands that are used for one or more of these purposes:

- Computing a value from the operands.
- Designating objects or functions.
- Generating "side effects." (Side effects are any actions other than the evaluation of the expression—for example, modifying the value of an object.)

In C++, operators can be overloaded and their meanings can be user-defined. However, their precedence and the number of operands they take cannot be modified. This chapter describes the syntax and semantics of operators as they are supplied with the language, not overloaded. (For more information about overloaded operators, see "Overloaded Operators" in Chapter 12, on page 351.)

Note Operators for built-in types cannot be overloaded; their behavior is predefined.

4.1 Types of Expressions

C++ expressions are divided into several categories:

- Primary expressions. These are the building blocks from which all other expressions are formed. (See "Primary Expressions" on page 78.)
- Postfix expressions. These are primary expressions followed by an operator—for example, the array subscript or postincrement operator. (See "Postfix Expressions" on page 81.)
- Expressions formed with unary operators. Unary operators act on only one operand in an expression. (See "Expressions with Unary Operators" on page 91.)
- Expressions formed with binary operators. Binary operators act on two operands in an expression. (See "Expressions with Binary Operators" on page 102.)

- Expressions formed with the conditional operator. The conditional operator is a ternary operator—the only such operator in the C++ language—and takes three operands. (See "Expressions with the Conditional Operator" on page 117.)
- Constant expressions. Constant expressions are formed entirely of constant data. (See "Constant Expressions" on page 118.)
- Expressions with explicit type conversions. Explicit type conversions, or "casts" can be used in expressions. (See "Expressions with Explicit Type Conversions" on page 119.)
- Expressions with pointer-to-member operators. (See "Expressions with Pointer-to-Member Operators" on page 124.)

Primary Expressions

Primary expressions are the building blocks of more complex expressions. They are literals, names, and names qualified by the scope-resolution operator (**::**).

Syntax

primary-expression:
literal
this
:: *identifier*
:: *operator-function-name*
:: *qualified-name*
(*expression*)
name

A *literal* is a constant primary expression. Its type depends on the form of its specification. See "Literals" in Chapter 1, on page 14 for complete information about specifying literals.

The **this** keyword is a pointer to a class object. It is available within nonstatic member functions and points to the instance of the class for which the function was invoked. The **this** keyword cannot be used outside the body of a class-member function.

The type of the **this** pointer is *type* ***const** (where *type* is the class name) within functions not specifically modifying the **this** pointer. The following example shows member function declarations and the types of **this**:

```
class Example
{
public:
    void Func();          // Example: * const this
    void Func() const;    // Example: const * const this
    void Func() volatile; // Example: volatile * const this
};
```

See "Type of this Pointer" in Chapter 8, on page 245 for more information about modifying the type of the **this** pointer.

Microsoft Specific

The **this** pointer has an associated addressing model—from the compilation options, the "ambient data model" of the class, or one that is explicitly specified in the declaration of an object of class type. Therefore, the **this** pointer can be a near, far, or huge pointer. For more information about addressing models as they affect classes, see "Memory-Model Modifiers and Member Functions" in Appendix B, on page 400.♦

The scope-resolution operator (**::**), followed by an *identifier*, *operator-function-name*, or *qualified-name* constitutes a primary expression. The type of this expression is determined by the declaration of the *identifier*, *operator-function-name*, or *name*. It is an l-value if the declaring name is an l-value. The scope-resolution operator allows a global name to be referred to, even if that name is hidden in the current scope. See "Scope" in Chapter 2, on page 28 for an example of how to use the scope-resolution operator.

An expression enclosed in parentheses is a primary expression whose type and value are identical to those of the unparenthesized expression. It is an l-value if the unparenthesized expression is an l-value.

Names

A name is a primary expression that can appear only after the member-selection operators (**.** or **->**).

Syntax

name:
identifier
operator-function-name
conversion-function-name
~ class-name
qualified-name

Any *identifier* that has been declared is a *name*.

An *operator-function-name* is a name that is declared in the form:

operator operator-name(*argument 1* [[, *argument 2*]]);

See "Overloaded Operators" in Chapter 12, on page 351 for more information about declaration of *operator-function-name.*

A *conversion-function-name* is a name that is declared in the form:

operator type-name( )

Note You can supply a derivative type name, such as **char** * in place of the *type-name* when declaring a conversion function.

Conversion functions supply conversions to and from user-defined types. For more information about user-supplied conversions, see "Conversion Functions" in Chapter 11, on page 315.

A name declared as ~ *class-name* is taken as the "destructor" for objects of a class type. Destructors typically perform cleanup operations at the end of an object's lifetime. Destructors are discussed in-depth in "Destructors" in Chapter 11, on page 305.

Qualified Names

Syntax

qualified-name:
  qualified-class-name **::** *name*

If a *qualified-class-name* is followed by the scope-resolution operator (**::**), and then the name of a member of either that class or a base of that class, then the scope-resolution operator is considered a *qualified-name.* The type of a *qualified-name* is the same as the type of the member, and the result of a *qualified-name* expression is the member. If the member is an l-value, then the *qualified-name* is also an l-value. For information about declaring *qualified-class-name*, see "Type Specifiers" in Chapter 6, on page 168 or "Class Names" in Chapter 8, on page 232.

The *class-name* part of a *qualified-class-name* can be hidden by redeclaration of the same name in the current or enclosing scope; the *class-name* is still found and used. See "Scope" in Chapter 2, on page 28 for an example of how to use a *qualified-class-name* to access a hidden *class-name.*

Note Class constructors and destructors of the form *class-name* **::** *class-name*, and *class-name* **::** ~ *class-name*, respectively, must refer to the same *class-name.*

A multiply-qualified name, such as the following, designates a member of a nested class:

class-name **::** *class-name* **::** *name*

Postfix Expressions

Postfix expressions consist of primary expressions or expressions in which postfix operators (see Table 4.1) follow a primary expression.

Table 4.1 Postfix Operators

Operator Name	Operator Notation
Subscript operator	[ ]
Function-call operator	( )
Explicit type conversion operator	*type-name*( )
Member-selection operator	. or –>
Postfix increment operator	++
Postfix decrement operator	– –

Syntax

postfix-expression:
primary-expression
postfix-expression [*expression*]
postfix-expression (*expression-list*$_{opt}$)
simple-type-name (*expression-list*$_{opt}$)
postfix-expression . *name*
postfix-expression –> *name*
postfix-expression ++
postfix-expression – –

expression-list:
assignment-expression
expression-list , *assignment-expression*

Subscript Operator

A *postfix-expression* followed by the subscript operator, [ ], specifies array indexing. One of the expressions must be of pointer or array type—that is, it must have been declared as *type** or type[ ]. The other expression must be of an integral type (including enumerated types). In common usage, the expression enclosed in the brackets is the one of integral type, but that is not strictly required. Consider the following example:

```
MyType m[10];     // Declare an array of a user-defined type.

MyType n1 = m[2]; // Select third element of array.
MyType n2 = 2[m]; // Select third element of array.
```

In the example above, the expression `m[2]` is identical to `2[m]`. Although `m` is not of an integral type, the effect is the same. The reason that `m[2]` is equivalent to `2[m]` is that the result of a subscript expression *e1*[*e2*] is given by:

*((*e2*) + (*e1*))

The address yielded by the above expression is not *e2* bytes from the address *e1*. Rather, the address is scaled to yield the next object in the array *e2*. For example:

```
#include <iostream.h>

int main()
{
    double aDbl[2];

    cout << "Address of first element is: "
         << &aDbl[0] << "\n";
    cout << "Address of second element is: "
         << &aDbl[1] << "\n";

    return 0;
}
```

The preceding program prints two addresses that are 8 bytes apart—the size of an object of type **double**. This scaling according to object type is done automatically by the C++ language, and is defined in "Additive Operations" on page 104 where addition and subtraction of operands of pointer type is discussed.

Positive and Negative Subscripts The first element of an array is element 0. Therefore, the range of a C++ array is from *array*[0] to *array*[*size* – 1]. However, since C++ supports both positive and negative subscripts, it can be convenient to use them in arrays. Although C++ permits negative subscripts, they must fall within the array boundaries or the results are unpredictable. The following code illustrates this concept:

```
#include <iostream.h>

main()
{
    int iNumberArray[1024];
    int *iNumberLine = &iNumberLine[512];

    cout << iNumberArray[-256] << "\n"; // Run-time error
    cout << iNumberLine[-256] << "\n";  // OK

    return 0;
}
```

The negative subscript in `iNumberArray` can produce a run-time error because it yields an address 256 bytes lower in memory than the actual origin of the array. The object `iNumberLine` is initialized to the middle of `iNumberArray`; it is therefore possible to use both positive and negative array indices on it. Array subscript errors do not generate compile-time errors, but they yield unpredictable results.

The subscript operator is commutative. Therefore, the expressions *array*[*index*] and *index*[*array*] are guaranteed to be equivalent as long as the subscript operator is not overloaded (see "Overloaded Operators" in Chapter 12, on page 351). The first form is the most common coding practice, but either works.

Function-Call Operator

A *postfix-expression* followed by the function-call operator, ( ), specifies a function call. The arguments to the function-call operator are zero or more expressions separated by commas—the actual arguments to the function.

The *postfix-expression* must be of one of these types:

- Function returning type `T`. An example declaration is
  ```
  T func( int i )
  ```
- Pointer to a function returning type `T`. An example declaration is
  ```
  T (*func)( int i )
  ```
- Reference to a function returning type `T`. An example declaration is
  ```
  T (&func)(int i)
  ```
- Pointer-to-member function dereference returning type `T`. Example function calls are:
  ```
  (pObject->*pmf)();
  (Object.*pmf)();
  ```

Formal and Actual Arguments Calling programs pass information to called functions in "actual arguments." The called functions access the information using corresponding "formal arguments."

When a function is called, the following tasks are performed:

- All actual arguments (those supplied by the caller) are evaluated. There is no implied order in which these arguments are evaluated, but all arguments are evaluated and all side effects completed prior to entry to the function.
- Each formal argument is initialized with its corresponding actual argument in the expression list. (A formal argument is an argument that is declared in the function header and used in the body of a function.) Conversions are done as if by initialization—both standard and user-defined conversions are performed in converting an actual argument to the correct type. The initialization performed is illustrated conceptually by the following code:

```
void Func( int i ); // Function prototype
...
Func( 7 );          // Execute function call
```

The conceptual initializations prior to the call are shown below:

```
int Temp_i = 7;
Func( Temp_i );
```

Note that the initialization is performed as if using the equal-sign syntax instead of the parentheses syntax. A copy of `i` is made prior to passing the value to the function. (For more information, see "Initializers" in Chapter 7, on page 217 and "Conversions," "Initialization Using Special Member Functions," and "Explicit Initialization" in Chapter 11, on pages 312, 325, and 326, respectively.

Therefore, if the function prototype (declaration) calls for an argument of type **long**, and the calling program supplies an actual argument of type **int**, the actual argument is promoted using a standard type conversion to type **long** (see Chapter 3, "Standard Conversions").

It is an error to supply an actual argument for which there is no standard or user-defined conversion to the type of the formal argument.

For actual arguments of class type, the formal argument is initialized by calling the class's constructor. (See "Constructors" in Chapter 11, on page 300 for more about these special class member functions.)

- The function call is executed.

The following program fragment demonstrates a function call:

```
void func( long param1, double param2 );

int main()
{
    int i, j;

    // Call func with actual arguments i and j.
    func( i, j );
    ...
}

// Define func with formal parameters param1
// and param2.
void func( long param1, double param2 )
{
    ...
}
```

When `func` is called from **main**, the formal parameter `param1` is initialized with the value of `i` (`i` is converted to type **long** to correspond to the correct type using a standard conversion), and the formal parameter `param2` is initialized with the value of `j` (`j` is converted to type **double** using a standard conversion).

Treatment of Argument Types Formal arguments declared as **const** types cannot be changed within the body of a function. Functions can change any argument that is not of type **const**. However, the change is local to the function and does not affect the actual argument's value unless the actual argument was a reference to an object not of type **const**.

The following functions illustrate some of these concepts:

```
int func1( int i, int j, char *c )
{
    i = 7;          // Error: i is const.
    j = i;          // OK, but value of j is
                    //  lost at return.
    *c = 'a' + j;   // OK: changes value of c
                    //  in calling function.

    return i;
}

double& func2( double& d, const char *c )
{
    d = 14.387;     // OK: changes value of d
                    //  in calling function.
    *c = 'a';       // Error: c is a pointer to
                    //  a const object.

    return d;
}
```

Ellipses and Default Arguments Functions can be declared to accept fewer arguments than specified in the function definition, using one of two methods: ellipsis (...) or default arguments.

Ellipses denote that arguments may be required but that the number and types are not specified in the declaration. This is normally poor C++ programming practice because it defeats one of the benefits of C++: type safety. Different conversions are applied to functions declared with ellipses than to those functions for which the formal and actual argument types are known:

- If the actual argument is of type **float**, it is promoted to type **double** prior to the function call.
- Any signed or unsigned **char**, **short**, enumerated type, or bit field is converted to either a signed or unsigned **int** using integral promotion.
- Any argument of class type is passed by value as a data structure; the copy is created by binary copying instead of by invoking the class's copy constructor (if one exists).

Ellipses, if used, must be declared last in the argument list. For more information about the use of ellipses to pass a variable number of arguments, see the *Run-Time Library Reference* manual, under the topics: **va_arg**, **va_list**, and **va_start**.

Default arguments allow the programmer to specify the value an argument should assume if none is supplied in the function call. The following code fragment shows how default arguments work (for more information about default arguments, see "Default Arguments" in Chapter 7, on page 210):

```
#include <iostream.h>

// Declare the function print that prints a string,
// then a terminator.
void print( const char *string,
            const char *terminator = "\n" );

int main()
{
    print( "hello," );
    print( "world!" );

    print( "good morning", " ," );
    print( "sunshine." );

    return 0;
}

// Define print.
void print( char *string, char *terminator )
{
    if( string != NULL )
        cout << string;

    if( terminator != NULL )
        cout << terminator;
}
```

The above program declares a function, `print`, that takes two arguments. However, the second argument, `terminator`, has a default value, `"\n"`. In **main**, the

first two calls to `print` allow the default second argument to supply a new line to terminate the printed string. The third call specifies an explicit value for the second argument. The output from the program is:

```
hello,
world
good morning, sunshine.
```

Function Call Results A function call evaluates to an r-value unless the function is declared as a reference type. Functions with reference return type evaluate to l-values, and it is legal to use them on the left side of an assignment statement as follows:

```
#include <iostream.h>

class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
private:
    unsigned _x;
    unsigned _y;
};

int main()
{
    Point ThePoint;

    ThePoint.x() = 7;           // Use x() as an l-value.
    unsigned y = ThePoint.y();  // Use y() as an r-value.

    // Use x() and y() as rvalues.
    cout << "x = " << ThePoint.x() << "\n"
         << "y = " << ThePoint.y() << "\n";

    return 0;
}
```

The above code defines a class called `Point`, which contains private data objects that represent *x* and *y* coordinates. These data objects must be modified and their values retrieved. The above program is only one of several designs for such a class; use of the `GetX` and `SetX` or `GetY` and `SetY` functions is another possible design.

A function returning a pointer to an object can appear on the left side of an assignment statement as follows:

```
struct A
{
    int i;
};

A *func();

...

func()->i = 7;
```

The above statement is legal because `func` returns a pointer to an object of type `A`. Therefore, the member-selection operator, `->`, dereferences the pointer, making it an l-value. See "L-Values and R-Values" in Chapter 2, on page 60 for more about expressions that are l-values.

Functions that return class types, pointers to class types, or references to class types can be used as the left operand to member-selection operators. Therefore, the following code is legal:

```
class A
{
public:
    int SetA( int i ) { return (I = i); }
    int GetA()    { return I; }
private:
    int I;
};

// Declare three functions:
//  func1, which returns type A
//  func2, which returns a pointer to type A
//  func3, which returns a reference to type A
A func1();
A* func2();
A& func3();

int iResult = func1().GetA();
func2()->SetA( 3 );
func3().SetA( 7 );
```

Functions can be called recursively. For more information about function declarations, see "Function Specifiers" in Chapter 6, on page 159 and "Member Functions" in Chapter 8, on page 240. Related material is in "Program and Linkage" in Chapter 2, on page 33.

Member-Selection Operator

A *postfix-expression* followed by the member-selection operator (.) and a *name* is also a *postfix-expression.* The first operand of the member-selection operator must be a class object (an object declared as **class**, **struct**, or **union** type) or a reference to a class object, and the second operand must identify a member of that class.

The result of the expression is the value of the member, and it is an l-value if the named member is an l-value.

A *postfix-expression* followed by the member-selection operator (–>) and a *name* is a *postfix expression.* The first operand of the member-selection operator must be a pointer to a class object (an object declared as **class**, **struct**, or **union** type), and the second operand must identify a member of that class.

The result of the expression is the value of the member, and it is an l-value if the named member is an l-value. The –> operator dereferences the pointer. Therefore, the expressions *e–>member* and (**e*).*member* (where *e* represents an expression) yield identical results (except when the operators –> or * are overloaded).

When a value is stored through one member of a union but retrieved through another member, no conversion is performed. The following program stores data into the object `U` as **int**, but retrieves the data as two separate bytes of type **char**:

```
#include <iostream.h>

int main()
{
    struct ch
    {
      char b1;
      char b2;
    };
    union u
    {
        struct ch uch;
        int    i;
    };

    u U;

    U.i = 0x6361;  // Bit pattern for "ac"
    cout << U.uch.b1 << U.uch.b2 << "\n";

    return 0;
}
```

Note The preceding code is not portable because it assumes an **int** is two bytes long while a **char** is one byte long. In 32-bit target compilations, this assumption is false.

Postfix Increment and Decrement Operators

C++ provides prefix and postfix increment and decrement operators; this section describes only the postfix increment and decrement operators. (For more information, see "Increment and Decrement Operators" on page 94.) The difference between the two is that in the postfix notation, the operator appears after *postfix-expression*, whereas in the prefix notation, the operator appears before *expression*. The following example shows a postfix-increment operator:

```
i++
```

The effect of applying the postfix increment, or "postincrement," operator (**++**) is that the operand's value is increased by one unit of the appropriate type. Similarly, the effect of applying the postfix decrement or "postdecrement" operator (**– –**) is that the operand's value is decreased by one unit of the appropriate type.

For example, applying the postincrement operator to a pointer to an array of objects of type **long** actually adds four to the internal representation of the pointer. This behavior causes the pointer, which previously referred to the *n*th element of the array, to refer to the (*n*+1)th element.

The operands to postincrement and postdecrement operators must be modifiable (not **const**) l-values of arithmetic or pointer type. The result of the postincrement or postdecrement expression is the value of the *postfix-expression* prior to application of the increment operator. The type of the result is the same as that of the *postfix-expression*, but it is no longer an l-value.

Postincrement and postdecrement, when used on enumerated types, yield integral values. Therefore, the following code is illegal:

```
enum Days {
    Sunday = 1,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};

int main()
{
  Days Today = Tuesday;
  Days SaveToday;

  SaveToday = Today++;

  return 0;
}
```

The intent of the above code is to save today's day, then move to tomorrow. However, the result is that the expression `Today++` yields an **int**—an error when assigned to an object of the enumerated type `Days`.

Expressions with Unary Operators

Unary operators are those operators that act on only one operand in an expression. The unary operators are:

- Indirection operator (*)
- Address-of operator (&)
- Unary plus operator (+)
- Unary negation operator (–)
- Logical NOT operator (!)
- One's complement operator (~)
- Preincrement operator (++)
- Predecrement operator (– –)
- **sizeof** operator
- **new** operator
- **delete** operator

These operators have right-to-left associativity.

Syntax

unary-expression:
postfix-expression
++*unary-expression*
–– *unary-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof (*type-name*)
allocation-expression
deallocation-expression

unary-operator: one of
* & + – ! ~

Indirection Operator (*)

The unary indirection operator (*) "dereferences" a pointer; that is, it converts a pointer value to an l-value. The operand of the indirection operator must be a pointer to a type. The result of the indirection expression is the type from which the pointer type is derived. The use of the * operator in this context is different from its meaning as a binary operator, which is multiplication.

Address-Of Operator (&)

The unary address-of operator (**&**) takes the address of its operand. The address-of operator can be applied only to the following:

- Functions (although its use for taking the address of a function is unnecessary)
- L-values
- *qualified-names*

In the first two cases listed above, the result of the expression is a pointer type (an r-value) derived from the type of the operand. For example, if the operand is of type **char**, the result of the expression is of type pointer to **char**. The address-of operator, applied to **const** or **volatile** objects, evaluates to **const** *type* * or **volatile** *type* *, where *type* is the type of the original object.

The result produced by the third case, applying the address-of operator to a *qualified-name*, depends on whether the *qualified-name* specifies a static member. If so, the result is a pointer to the type specified in the declaration of the member. If the member is not static, the result is a pointer to the member *name* of the class indicated by *qualified-class-name*. (See "Primary Expressions" on page 78 for more about *qualified-class-name*.) The following code fragment shows how the result differs, depending on whether the member is static:

```
class PTM
{
public:
           int   iValue;
    static float fValue;
};

int   PTM::*piValue = &PTM::iValue;  // OK: non-static
float PTM::*pfValue = &PTM::fValue;  // Error: static
float *spfValue     = &PTM::fValue;  // OK
```

In the above example, the expression `&PTM::fValue` yields type `float *` instead of type `float PTM::*` because `fValue` is a static member.

The address of an overloaded function can be taken only when it is clear which version of the function is being referred to. See "Address Of Overloaded Functions" in Chapter 12, on page 351 for information about how to obtain the address of a particular overloaded function.

Applying the address-of operator to a reference type gives the same result as applying the operator to the object to which the reference is bound. The following program demonstrates this concept:

```
#include <iostream.h>

int main()
{
    double d;         // Define an object of type double.
    double& rd = d;   // Define a reference to the object.

    // Compare the address of the object to the address
    //  of the reference to the object.
    if( &d == &rd )
        cout << "&d equals &rd" << "\n";
    else
        cout << "&d is not equal to &rd" << "\n";

    return 0;
}
```

The output from the program is always `&d equals &rd`.

Unary Plus Operator (+)

The result of the unary plus operator (+) is the value of its operand. The operand to the unary plus operator must be of an arithmetic type.

Integral promotion is performed on integral operands. The resultant type is the type to which the operand is promoted. Thus, the expression `+ch`, where `ch` is of type **char**, results in type **int**; the value is unmodified.

See "Integral Promotions" in Chapter 3, on page 66 for more information about how the promotion is done.

Unary Negation Operator (–)

The unary negation operator (–) produces the negative of its operand. The operand to the unary negation operator must be of an arithmetic type.

Integral promotion is performed on integral operands, and the resultant type is the type to which the operand is promoted. See "Integral Promotions" in Chapter 3, on page 66 for more information on how the promotion is done.

Microsoft Specific

Unary negation of unsigned quantities is performed by subtracting the value of the operand from 2^n, where n is the number of bits in an object of the given unsigned type. (Microsoft C++ runs on processors that utilize two's-complement arithmetic. On other processors, the algorithm for negation can differ.) ♦

Logical NOT Operator (!)

The result of the logical NOT operator (**!**) is 0 if its operand evaluates to a nonzero value; the result is 1 only if the operand is equal to 0. The operand must be of arithmetic or pointer type. The result is of type **int**.

For an expression *e*, the unary expression *!e* is equivalent to the expression (*e* == **0**), except where overloaded operators are involved.

One's Complement Operator (~)

The one's complement operator (~), sometimes called the "bitwise complement" operator, yields a bitwise one's complement of its operand. That is, every bit that is set in the operand is 0 in the result. Conversely, every bit that is 0 in the operand is set in the result. The operand to the one's complement operator must be an integral type.

Integral promotion is performed on integral operands, and the resultant type is the type to which the operand is promoted. See "Integral Promotions" in Chapter 3, on page 66 for more information on how the promotion is done.

Increment and Decrement Operators (++, --)

The prefix increment operator (++), also called the "preincrement" operator, adds one to its operand; this incremented value is the result of the expression. The operand must be an l-value not of type **const**. The result is an l-value of the same type as the operand.

The prefix decrement operator (--), also called the "predecrement" operator, is analogous to the preincrement operator, except that the operand is decremented by one and the result is this decremented value.

Both the prefix and postfix increment and decrement operators affect their operands. The key difference between them is when the increment or decrement takes place in the evaluation of an expression. (For more information, see "Postfix Increment and Decrement Operators" on page 90). In the prefix form, the increment or

decrement takes place before the value is used in expression evaluation, so the value of the expression is different from the value of the operand. In the postfix form, the increment or decrement takes place after the value is used in expression evaluation, so the value of the expression is the same as the value of the operand.

Because increment and decrement operators have side effects, using expressions with increment or decrement operators in a macro can have undesirable results (see "The Role of Preprocessing in C++" in Chapter 13, on page 367 for more information about macros). Consider this example:

```
#define max(a,b) ((a)<(b))?(b):(a)

...

int i, j, k;

k = max( ++i, j );
```

In the code fragment above, the macro expands to:

```
k = ((++i)<(j))?(j):(++i);
```

If `i` is greater than or equal to `j`, it will be incremented twice.

Note C++ inline functions are preferable to macros in many cases because they eliminate side effects such as those described above, and they allow the language to perform more complete type checking.

sizeof Operator

The **sizeof** operator yields the size of its operand with respect to the size of type **char** (the size in **char**s). The result of the **sizeof** operator is of type **size_t**, an integral type defined in the include file STDDEF.H. The operand to **sizeof** can be one of the following:

- A type name. To use **sizeof** with a type name, the name must be enclosed in parentheses.
- An expression. When used with an expression, **sizeof** can be specified with or without the parentheses. The expression is not evaluated.

When the **sizeof** operator is applied to an object of type **char**, it yields 1. When the **sizeof** operator is applied to an array, it yields the total number of bytes in that array. For example:

```
#include <iostream.h>

int main()
{
    char szHello[] = "Hello, world!";

    cout << "The size of the type of " << szHello << " is: "
         << sizeof( char ) << "\n";
    cout << "The length of " << szHello << " is: "
         << sizeof szHello << "\n";

    return 0;
}
```

The program output is:

```
The size of the type of Hello, world! is: 1
The length of Hello, world! is: 14
```

When the **sizeof** operator is applied to a **class**, **struct**, or **union** type, the result is the number of bytes in an object of that **class**, **struct**, or **union** type, plus any padding added to align members on word boundaries. (The /Zp compiler option and the **pack** pragma affect alignment boundaries for members.) The **sizeof** operator never yields 0, even for an empty class.

The **sizeof** operator cannot be used with the following operands:

- Functions. (However, **sizeof** can be applied to pointers to functions.)
- Bit fields.
- Undefined classes.
- The type **void**.
- Incomplete types.
- Parenthesized names of incomplete types.

When the **sizeof** operator is applied to a reference, the result is the same as if **sizeof** had been applied to the object itself.

The **sizeof** operator is often used to calculate the number of elements in an array using an expression of the form:

```
sizeof array / sizeof array[0]
```

new Operator

The **new** operator attempts to dynamically allocate (at run time) one or more objects of *type-name*. The **new** operator cannot be used to allocate a function; however, it can be used to allocate a pointer to a function.

Syntax

allocation-expression:
::$_{opt}$ **new** *model*$_{opt}$ *placement*$_{opt}$ *new-type-name* *new-initializer*$_{opt}$
::$_{opt}$ **new** *model*$_{opt}$ *placement*$_{opt}$ (*type-name*) *new-initializer*$_{opt}$

placement:
(*expression-list*)

new-type-name:
type-specifier-list new-declarator$_{opt}$

The **new** operator is used to allocate objects and arrays of objects. The **new** operator allocates from an area of program memory called the "free store." In C, the free store is often referred to as the "heap."

When **new** is used to allocate a single object, it yields a pointer to that object; the resultant type is *new-type-name* * or *type-name* *. When **new** is used to allocate a singly dimensioned array of objects, it yields a pointer to the first element of the array, and the resultant type is *new-type-name* * or *type-name* *. When **new** is used to allocate a multiply dimensioned array of objects, it yields a pointer to the first element of the array, and the resultant type preserves the size of all but the left-most array dimension. For example:

```
new float[10][25][10]
```

yields type `float (*)[25][10]`. Therefore, the following code will not work because it attempts to assign a pointer to an array of `float` with the dimensions `[25][10]` to a pointer to type `float`:

```
float *fp;
fp = new float[10][25][10];
```

The correct expression is:

```
float (*cp)[25][10];
cp = new float[10][25][10];
```

The definition of `cp` allocates a pointer to an array of type `float` with dimensions `[25][10]`—it does not allocate an array of pointers.

All array dimensions but the leftmost must be constant expressions that evaluate to positive values; the leftmost array dimension may be any expression that evaluates to a positive value. When allocating an array using the **new** operator, the first dimension can be zero—the **new** operator returns a unique pointer.

The *type-specifier-list* may not contain **const**, **volatile**, class declarations, or enumeration declarations. Therefore, the following expression is illegal:

```
volatile char *vch = new volatile char[20];
```

The **new** operator does not allocate reference types because they are not objects.

Lifetime of Objects Allocated with new Objects allocated with the **new** operator are not destroyed when the scope in which they are defined is exited. Because the **new** operator returns a pointer to the objects it allocates, the program must define a pointer with suitable scope to access those objects. For example:

```
int main()
{
    // Use new operator to allocate an array of 20 characters.
    char *AnArray = new char[20];

    for( int i = 0; i < 20; ++i )
    {
        // On the first iteration of the loop, allocate
        //  another array of 20 characters.
        if( i == 0 )
        {
            char *AnotherArray = new char[20];
        }
        ...
    }

    delete AnotherArray; // Error: pointer out of scope.
    delete AnArray;      // OK: pointer still in scope.
}
```

By letting the pointer `AnotherArray` go out of scope in the above example, the programmer has allocated an object that can no longer be deleted.

Initializing Objects Allocated with new An optional *new-initializer* field is included in the syntax for the **new** operator. This allows new objects to be initialized with user-defined constructors. For more information about how initialization is done, see "Initializers" in Chapter 7, on page 217.

The following example illustrates how to use an initialization expression with the **new** operator:

```
#include <iostream.h>

class Acct
{
public:
    // Define default constructor and a constructor that accepts
    //  an initial balance.
    Acct() { balance = 0.0; }
    Acct( double init_balance ) { balance = init_balance; }
private:
    double balance;
};

int main()
{
    Acct *CheckingAcct = new Acct;
    Acct *SavingsAcct = new Acct ( 34.98 );
    double *HowMuch = new double ( 43.0 );

    ...

    return 0;
}
```

In the example above, the object `CheckingAcct` is allocated using the **new** operator, but no default initialization is specified. Therefore, the default constructor for the class, `Acct()`, is called. Then, the object `SavingsAcct` is allocated the same way, except that it is explicitly initialized to 34.98. Because 34.98 is of type **double**, the constructor that takes an argument of that type is called to handle the initialization. Finally, the nonclass type `HowMuch` is initialized to 43.0.

If an object is of a class type, and that class has constructors (as in the above example), the object can be initialized by the **new** operator only if one of these conditions is met:

- The arguments provided in the initializer agree with those of a constructor
- The class has a default constructor (a constructor that can be called with no arguments)

Access control and ambiguity control are performed on **operator new** and on the constructors according to the rules set forth in "Ambiguity" in Chapter 9, on page 282 and "Initialization Using Special Member Functions" in Chapter 11, on page 325.

No explicit per-element initialization can be done when allocating arrays using the **new** operator; only the default constructor, if present, is called. (Note that a default constructor is a constructor that takes no arguments. Constructors declared with all default arguments are default constructors.)

If the memory allocation fails (**operator new** returns a value of 0), no initialization is performed. This protects against attempts to initialize data that does not exist.

As with function calls, the order of evaluation of initialization expressions is not defined. Furthermore, it is unsafe to rely on these expressions being completely evaluated before the memory allocation is performed. If the memory allocation fails and the **new** operator returns zero, it is possible that not all expressions in the initializer are completely evaluated.

How new Works The *allocation-expression*—the expression containing the **new** operator—does three things:

- Locates and reserves storage for the object or objects to be allocated. When this stage is complete, the correct amount of storage is allocated, but it is not yet an object.
- Initializes the object(s). Once initialization is complete, enough information is present for the allocated storage to be an object.
- Returns a pointer to the object(s) of a pointer type derived from *new-type-name* or *type-name*. This pointer is used by the program to access the newly allocated object.

The **new** operator actually invokes the function **operator new**. For arrays of any type, and for objects that are not of **class**, **struct**, or **union** types, a global function, **::operator new**, is called to allocate storage. Class-type objects can define their own **operator new** on a per-class basis.

When the compiler encounters the **new** operator to allocate an object of type *type*, it issues a call to *type*::**operator new(** **sizeof(** *type*)), or if no user-defined **operator new** is defined, **::operator new(** **sizeof(** *type*)). Therefore, the **new** operator can allocate the correct amount of memory for the object.

Note The argument to operator **new** is of type **size_t**. This type is defined in DIRECT.H, MALLOC.H, MEMORY.H, SEARCH.H, STDDEF.H, STDIO.H, STDLIB.H, STRING.H, and TIME.H.

An option in the syntax allows specification of *placement* (see the **new** operator syntax on page 97). The *placement* field can be used only for user-defined implementations of **operator new**; it allows extra information to be passed to **operator new**. An expression with a *placement* field such as:

```
T *TObject = new ( 0x0040 ) T;
```

is translated to

```
T *TObject = T::operator new( sizeof( T ), 0x0040 );
```

The original intention of the *placement* field was to allow hardware-dependent objects to be allocated at user-specified addresses.

Note Although the example above shows only one argument in the *placement* field, there is no restriction on how many extra arguments can be passed to **operator new** this way.

Even when **operator new** has been defined for a class type, the global operator can be used by using the form of this example:

```
T *TObject =::new TObject;
```

The scope-resolution operator (**::**) forces use of the global **new** operator.

delete Operator

The **delete** operator deallocates an object created with the **new** operator. The **delete** operator has a result of type **void** and therefore does not return a value. The operand to **delete** must be a pointer returned by the **new** operator.

Using **delete** on a pointer to an object not allocated with **new** gives unpredictable results. You can, however, use **delete** on a pointer with the value 0. This provision means that, because **new** always returns 0 on failure, deleting the result of a failed **new** operation is harmless.

Syntax

deallocation-expression:
  ::$_{opt}$ **delete** *cast-expression*
  ::$_{opt}$ **delete [ ]** *cast-expression*

Using the **delete** operator on an object deallocates its memory. A program that dereferences a pointer after the object is deleted can have unpredictable results or crash.

If the operand to the **delete** operator is a modifiable l-value, its value is undefined after the object is deleted.

Pointers to **const** objects cannot be deallocated with the **delete** operator.

How delete Works The **delete** operator actually invokes the function **operator delete**. For objects of class types (**class**, **struct**, and **union**), the **delete** operator invokes the destructor for an object prior to deallocating memory (if the pointer is not null). For objects not of class type, the global **delete** operator is invoked. For objects of class type, the **delete** operator can be defined on a per-class basis; if there is no such definition for a given class, then the global operator is invoked.

Microsoft Specific Microsoft C++ allows multiple **delete** operators to be present for a given class type—one for each addressing option. This guarantees that the correct object is deleted when the **delete** operator is invoked in the program. See "Declaring Destructors" in Chapter 11, on page 306, "Overloaded Operators" in Chapter 12, on page 351, and "Memory-Model Specifiers and Overloading" in Appendix B, on page 401.♦

Using delete There are two syntactic variants for the **delete** operator: one for single objects and the other for arrays of objects. The following code fragment shows how these differ:

```
int main()
{
    // Allocate a user-defined object, UDObject, and an object
    //  of type double on the free store using the
    //  new operator.
    UDType *UDObject = new UDType;
    double *dObject = new double;
    ...
    // Delete the two objects.
    delete UDObject;
    delete dObject;
    ...
    // Allocate an array of user-defined objects on the
    // free store using the new operator.
    UDType (*UDArr)[7] = new UDType[5][7];
    ...
    // Use the array syntax to delete the array of objects.
    delete [] UDArr;

    return 0;
}
```

These two cases produce undefined results: using the array form of **delete** (**delete [ ]**) on an object, and using the nonarray form of **delete** on an array.

Expressions with Binary Operators

Binary operators act on two operands in an expression. The binary operators are:

- Multiplication (*)
- Division (/)
- Modulus (%)
- Addition (+)
- Subtraction (–)
- Right shift (>>)

- Left shift (<<)
- Less than (<)
- Greater than (>)
- Less than or equal to (<=)
- Greater than or equal to (>=)
- Equal to (==)
- Not equal to (!=)
- Bitwise AND (&)
- Bitwise exclusive OR (^)
- Bitwise inclusive OR (|)
- Logical AND (&&)
- Logical OR (||)

Multiplicative Operators

The multiplicative operators are:

- Multiplication (*)
- Division (/)
- Modulus or "remainder from division" (%)

These binary operators have left-to-right associativity.

Syntax

multiplicative-expression:
pm-expression
multiplicative-expression * *pm-expression*
multiplicative-expression / *pm-expression*
multiplicative-expression % *pm-expression*

The multiplicative operators take operands of arithmetic types. The modulus operator (%) has a stricter requirement in that its operands must be of integral type. (To get the remainder of a floating-point division, use the run-time function, **fmod.**) The conversions covered in "Arithmetic Conversions" in Chapter 3, on page 69 are applied to the operands, and the result is of the converted type.

The multiplication operator yields the result of multiplying the first operand by the second.

The division operator yields the result of dividing the first operand by the second.

The modulus operator yields the remainder given by the following expression, where *e1* is the first operand and *e2* is the second: *e1* – (*e1* / *e2*) * *e2*, where both operands are of integral types.

Division by 0 in either a division or modulus expression is undefined and causes a run-time error. Therefore the following expressions generate undefined, erroneous results:

```
i % 0
f / 0.0
```

If both operands to a multiplication, division, or modulus expression have the same sign, the result is positive. Otherwise, the result is negative. The result of a modulus operation's sign is implementation-defined.

Microsoft Specific

In Microsoft C++, the result of a modulus expression is always the same as the sign of the first operand. ♦

If the computed division of two integers is inexact and only one of the operands is negative, the result is the largest integer (in magnitude, disregarding the sign) that is less than the exact value the division operation would yield. For example, the exact value of –11 / 3 is –3.666666666 (not a rational number). The result of that integral division is –3.

The relationship between the multiplicative operators is given by the identity (*e1* / *e2*) * *e2* + *e1* % *e2* == *e1*.

Additive Operators

The additive operators are:

- Addition (+)
- Subtraction (–)

These binary operators have left-to-right associativity.

Syntax

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression – *multiplicative-expression*

The additive operators take operands of arithmetic or pointer types. The result of the addition (+) operator is the sum of the operands. The result of the subtraction (–) operator is the difference between the operands. If one or both of the operands are pointers, they must be pointers to objects, not to functions.

Additive operators take operands of *arithmetic*, *integral*, and *scalar* types. These are defined in Table 4.2:

Table 4.2 Types Used with Additive Operators

Type	Meaning
arithmetic	Integral and floating types are collectively called "arithmetic" types.
integral	Types **char** and **int** of all sizes (**long**, **short**) and enumerations are "integral" types.
scalar	Scalar operands are operands of either arithmetic or pointer type.

The legal combinations for these operators are:

arithmetic + *arithmetic*
scalar + *integral*
integral + *scalar*
arithmetic – *arithmetic*
scalar – *scalar*

Note that addition and subtraction are not equivalent operations.

If both operands are of arithmetic type, then the conversions covered in "Arithmetic Conversions" in Chapter 3, on page 69 are applied to the operands, and the result is of the converted type.

Addition of Pointer Types If one of the operands in an addition operation is a pointer to an array of objects, the other must be of integral type. The result is a pointer that is of the same type as the original pointer and that points to another array element. The following code fragment illustrates this concept:

```
short IntArray[10]; // Objects of type short occupy 2 bytes
short *pIntArray = IntArray;

for( int i = 0; i < 10; ++i )
{
    *pIntArray = i;
    cout << *pIntArray << "n";
    pIntArray = pIntArray + 1;
}
```

Although the integral value 1 is added to `pIntArray`, it does not mean "add 1 to the address"; rather it means "adjust the pointer to point to the next object in the array" (which happens to be 2 bytes away).

Note Code of the form `pIntArray = pIntArray + 1` is rarely found in C++ programs; to perform an increment, these forms are preferable: `pIntArray++` or `pIntArray += 1`.

Subtraction of Pointer Types If both operands are pointers, then the result of subtraction is the difference (in array elements) between the operands. The subtraction expression yields a signed integral result of type **ptrdiff_t** (defined in the standard include file STDDEF.H).

One of the operands can be of integral type, as long as it is the second operand. The result of the subtraction is of the same type as the original pointer. The value of the subtraction is a pointer to the $(n - i)$th array element, where n is the element pointed to by the original pointer and i is the integral value of the second operand.

Shift Operators

The bitwise shift operators are:

- Right shift (>>)
- Left shift (<<)

These binary operators have left-to-right associativity.

Syntax

shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*

Both operands of the shift operators must be of integral types. Integral promotions are performed according to the rules described in "Integral Promotions" in Chapter 3, on page 66. The type of the result is the same as the type of the left operand. The value of a right shift expression *e1* >> *e2* is $e1 / 2^{e2}$, and the value of a left shift expression *e1* << *e2* is $e1 * 2^{e2}$.

The results are undefined if the right operand of a shift expression is negative, or if the right operand is greater than or equal to the number of bits in the (promoted) left operand.

The left-shift operator causes the bit pattern in the first operand to be shifted left the number of bits specified by the second operand. Bits vacated by the shift operation are zero-filled. The shift is a logical shift as opposed to a shift-and-rotate operation.

The right-shift operator causes the bit pattern in the first operand to be shifted right the number of bits specified by the second operand. Bits vacated by the shift operation are zero-filled for unsigned quantities. For signed quantities, the sign bit is propagated into the vacated bit positions. The shift is a logical shift if the left operand is an unsigned quantity; otherwise, it is an arithmetic shift.

Microsoft Specific

The result of a right shift of a signed negative quantity is implementation dependent. Although Microsoft C++ propagates the most-significant bit to fill vacated bit positions, there is no guarantee other implementations will do likewise. ♦

Relational and Equality Operators

The relational and equality operators determine equality, inequality, or relative values of their operands. The relational operators are shown in Table 4.3.

Table 4.3 Relational and Equality Operators

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Relational Operators The binary relational operators determine the following relationships:

- Less than
- Greater than
- Less than or equal to
- Greater than or equal to

Syntax

relational-expression:
shift-expression
relational-expression **<** *shift-expression*
relational-expression **>** *shift-expression*
relational-expression **<=** *shift-expression*
relational-expression **>=** *shift-expression*

The relational operators have left-to-right associativity. Both operands of relational operators must be of arithmetic or pointer type. They yield values of type **int**. The value returned is 0 if the relationship in the expression is false; otherwise, it is 1. Consider the following code that demonstrates several relational expressions:

```
#include <iostream.h>

int main()
{
    cout << "The true expression 3 > 2 yields: "
         << (3 > 2) << "\n";
    cout << "The false expression 20 < 10 yields: "
         << (20 < 10) << "\n";
    cout << "The expression 10 < 20 < 5 yields: "
         << (10 < 20 < 5) << "\n";

    return 0;
}
```

The output from this program is:

```
The true expression 3 < 2 yields 1
The false expression 20 < 10 yields 0
The expression 10 < 20 < 5 yields 1
```

The expressions in the example above must be parenthesized because the insertion operator (**<<**) has higher precedence than the relational operators. Therefore, the first expression without the parentheses would be evaluated as follows:

```
(cout << "The true expression 3 > 2 yields: " << 3) < (2 << "\n");
```

Note that the third expression evaluates to 1—because of the left-to-right associativity of relational operators, the explicit grouping of the expression `10 < 20 < 5` is:

```
(10 < 20) < 5
```

Therefore, the test performed is:

```
1 < 5
```

and the result is 1 (or true).

The usual arithmetic conversions covered in "Arithmetic Conversions" in Chapter 3, on page 69 are applied to operands of arithmetic types.

Comparing Pointers Using Relational Operators When two pointers to objects of the same type are compared, the result is determined by the location of the objects pointed to in the program's address space. Pointers can also be compared to a constant expression that evaluates to 0 or to a pointer of type **void** *. If a pointer comparison is made against a pointer of type **void** *, the other pointer is implicitly converted to type **void** *. Then the comparison is made.

Two pointers of different types cannot be compared unless:

- One type is a class type derived from the other type.
- At least one of the pointers is explicitly converted (cast) to type **void** *. (The other pointer is implicitly converted to type **void** * for the conversion.)

Two pointers of the same type that point to the same object are guaranteed to compare equal. If two pointers to nonstatic members of an object are compared, the following rules apply:

- If the class type is not a union, and if the two members are not separated by an *access-specifier*, such as **public**, **protected**, or **private**, the pointer to the member declared last will compare greater than the pointer to the member declared earlier. (For information on *access-specifier*, see the "Syntax" section on page 286 in Chapter 10.)
- If the two members are separated by an *access-specifier*, the results are undefined.
- If the class type is a union, pointers to different data members in that union compare equal.

If two pointers point to elements of the same array or to the element one beyond the end of the array, the pointer to the object with the higher subscript compares higher. Comparison of pointers is guaranteed valid only when the pointers refer to objects in the same array or to the location one past the end of the array.

Microsoft Specific

In Microsoft C++, far pointers that are compared for magnitude (using any operator other than ==) are compared using their offsets only. The segments are not considered in the comparison. Far pointers compared for equality, however, are compared using a segment and offset comparison. ♦

Equality Operators The binary equality operators compare their operands for strict equality or inequality.

Syntax

equality-expression:
relational-expression
equality-expression == *relational-expression*
equality-expression != *relational-expression*

The equality operators, equal to (==) and not equal to (!=), have lower precedence than the relational operators, but they behave similarly in all other respects.

The equal-to operator (==) returns true if both operands have the same value; otherwise it returns false. The not-equal-to operator (!=) returns true if the operands do not have the same value; otherwise it returns false.

Equality operators can compare pointers to members of the same type. In such a comparison, pointer-to-member conversions, as discussed in "Pointer-to-Member Conversions" in Chapter 3, on page 75, are performed. Pointers to members can also be compared to a constant expression that evaluates to 0.

Bitwise Operators

The bitwise operators are:

- Bitwise AND (&)
- Bitwise exclusive OR (^)
- Bitwise inclusive OR (|)

These operators return bitwise combinations of their operands.

Bitwise AND Operator The bitwise AND operator (&) returns the bitwise AND of the two operands. All bits that are on (1) in both the left and right operand are on in the result; bits that are off (0) in either the left or the right operand are off in the result.

Syntax

and-expression:
relational-expression
and-expression & *equality-expression*

Both operands to the bitwise AND operator must be of integral types. The usual arithmetic conversions covered in "Arithmetic Conversions" in Chapter 3, on page 69 are applied to the operands.

Bitwise Exclusive OR Operator The bitwise exclusive OR operator (^) returns the bitwise exclusive OR of the two operands. All bits that are on (1) in either the left or right operand, but not both, are on in the result. Bits that are the same (either on or off) in both operands are off in the result.

Syntax

exclusive-or-expression:
and-expression
exclusive-or-expression ^ *and-expression*

Both operands to the bitwise exclusive OR operator must be of integral types. The usual arithmetic conversions covered in "Arithmetic Conversions" in Chapter 3, on page 69 are applied to the operands.

Bitwise Inclusive OR Operator The bitwise inclusive OR operator (|) returns the bitwise inclusive OR of the two operands. All bits that are on (1) in either the left or right operand are on in the result. Bits that are off (0) in both operands are off in the result.

Syntax

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression | *exclusive-or-expression*

Both operands to the bitwise inclusive OR operator must be of integral types. The usual arithmetic conversions covered in "Arithmetic Conversions" in Chapter 3, on page 69 are applied to the operands.

Logical Operators

The logical operators, logical AND (**&&**) and logical OR (||), are used to combine multiple conditions formed using relational or equality expressions.

Logical AND Operator The logical AND operator (**&&**) returns the integral value 1 if both operands are nonzero; otherwise it returns 0. Logical AND has left-to-right associativity.

Syntax

logical-and-expression:
inclusive-or-expression
logical-and-expression **&&** *inclusive-or-expression*

The operands to the logical AND operator need not be of the same type, but they must be of integral or pointer type. The operands are commonly relational or equality expressions.

The first operand is completely evaluated and all side effects are completed before continuing evaluation of the logical AND expression.

The second operand is evaluated only if the first operand evaluates to true (nonzero). This "short-circuit" evaluation eliminates needless evaluation of the second operand when the logical AND expression has already been determined false. Short-circuit evaluation can be used to prevent null-pointer dereferencing as shown in the following example:

```
char *pch = 0;
...
(pch) && (*pch = 'a');
```

If `pch` is null (0), the right side of the expression is never evaluated. Therefore, the assignment through a null pointer is impossible.

Logical OR Operator The logical OR operator (||) returns the integral value 1 if either operand is nonzero; otherwise it returns 0. Logical OR has left-to-right associativity.

Syntax

logical-or-expression:
logical-and-expression
logical-or-expression || *logical-and-expression*

The operands to the logical OR operator need not be of the same type, but they must be of integral or pointer type. The operands are commonly relational or equality expressions.

The first operand is completely evaluated and all side effects are completed before continuing evaluation of the logical OR expression.

The second operand is evaluated only if the first operand evaluates to false (0). This "short-circuit" evaluation eliminates needless evaluation of the second operand when the logical OR expression has already been determined true.

Assignment Operators

Assignment operators store a value in the object designated by the left operand. There are two kinds of assignment operations: "simple assignment," in which the value of the second operand is stored in the object specified by the first operand, and "compound assignment," in which an arithmetic, shift, or bitwise operation is performed prior to storing the result. All of the assignment operators in Table 4.4 except the = operator are compound assignment operators.

Table 4.4 Assignment Operators

Operator	Meaning
=	Store the value of the second operand in the object specified by the first operand ("simple assignment").
*=	Multiply the value of the first operand by the value of the second operand; store the result in the object specified by the first operand.
/=	Divide the value of the first operand by the value of the second operand; store the result in the object specified by the first operand.
%=	Take modulus of the first operand specified by the value of the second operand; store the result in the object specified by the first operand.
+=	Add the value of the second operand to the value of the first operand; store the result in the object specified by the first operand.
-=	Subtract the value of the second operand from the value of the first operand; store the result in the object specified by the first operand.
<<=	Shift the value of the first operand left the number of bits specified by the value of the second operand; store the result in the object specified by the first operand.
>>=	Shift the value of the first operand right the number of bits specified by the value of the second operand; store the result in the object specified by the first operand.
&=	Obtain the bitwise AND of the first and second operands; store the result in the object specified by the first operand.
^=	Obtain the bitwise exclusive OR of the first and second operands; store the result in the object specified by the first operand.
|=	Obtain the bitwise inclusive OR of the first and second operands; store the result in the object specified by the first operand.

Syntax

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of
=
*=
/=
%=
+=
-=
<<=
>>=
&=
^=
|=

Result of Assignment Operators The assignment operators return the value of the object specified by the left operand after the assignment. The resultant type is the type of the left operand. The result of an assignment expression is always an l-value. These operators have right-to-left associativity. The left operand must be an l-value not of type **const**.

Note In ANSI C, the result of an assignment expression is not an l-value. Therefore, the legal C++ expression `(a += b) += c` is illegal in C.

Simple Assignment The simple assignment operator (=) causes the value of the second operand to be stored in the object specified by the first operand. If both objects are of arithmetic types, the right operand is converted to the type of the left, prior to storing the value.

A **const** pointer of a given type can be assigned to a pointer of the same type. However, a pointer that is not **const** cannot be assigned to a **const** pointer. The following code shows correct and incorrect assignments:

```
int *const cpObject = 0;
int *pObject;

int main()
{
    pObject = cpObject; // OK
    cpObject = pObject; // Error

    return 0;
}
```

Objects of **const** and **volatile** types can be assigned to l-values of types that are just **volatile** or that are neither **const** nor **volatile**.

When the left operand is a pointer to member, the right operand must be of pointer-to-member type or be a constant expression that evaluates to 0. This assignment is valid only in the following cases:

- The right operand is a pointer to a member of the same class as the left operand.
- The left operand is a pointer to a member of a class derived publicly and unambiguously from the class of the right operand.

Just as with other assignments, when a pointer to member assignment is evaluated, the right operand is converted to the type of the left operand before carrying out the assignment.

Assignment to objects of class type (**struct**, **union**, and **class** types) is performed by a function named **operator=**. The default behavior of this operator function is to perform a bitwise copy; however, this behavior can be modified using overloaded operators. (See "Overloaded Operators" in Chapter 12, on page 351 for more information about operator overloading.)

An object of any unambiguously derived class from a given base class can be assigned to an object of the base class; the reverse is not true. For example:

```
#include <iostream.h>

class ABase
{
public:
    ABase() { cout << "constructing ABase\n"; }
};

class ADerived : public ABase
{
public:
    ADerived() { cout << "constructing ADerived\n"; }
};

int main()
{
    ABase aBase;
    ADerived aDerived;

    aBase = aDerived; // OK
    aDerived = aBase; // Error

    return 0;
}
```

Assignments to reference types behave as if the assignment were being made to the object to which the reference points.

For class-type objects, assignment is different from initialization. To illustrate how different assignment and initialization can be, consider the code:

```
UserType1 A;
UserType2 B = A;
```

The above code shows an initializer; it calls the constructor for `UserType1` that takes an argument of type `UserType1`. Given the code

```
UserType1 A;
UserType2 B;

B = A;
```

the assignment statement

```
B = A;
```

can have one of the effects listed below:

- Call the function **operator=** for `UserType2`, provided **operator=** is provided with a `UserType1` argument.
- Call the explicit conversion function `UserType1::operator UserType2`, if such a function exists.
- Call a constructor `UserType2::UserType2`, provided such a constructor is provided, that takes a `UserType1` argument, then copy the result.

Compound Assignment The compound assignment operators, shown in Table 4.4, are specified in the form *e1 op= e2*, where *e1* is an l-value not of **const** type, and *e2* is one of the following:

- An arithmetic type
- A pointer, if *op* is + or –

Compound assignment to an enumerated type generates an error message. If the left operand is of a pointer type, the right operand must be of pointer type, or it must be a constant expression that evaluates to 0. If the left operand is of an integral type, the right operand must not be of a pointer type.

Comma Operator

The comma operator allows grouping two statements where one is expected.

Syntax

expression:
assignment-expression
expression , *assignment-expression*

The comma operator has left-to-right associativity. Two expressions separated by a comma are evaluated left to right. The left operand is guaranteed evaluated, and all side effects are completed before evaluation of the right operand.

Consider the expression

e1 , *e2*

The type and value of the expression are the type and value of *e2*; the result of evaluating *e1* is discarded. The result is an l-value if the right operand is an l-value.

Where the comma has special meaning (for example in actual arguments to functions or aggregate initializers), the comma operator and its operands must be enclosed in parentheses. Therefore, the following function calls are not equivalent:

```
// Declare functions:
void Func( int, int );
void Func( int );

Func( arg1, arg2 );     // Call Func( int, int )
Func( (arg1, arg2) );  // Call Func( int )
```

Expressions with the Conditional Operator

The conditional operator (**? :**) is a ternary operator (it takes three operands). The conditional operator works as follows:

- The first operand is evaluated and all side effects are completed before continuing.
- If the first operand evaluates to true (a nonzero value), the second operand is evaluated.
- If the first operand evaluates to false (0), the third operand is evaluated.

The result of the conditional operator is the result of whichever operand is evaluated—the second or the third. Only one of the last two operands is evaluated in a conditional expression.

Syntax

conditional-expression:
logical-or-expression
logical-or-expression **?** *expression* **:** *conditional-expression*

Conditional expressions have no associativity. The first operand must be of integral or pointer type. The following rules apply to the second and third expressions:

- If both expressions are of the same type, the result is of that type.
- If both expressions are of arithmetic types, usual arithmetic conversions (covered in "Arithmetic Conversions" in Chapter 3, on page 69) are performed to convert them to a common type.

- If both expressions are of pointer types or if one is a pointer type and the other is a constant expression that evaluates to 0, pointer conversions are performed to convert them to a common type.
- If both expressions are of reference types, reference conversions are performed to convert them to a common type.
- If both expressions are of type void, the common type is type void.
- If both expressions are of a given class type, the common type is that class type.

Any combinations of second and third operands not in the list above are illegal. The type of the result is the common type, and it is an l-value if both the second and third operands are of the same type and both are l-values.

Constant Expressions

C++ requires constant expressions—expressions that evaluate to a constant—for declarations of:

- Array bounds
- Selectors in **case** statements
- Bit-field length specification
- Enumeration initializers

Syntax

constant-expression:
conditional-expression

The only operands that are legal in constant expressions are:

- Literals
- Enumeration constants
- Values declared as **const** that are initialized with constant expressions
- **sizeof** expressions

Non-integral constants must be converted (either explicitly or implicitly) to integral types to be legal in a constant expression. Therefore, the following code is legal:

```
const double Size = 11.0;

char chArray[(int)Size];
```

Explicit conversions to integral types are legal in constant expressions; all other types and derived types are illegal except when used as operands to the **sizeof** operator.

The comma operator and assignment operators cannot be used in constant expressions.

Expressions with Explicit Type Conversions

C++ provides implicit type conversion, as described in Chapter 3, "Standard Conversions." You can also specify explicit type conversions when you need more precise control of the conversions applied.

Explicit Type Conversion Operator

C++ allows explicit type conversion using a syntax similar to the function-call syntax. A *simple-type-name* followed by an *expression-list* enclosed in parentheses constructs an object of the specified type using the specified expressions. The following example shows an explicit type conversion to type **int**:

```
int i = int( d );
```

The following example uses the `Point` class defined in "Function Call Results" on page 87 with a few modifications.

```
#include <iostream.h>

class Point
{
public:
    // Define default constructor.
    Point() { _x = _y = 0; }
    // Define another constructor.
    Point( int X, int Y ) { _x = X; _y = Y; }

    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
    void Show()   { cout << "x = " << _x << ", "
                         << "y = " << _y << "\n"; }
private:
    unsigned _x;
    unsigned _y;
};
```

```
int main()
{
    Point Point1, Point2;

    // Assign Point1 the explicit conversion
    //  of ( 10, 10 ).
    Point1 = Point( 10, 10 );

    // Use x() as an l-value by assigning an explicit
    //  conversion of 20 to type unsigned.
    Point1.x() = unsigned( 20 );
    Point1.Show();

    // Assign Point2 the default Point object.
    Point2 = Point();
    Point2.Show();

    return 0;
}
```

The output from this program is:

```
20, 10
0, 0
```

Although the above example demonstrates explicit type conversion using constants, the same technique works to perform these conversions on objects. The following code fragment demonstrates this:

```
int i = 7;
float d;

d = float( i );
```

Explicit type conversions can also be specified using the "cast" syntax. The previous example, rewritten using the cast syntax, follows:

```
d = (float)i;
```

Both cast and function-style conversions have the same results when converting from single values. However, in the function-style syntax, you can specify more than one argument for conversion. This difference is important for user-defined types. Consider a `Point` class, and its conversions:

```
struct Point
{
    Point( short x, short y ) { _x = x; _y = y; }
    ...
    short _x, _y;
};
...
Point pt = Point( 3, 10 );
```

The example above, which uses function-style conversion, shows how to convert two values (one for *x* and one for *y*) to the user-defined type `Point`.

Important Explicit type conversions override the C++ compiler's built-in type checking. These conversions should be used with care.

Syntax

cast-expression:
  unary-expression
  (*type-name*) *cast-expression*

The cast notation must be used for conversions to types that do not have a *simple-type-name* (pointer or reference types, for example). Conversion to types that can be expressed with a *simple-type-name* can be written in either form. See "Type Specifiers" in Chapter 6, on page 168 for more information about what constitutes a *simple-type-name.*

Type definition within casts is illegal. Therefore, the following code generates an error:

```
int AnAction = 2;

Action ThisAction = (enum Action{ Deposit=1, Withdraw,
      GetBalance }) AnAction;
```

Legal Conversions

Explicit conversions from a given type to another type can be done if the conversion can be done using standard conversions. The results are the same.

In addition, the conversions described in the following sections are legal; any other conversions not explicitly defined by the user (for a class type) are illegal.

A value of integral type can be explicitly converted to a pointer if the pointer is large enough to hold the integral value. A pointer that is converted to an integral value can be converted back to a pointer; its value is the same. This identity is given by the following (where *p* represents a pointer of any type):

p == (*type* *) *integral-conversion*(*p*)

Note that with explicit conversions, the compiler does not check whether the converted value fits in the new type except when converting from pointer to integral type or vice versa.

Converting Pointer Types A pointer to one object type can be explicitly converted to a pointer of another object type. A pointer declared as **void** * is considered a pointer to any object type.

A pointer to a base class can be explicitly converted to a pointer to a derived class as long as these conditions are met:

- There is an unambiguous conversion.
- The base class is not declared as **virtual** at any point.

Because conversion to type **void** * can change the representation of an object, there is no guarantee that the conversion *type1** **void** * *type2** is equivalent to the conversion *type1** *type2** (which is a change in value only).

When such a conversion is performed, the result is a pointer to the subobject of the original object representing the base class.

See Chapter 9, "Derived Classes," for more information about ambiguity and virtual base classes.

A pointer to a function can be explicitly converted to a pointer-to-object type, provided the pointer to the object has enough bits. This restriction is sensitive to memory-model selection; as a result, memory-model selection can change the size of pointers to functions. Code that works in small model can fail in medium model, as shown in this example:

```
void func();

int main()
{
    int iPtr;
    void (*fPtr)() = func;

    iPtr = (int)fPtr;

    return 0;
}
```

In small model (for the 16-bit compiler), both objects of type **int** and pointers to functions are 16 bits. However, changing to medium model changes pointers to functions to 32-bit pointers, causing a compilation error.

C++ allows explicit conversions of pointers to objects or functions to type **void ***.

Pointers to object types can be explicitly converted to pointers to functions if the function pointer type has enough bits to accommodate the pointer to object type.

A pointer to a **const** object can be explicitly converted to a pointer not of **const** type. The result of this conversion points to the original object. An object of **const** type, or a reference to an object of **const** type, can be cast to a reference to a non-**const** type. The result is a reference to the original object. The original object was probably declared as **const** because it was to remain constant across the duration of the program. Therefore, an explicit conversion defeats this safeguard, allowing modification of such objects. The behavior in such cases is undefined.

A pointer to an object of **volatile** type can be cast to a pointer to a non**volatile** type. The result of this conversion refers to the original object. Similarly, an object of **volatile** type can be cast to a reference to a non**volatile** type.

Converting the Null Pointer The null pointer (0) is converted into itself.

Converting to a Forward-Reference Class Type A class that has been declared but not yet defined (a forward reference) can be used in a pointer cast. In this case, the compiler returns a pointer to the original object, not to a subobject as it might if the class's relationships were known.

Converting to Reference Types Any object whose address can be converted to a given pointer type can also be converted to the analogous reference type. For example, any object whose address can be converted to type **char *** can also be converted to type **char &**. No constructors or class conversion functions are called to make a conversion to a reference type.

Objects or values can be converted to class-type objects only if a constructor or conversion operator has been provided specifically for this purpose. For more information about these user-defined functions, see "Conversion Constructors" in Chapter 11, on page 313.

Conversion of a reference to a base class, to a reference to a derived class (and vice versa) is done the same way as for pointers.

A cast to a reference type results in an l-value. The results of casts to other types are not l-values. Operations performed on the result of a pointer or reference cast are still performed on the original object.

Converting Among Pointer-to-Member Types A pointer to a member can be converted to a different pointer-to-member type subject to these rules: Either the pointers must both be pointers to members in the same class or they must be pointers to members of classes, one of which is derived unambiguously from the other. When converting pointer-to-member functions, the return and argument types must match.

Expressions with Pointer-to-Member Operators

The pointer-to-member operators, **.*** and **->***, return the value of a specific class member for the object specified on the left side of the expression. The following example shows how to use these operators:

```
#include <iostream.h>

class Window
{
public:
    void Paint(); // Causes window to repaint.
    int WindowId;
};

// Define derived types pmfnPaint and pmWindowId.
// These types are pointers to members Paint() and
// WindowId, respectively.
void (Window::*pmfnPaint)() = &Window::Paint;
int Window::*pmWindowId = &Window::WindowId;

int main()
{
    Window AWindow;
    Window *pWindow = new Window;
```

```
    // Invoke the Paint function normally, then
    // use pointer to member.
    AWindow.Paint();
    (AWindow.*pmfnPaint)();

    pWindow->Paint();
    (pWindow->*pmfnPaint)();

    int Id;
    // Retrieve window id.
    Id = AWindow.*pmWindowId;
    Id = pWindow->*pmWindowId;

    return 0;
}
```

In the above example, a pointer to a member, `pmfnPaint`, is used to invoke the member function `Paint`. Another pointer to a member, `pmWindowId`, is used to access the `WindowId` member.

Syntax

pm-expression:
cast-expression
pm-expression **.*** *cast-expression*
pm-expression **–>*** *cast-expression*

The binary operator **.*** combines its first operand, which must be an object of class type, with its second operand, which must be a pointer-to-member type.

The binary operator **–>*** combines its first operand, which must be a pointer to an object of class type, with its second operand, which must be a pointer-to-member type.

In an expression containing the **.*** operator, the first operand must be of the class type of the pointer to member specified in the second operand or of a type unambiguously derived from that class.

In an expression containing the **–>*** operator, the first operand must be of the type "pointer to the class type" of the type specified in the second operand, or it must be of a type unambiguously derived from that class.

Consider the following classes and program fragment:

```
class BaseClass
{
public:
    BaseClass(); // Base class constructor.
    void Func1();
};

// Declare a pointer to member function Func1.
void (BaseClass::*pmfnFunc1)() = &BaseClass::Func1;

class Derived : public BaseClass
{
public:
    Derived();  // Derived class constructor.
    void Func2();
};

// Declare a pointer to member function Func2.
void (Derived::*pmfnFunc2)() = &Derived::Func2;

int main()
{
    BaseClass ABase;
    Derived ADerived;

    (ABase.*pmfnFunc1)();     // OK: defined for BaseClass.
    (ABase.*pmfnFunc2)();     // Error: cannot use base class to
                              //  access pointers to members of
                              //  derived classes.
    (ADerived.*pmfnFunc1)(); // OK: Derived is unambiguously
                              //  derived from BaseClass.
    (ADerived.*pmfnFunc2)(); // OK: defined for Derived.

    return 0;
}
```

The result of the **.*** or **->*** pointer-to-member operators is an object or function of the type specified in the declaration of the pointer to member. So, in the example above, the result of the expression `ADerived.*pmfnFunc1()` is a pointer to a function that returns **void**. This result is an l-value if the second operand is an l-value.

Note If the result of one of the pointer-to-member operators is a function, then the result can be used only as an operand to the function call operator.

4.2 Semantics of Expressions

This section explains when, and in what order expressions are evaluated ("Order of Evaluation" on this page and "Sequence Points" on page 129). In addition, it describes certain expressions that are ambiguous in their meaning ("Gray Expressions" on page 130) and compatible types that can be used in expressions ("Notation in Expressions" on page 130).

Order of Evaluation

This section discusses the order in which expressions are evaluated but does not explain the syntax or the semantics of the operators in these expressions. The earlier sections in this chapter provide a complete reference for each of these operators.

Expressions are evaluated according to the precedence and grouping of their operators. (Table 1.2 in Chapter 1, "Lexical Conventions," shows the relationships the C++ operators impose on expressions.) Consider this example:

```
#include <iostream.h>

int main()
{
    int a = 2, b = 4, c = 9;

    cout << a + b * c << "\n";
    cout << a + (b * c) << "\n";
    cout << (a + b) * c << "\n";

    return 0;
}
```

The output from the above code is:

```
38
38
54
```

Figure 4.1 shows the order in which the expressions are evaluated.

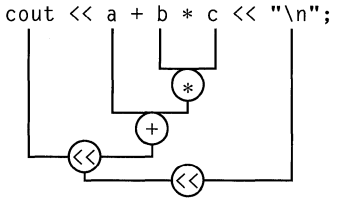


Figure 4.1 Expression-Evaluation Order

The order in which the expression shown in Figure 4.1 is evaluated is determined by the precedence and associativity of the operators:

1. Multiplication (*) has the highest precedence in this expression; hence the subexpression `b * c` is evaluated first.
2. Addition (+) has the next highest precedence, so `a` is added to the product of `b` and `c`.
3. Left shift (<<) has the lowest precedence in the expression, but there are two occurrences. Because the left-shift operator groups left-to-right, the left subexpression is evaluated, then the right one.

When parentheses are used to group the subexpressions, they alter the precedence and also the order in which the expression is evaluated, as shown in Figure 4.2.

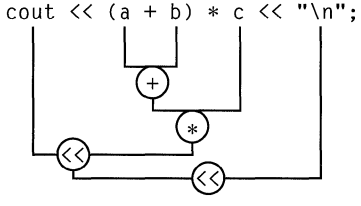


Figure 4.1 Expression-Evaluation Order with Parentheses

Expressions such as those shown in Figure 4.2 are evaluated purely for their side effects—in this case, to transfer information to the standard output device.

Note The left-shift operator is used to insert an object in an object of class **ostream**. It is sometimes called the "insertion" operator when used with iostream. For more about the iostream library, see Chapters 18 and 19 of the *Class Libraries User's Guide* manual and Part 3 of the *Class Libraries Reference* manual.

Sequence Points

An expression can modify an object's value only once between consecutive "sequence points."

Microsoft Specific

The C++ language definition does not currently specify sequence points, Microsoft C++ uses the same sequence points as ANSI C for any expression involving C operators and not involving overloaded operators. When operators are overloaded, the semantics change from operator sequencing to function-call sequencing. Microsoft C++ uses the following sequence points:

- Left operand of the logical AND operator. The left operand of the logical AND operator is completely evaluated and all side effects completed before continuing. There is no guarantee that the right operand of the logical AND operator will be evaluated.
- Left operand of the logical OR operator. The left operand of the logical OR operator is completely evaluated and all side effects completed before continuing. There is no guarantee that the right operand of the logical OR operator will be evaluated.
- Left operand of the comma operator. The left operand of the comma operator is completely evaluated and all side effects completed before continuing. Both operands of the comma operator are always evaluated.
- Function-call operator. All arguments to a function, including default arguments, are evaluated and all side effects completed prior to entry to the function. No order of evaluation among the arguments or the function-call expression is specified.
- First operand of the conditional operator. The first operand of the conditional operator is completely evaluated and all side effects completed before continuing.

- The end of a full initialization expression.
- The expression in an expression statement. Expression statements consist of an optional expression followed by a semicolon (**;**). The expression is completely evaluated for its side effects.
- The controlling expression in a selection (**if** or **switch**) statement. The expression is completely evaluated and all side effects completed before the code dependent on the selection is executed.
- The controlling expression of a **while** or **do** statement. The expression is completely evaluated and all side effects completed before any statements in the next iteration of the **while** or **do** loop are executed.
- Each of the three expressions of a **for** statement. Each expression is completely evaluated and all side effects completed before moving to the next expression. The three expressions are (collectively) completely evaluated and all side effects completed before any statements in the next iteration of the **for** loop are executed.
- The expression in a **return** statement. The expression is completely evaluated and all side effects completed before control returns to the calling function. ♦

Gray Expressions

Certain expressions, called "gray expressions," are ambiguous in their meaning. These expressions occur most frequently when an object's value is modified more than once in the same expression. Gray expressions rely on a particular order of evaluation where the language does not define one. Consider the following example:

```
int i = 7;

func( i, ++i );
```

The C++ language does not guarantee in which order the arguments to a function call are evaluated. Therefore, in the example above, `func` could receive the values 7 and 8, or 8 and 8 for its parameters, depending on whether the parameters are evaluated from left to right or from right to left.

Notation in Expressions

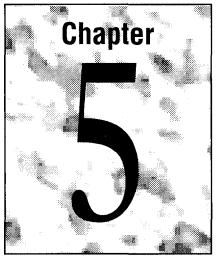
The C++ language specifies certain compatibilities when specifying operands. Table 4.5 shows the types of operands acceptable to operators that require operands of type *type*.

Table 4.5 Operand Types Acceptable to Operators

Type Expected	Types Allowed
type	**const** *type* **volatile** *type* *type***&** **const** *type***&** **volatile** *type***&** **volatile const** *type* **volatile const** *type***&**
*type**	type* **const** *type** **volatile** *type** **volatile const**
const *type*	*type* **const** *type* **const** *type***&**
volatile *type*	type **volatile** *type* **volatile** *type***&**

Because the preceding rules can always be used in combination, a **const** pointer to a **volatile** object can be supplied where a pointer is expected.

Statements



C++ statements are the program elements that control how, and in what order, objects are manipulated.

Statements fall into one of the following categories:

- Expression statements. These statements, discussed on page 136, evaluate an expression either for its side effects or for its return value.
- Null statements. These statements, discussed on page 136, can be provided where a statement is required by the C++ syntax but where no action is to be taken.
- Compound statements. These statements, discussed on page 137, are groups of statements enclosed in curly braces ({ }). They can be used wherever the grammar calls for a single statement.
- Selection statements. These statements, discussed on page 138, perform a test; they then execute one section of code if the test evaluates to true (nonzero). They may execute another section of code if the test evaluates to false.
- Iteration statements. These statements, discussed on page 142, provide for repeated execution of a block of code until some termination criterion is met.
- Jump statements. These statements, discussed on page 147, either transfer control immediately to another location in the function or return control from the function.
- Declaration statements. Declarations, discussed on page 149, introduce a name into a program. (Chapter 6 provides more detailed information about declarations.)

5.1 Overview

C++ statements are executed sequentially, except when an expression statement, a selection statement, an iteration statement, or a jump statement specifically modifies that sequence.

Syntax

statement:
labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
declaration-statement

In most cases, the C++ statement syntax is identical to that of ANSI C. The primary difference between the two is that in C, declarations are allowed only at the start of a block; C++ adds the *declaration-statement*, which effectively removes this restriction. This allows introduction of variables at a point in the program where a precomputed intialization value is present.

Declaring variables inside blocks also allows you to exercise precise control over the scope and lifetime of those variables.

5.2 Labeled Statements

In order for program control to be transferred directly to a given statement, the statement must be labeled.

Syntax

labeled-statement:
identifier **:** *statement*
case *constant-expression* **:** *statement*
default **:** *statement*

Using Labels with the goto Statement

The appearance of an *identifier* label in the source program declares a label. Only a **goto** statement can transfer control to an *identifier* label. The following code fragment illustrates use of the **goto** statement and an *identifier* label to escape a tightly nested loop:

```
for( p = 0; p < NUM_PATHS; ++p )
{
    NumFiles = FillArray( pFileArray, pszFNames )
    for( i = 0; i < NumFiles; ++i )
    {
        if( (pFileArray[i] = fopen( pszFNames[i], "r" )) == NULL )
            goto FileOpenError;

        // Process the files that were opened.
    }
}

FileOpenError:
    cerr << "Fatal file open error. Processing interrupted.\n" );
```

In the above example, the **goto** statement transfers control directly to the statement that prints an error message if an unknown file-open error occurs.

The label has function scope and cannot be redeclared within the function. However, the same name can be used as a label in different functions.

Using Labels in the case Statement

Labels that appear after the **case** keyword cannot also appear outside a **switch** statement. (This restriction also applies to the **default** keyword.) The following code fragment shows the correct use of **case** labels:

```
// Sample Microsoft Windows message processing loop.
switch( msg )
{
case WM_TIMER:    // Process timer event.
    SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++] );
    ShowWindow( hWnd, SW_SHOWNA );
    nIcon %= 14;
    Yield();
    break;

case WM_PAINT:
    // Obtain a handle to the device context.
    // BeginPaint will send WM_ERASEBKGND if appropriate.

    memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
    hDC = BeginPaint( hWnd, &ps );

    // Inform Windows that painting is complete.

    EndPaint( hWnd, &ps );
    break;
```

```
case WM_CLOSE:
    // Close this window and all child windows.

    KillTimer( hWnd, TIMER1 );
    DestroyWindow( hWnd );
    if( hWnd == hWndMain )
        PostQuitMessage( 0 );  // Quit the application.

    break;

default:
    // This choice is taken for all messages not specifically
    //  covered by a case statement.

         return DefWindowProc( hWnd, Message, wParam, lParam );
         break;
}
```

5.3 Expression Statement

Expression statements cause expressions to be evaluated. No transfer of control or iteration takes place as a result of an expression statement.

Syntax

expression-statement:
  expression$_{opt}$;

All expressions in an expression statement are evaluated and all side effects are complete before the next statement is executed. The most common expression statements are assignments and function calls. C++ also provides a null statement.

The Null Statement

The "null statement" is an expression statement with the *expression* missing. It is useful when the syntax of the language calls for a statement but no expression evaluation is called for. It consists of a semicolon.

Null statements are commonly used as placeholders in iteration statements or as statements on which to place labels at the end of compound statements or functions.

The following code fragment shows how to copy one string to another and incorporates the null statement:

```
char *strcpy( char *Dest, const char *Source )
{
    char *DestStart = Dest;

    // Assign value pointed to by Source to
    //  Dest until the end-of-string 0 is
    //  encountered.
    while( *Dest++ = *Source++ )
        ;                           // Null statement.

    return DestStart;
}
```

5.4 Compound Statements (Blocks)

A compound statement consists of zero or more statements enclosed in curly braces ({ }). A compound statement can be used anywhere a statement is expected. Compound statements are commonly called "blocks."

Syntax

compound-statement:
  { *statement-list*$_{opt}$ }

statement-list:
  statement
  statement-list statement

The following example uses a compound statement as the *statement* part of the **if** statement (see "The if Statement" on page 138 for details about the syntax):

```
if( Amount > 100 )
{
    cout << "Amount was too large to handle\n";
    Alert();
}
else
    Balance -= Amount;
```

Note Because a declaration is a statement, a declaration can be one of the statements in the *statement-list*. The result of this is that names declared inside a compound statement, but not explicitly declared as static, have local scope and (for objects) lifetime. See "Scope" in Chapter 2, on page 28 for details about treatment of names with local scope.

5.5 Selection Statements

The C++ selection statements, **if** and **switch**, provide a means to conditionally execute sections of code.

Syntax

selection-statement:
if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*
switch (*expression*) *statement*

The *statement* in the **if**, **else**, and **switch** statements cannot be a declaration.

The if Statement

The **if** statement evaluates the expression enclosed in parentheses. The expression must be of arithmetic or pointer type, or it must be of a class type that defines an unambiguous conversion to an arithmetic or pointer type. (For information about conversions, see Chapter 3, "Standard Conversions.")

In both forms of the **if** syntax, if the expression evaluates to a nonzero value (true), the statement dependent on the evaluation is executed; otherwise it is skipped.

In the **if...else** syntax, the second statement is executed if the result of evaluating the expression is zero.

The **else** clause of an **if...else** statement is associated with the **if** statement immediately preceding it. The following code fragment demonstrates how this works:

```
if( condition1 == true )
    if( condition2 == true )
        cout << "condition1 true; condition2 true\n";
    else
        cout << "condition1 true; condition2 false\n";
else
    cout << "condition 1 false\n";
```

Many programmers use curly braces ({ }) to explicitly clarify the pairing of complicated **if** and **else** clauses. The following example uses curly braces for this purpose:

```
if( condition1 == true )
{
    if( condition1 == true )
        cout << "condition1 true; condition2 true\n";
    else
        cout << "condition1 true; condition2 false\n";
}
else
    cout << "condition 1 false\n";
```

While the braces are not strictly necessary, they clarify the pairing between **if** and **else** statements.

The switch Statement

The C++ **switch** statement allows selection among multiple sections of code, depending on the value of an expression. The expression enclosed in parentheses, the "controlling expression," must be of an integral type or of a class type for which there is an unambiguous conversion to integral type. Integral promotion is performed as described in "Integral Promotions" in Chapter 3, on page 66.

The **switch** statement causes an unconditional jump to, into, or past the statement that is the "switch body," depending on the value of the controlling expression, the values of the **case** labels, and the presence or absence of a **default** label. The switch body is normally a compound statement (although this is not a syntactic requirement). Usually, some of the statements in the switch body are labeled with **case** labels or with the **default** label. Labeled statements are not syntactic requirements, but the **switch** statement is meaningless without them. The **default** label can appear only once.

Syntax

case *constant-expression* **:** *statement*

default **:** *statement*

The *constant-expression* in the **case** label is converted to the type of the controlling expression and is then compared for equality. In a given **switch** statement, no two *constant-expressions* in **case** statements can evaluate to the same value. The behavior is shown in Table 5.1.

Table 5.1 Switch Statement Behavior

Condition	Action
Converted value matches that of the promoted controlling expression.	Control is transferred to the statement following that label.
None of the constants match the constants in the **case** labels; **default** label is present.	Control is transferred to the **default** label.
None of the constants match the constants in the **case** labels; **default** label is not present.	Control is transferred to the statement after the **switch** statement.

The **switch** statement can contain declarations as long as they are reachable—that is, not bypassed by all possible execution paths. Names introduced using these dec larations have local scope. The following code fragment shows how the **switch** statement works:

```
switch( tolower( *argv[1] ) )
{
    // Error. Unreachable declaration.
    char szChEntered[] = "Character entered was: ";

case 'a' :
    {
    // Declaration of szChEntered OK. Local scope.
    char szChEntered[] = "Character entered was: ";
    cout << szChEntered << "a\n";
    }
    break;

case 'b' :
    // Value of szChEntered undefined.
    cout << szChEntered << "b\n";
    break;

default:
    // Value of szChEntered undefined.
    cout << szChEntered << "neither a nor b\n";
    break;
}
```

A **switch** statement can be nested. In such cases, **case** or **default** labels associate with the most deeply nested **switch** statements that enclose them. For example:

```
switch( msg )
{
case WM_COMMAND:      // Windows command. Find out more.
    switch( wParam )
    {
    case IDM_F_NEW:   // File New menu command.
        delete wfile;
        wfile = new WinAppFile;
        break;
    case IDM_F_OPEN:  // File Open menu command.
        wfile->FileOpenDlg();
        break;
    ...
    }

case WM_CREATE:       // Create window.
    ...
    break;

case WM_PAINT:        // Window needs repainting.
    ...
    break;

default:
    return DefWindowProc( hWnd, Message, wParam, lParam );
}
```

The above code fragment from a Microsoft Windows™ message loop shows how **switch** statements can be nested. The **switch** statement that selects on the value of `wParam` is executed only if `msg` is **WM_COMMAND**. The **case** labels for menu selections, `IDM_F_NEW` and `IDM_F_OPEN`, associate with the inner **switch** statement.

Control is not impeded by **case** or **default** labels. To stop execution at the end of a part of the compound statement, insert a **break** statement. This transfers control to the statement after the **switch** statement. This example demonstrates how control "drops through" unless a **break** statement is used:

```
BOOL fClosing = FALSE;

...

switch( wParam )
{
case IDM_F_CLOSE:       // File close command.
    fClosing = TRUE;

case IDM_F_SAVE:        // File save command.
    if( document->IsDirty() )
        if( document->Name() == "UNTITLED" )
            FileSaveAs( document );
        else
            FileSave( document );

    if( fClosing )
        document->Close();

    break;
}
```

The preceding code shows how to take advantage of the fact that **case** labels do not impede the flow of control. If the `switch` statement transfers control to `IDM_F_SAVE`, `fClosing` is `FALSE`. Therefore, after the file is saved, the document is not closed. However, if the **switch** statement transfers control to `IDM_F_CLOSE`, `fClosing` is set to `TRUE`, and the code to save a file is executed.

5.6 Iteration Statements

Iteration statements cause statements (or compound statements) to be executed zero or more times, subject to some loop-termination criteria. When these statements are compound statements, they are executed in order, except when either the **break** statement or the **continue** statement is encountered. (For a description of these statements, see "The break Statement" and "The continue Statement" on page 147.)

C++ provides three iteration statements—**while**, **do**, and **for**. Each of these iterates until its termination expression evaluates to zero (false), or until loop termination is forced with a **break** statement. Table 5.2 summarizes these statements and their actions; each is discussed in detail in the sections that follow.

Table 5.2 C++ Iteration Statements

Statement	Evaluated At	Initialization	Increment
while	Top of loop	No	No
do	Bottom of loop	No	No
for	Top of loop	Yes	Yes

Syntax

iteration-statement:
while (*expression*) *statement*
do *statement* **while** (*expression*) ;
for (*for-init-statement* $expression_{opt}$; $expression_{opt}$) *statement*

for-init-statement:
expression-statement
declaration-statement

The statement part of an iteration statement cannot be a declaration. However, it can be a compound statement containing a declaration.

The while Statement

The **while** statement executes a *statement* repeatedly until the termination condition (the *expression*) specified evaluates to zero. The test of the termination condition takes place before each execution of the loop; therefore, a **while** loop executes zero or more times, depending on the value of the termination expression. The code below uses a **while** loop to trim trailing spaces from a string:

```
char *trim( char *szSource )
{
    char *pszEOS;

    // Set pointer to end of string to point to the character just
    //  before the 0 at the end of the string.
    pszEOS = szSource + strlen( szSource ) - 1;

    while( pszEOS >= szSource && *pszEOS == ' ' )
        *pszEOS-- = '\0';

    return szSource;
}
```

The termination condition is evaluated at the top of the loop. If there are no trailing spaces, the loop never executes.

The expression must be of an integral type, a pointer type, or a class type with an unambiguous conversion to an integral type.

The do Statement

The **do** statement executes a *statement* repeatedly until the specified termination condition (the *expression*) evaluates to zero. The test of the termination condition is made after each execution of the loop; therefore, a **do** loop executes one or more times, depending on the value of the termination expression. The following function uses the **do** statement to wait for the user to press a specific key:

```
void WaitKey( char ASCIICode )
{
    char chTemp;

    do
    {
        chTemp = getch();
    }
    while( chTemp != ASCIICode );
}
```

A **do** loop rather than a **while** loop is used in the above code—with the **do** loop, the **getch** function is called to get a keystroke before the termination condition is evaluated. This function can be written using a **while** loop, but not as concisely:

```
void WaitKey( char ASCIICode )
{
    char chTemp;

    chTemp = getch();

    while( chTemp != ASCIICode )
    {
        chTemp = getch();
    }
}
```

The expression must be of an integral type, a pointer type, or a class type with an unambiguous conversion to an integral type.

The for Statement

The **for** statement can be divided into three separate parts, as shown in Table 5.3.

Table 5.3 for Loop Elements

Syntax Name	When Executed	Contents
for-init-statement	Before any other element of the **for** statement or the substatement.	Often used to initialize loop indices. It can contain expressions or declarations.
expression1	Before execution of a given iteration of the loop, including the first iteration.	An expression that evaluates to an integral type or a class type that has an unambiguous conversion to an integral type.
expression2	At the end of each iteration of the loop; *expression1* is tested after *expression2* is evaluated.	Normally used to increment loop indices.

The *for-init-statement* is commonly used to declare and initialize loop-index variables. The *expression1* is often used to test for loop-termination criteria. The *expression2* is commonly used to increment loop indices.

The **for** statement executes the *statement* repeatedly until *expression1* evaluates to zero. The *for-init-statement*, *expression1*, and *expression2* fields are all optional.

The following **for** loop:

```
for( for-init-statement; expression1; expression2 )
{
    // Statements
}
```

is equivalent to the following **while** loop:

```
for-init-statement;
while( expression1 )
{
    // Statements
    expression2
}
```

A convenient way to specify an infinite loop using the **for** statement is:

```
for( ; ; )
{
    // Statements to be executed.
}
```

This is equivalent to:

```
while( 1 )
{
    // Statements to be executed.
}
```

The initialization part of the **for** loop can be a declaration statement, or any other type of statement, including the null statement. The initializations can include any sequence of expressions and declarations, separated by commas. Any object declared inside a *for-init-statement* has local scope, as if it had been declared immediately prior to the **for** statement. While the name of the object can be used in more than one **for** loop in the same scope, the declaration can appear only once. For example:

```
int main()
{
    for( int i = 0; i < 100; ++i )
        cout << i << "\n";

    // The loop index, i, cannot be declared in the
    //  for-init-statement here because it is still in scope.
    for( i = 100; i >= 0; --i )
        cout << i << "\n";

    return 0;
}
```

Although the three fields of the **for** statement are normally used for initialization, testing for termination, and incrementing, they are not restricted to these uses. For example, the following code prints the numbers 1 to 100. The substatement is the null statement:

```
#include <iostream.h>

int main()
{
    for( int i = 0; i < 100; cout << ++i << "\n" )
        ;


    return 0;
}
```

5.7 Jump Statements

The C++ jump statements perform an immediate local transfer of control.

Syntax

jump-statement:
break ;
continue ;
return *expression*$_{opt}$;
goto *identifier* ;

The break Statement

The **break** statement is used to exit an iteration or **switch** statement. It transfers control to the statement immediately following the iteration substatement or **switch** statement.

The **break** statement terminates only the most tightly enclosing loop or **switch** statement. In loops, **break** is used to terminate before the termination criteria evaluate to 0. In the **switch** statement, **break** is used to terminate sections of code—normally before a **case** label. The following example illustrates the use of the **break** statement in a **for** loop:

```
for( ; ; )    // No termination condition.
{
    if( List->AtEnd() )
        break;

    List->Next();
}

cout << "Control transfers to here.\n";
```

Note There are other ways to escape a loop as simply as the one above. In more complex loops, where it can be difficult to tell if the loop should be terminated before several statements have been executed, using the **break** statement makes more sense.

For an example of using the **break** statement within the body of a **switch** statement, see "The switch Statement" on page 139.

The continue Statement

The **continue** statement forces immediate transfer of control to the loop-continuation statement of the smallest enclosing loop. (The "loop-continuation" is the statement that contains the controlling expression for the loop.) Therefore, the **continue** statement can appear only in the dependent *statement* of an iteration

statement (although it may be the sole statement in that *statement*). In a **for** loop, execution of a **continue** statement causes *expression2* to be evaluated, then *expression1*.

The following example shows how the **continue** statement can be used to bypass sections of code and skip to the next iteration of a loop:

```
// Get a character that is a member of the zero-terminated
//  string, szLegalString. Return the index of the character
//  entered.
int GetLegalChar( char *szLegalString )
{
    char *pch;

    do
    {
        char ch = getch();

        // Use strchr library function to determine if the
        //  character read is in the string. If not, use the
        //  continue statement to bypass the rest of the
        //  statements in the loop.
        if( (pch = strchr( szLegalString, ch )) == NULL )
            continue;

        // A character that was in the string szLegalString
        //  was entered. Return its index.
        return (pch - szLegalString);

        // The continue statement transfers control to here.
    } while( 1 );

    return 0;
}
```

The return Statement

The **return** statement allows a function to immediately transfer control back to the calling function (or, in the case of the main function, transfer control back to the operating system). The **return** statement accepts an expression, which is the value passed back to the calling function. Functions of type **void**, constructors, and destructors cannot specify expressions in the **return** statement; functions of all other types must specify an expression in the **return** statement.

The expression, if specified, is converted to the type specified in the function declaration, as if an initialization were being performed. Conversion from the type of the expression to the **return** type of the function can cause temporary objects to be created. See "Temporary Objects" in Chapter 11, on page 311 for more information about how and when temporaries are created.

When the flow of control exits the block enclosing the function definition, the result is the same as if a **return** statement with no expression had been executed. This is illegal for functions that are declared as returning a value.

A function can have any number of **return** statements.

The goto Statement

The **goto** statement performs an unconditional transfer of control to the named label. The label must be in the current function.

For more information about labels and the **goto** statement, see "Labeled Statements" and "Using Labels with the goto Statement" on page 134.

5.8 Declaration Statements

Declaration statements introduce new names into the current scope. These names can be:

- Type names (**class**, **struct**, **union**, **enum**, **typedef**, and pointer-to-member)
- Object names
- Function names

Syntax

declaration-statement:
declaration

If a declaration within a block introduces a name that is already declared outside the block, the previous declaration is hidden for the duration of the block. After termination of the block, the previous declaration is again visible.

Multiple declarations of the same name in the same block are illegal.

For more information about declarations and name hiding, see "Declarations and Definitions" and "Scope" in Chapter 2, on pages 27 and 28, respectively.

Declaration of Automatic Objects

In C++, objects can be declared with automatic storage class using the **auto** or **register** keyword. If no storage-class keyword is used for a local object (an object declared inside a function), **auto** is assumed. C++ handles initializations and declarations of these objects differently than objects declared with static storage classes.

Initialization

Each time declaration statements for objects of storage class **auto** or **register** are executed, initialization takes place. The following example, from "The continue Statement" on page 147 shows initialization of the automatic object `ch` inside the `do` loop.

```
// Get a character that is a member of the zero-terminated
//  string, szLegalString. Return the index of the character
//  entered.
int GetLegalChar( char *szLegalString )
{
    char *pch;

    do
    {

        // This declaration statement is executed once for each
        //  execution of the loop.
        char ch = getch();

        if( (pch = strchr( szLegalString, ch )) == NULL )
            continue;

        // A character that was in the string szLegalString
        //  was entered. Return its index.
        return (pch - szLegalString);
    } while( 1 );
}
```

For each iteration of the loop (each time the declaration is encountered), the macro `getch` is evaluated and `ch` is initialized with the results. When control is transferred outside the block using the `return` statement, `ch` is destroyed (in this case, the storage is deallocated).

See "Storage Classes" in Chapter 2, on page 46 for another example of initialization.

Destruction

Objects declared in a loop are destroyed once per iteration of the loop, on exit from the block, or when control transfers to a point prior to the declaration. Objects declared in a block that is not a loop are destroyed on exit from the block, or when control transfers to a point prior to the declaration.

Note Destruction can mean simply deallocating the object, or, for class-type objects, invoking the object's destructor.

When a **jump** statement transfers control out of a loop or block, objects declared in the block transferred from are destroyed; objects in the block transferred to are not destroyed.

When control is transferred to a point prior to a declaration, the object is destroyed.

Transfers of Control

Using the **goto** statement or a **case** label in a **switch** statement, it is possible to specify a program that branches past an automatic object initializer. Such code is illegal unless the declaration that contains the initializer is in a block enclosed by the block in which the **jump** statement occurs.

The following example shows a loop that declares and initializes the objects `total`, `ch`, and `i`. There is also an erroneous `goto` statement that transfers control past an initializer.

```
// Read input until a nonnumeric character is entered.
while( 1 )
{
    int total = 0;

    char ch = getch();

    if( ch >= '0' || ch <= '9' )
    {
        goto Label1;        // Error: transfers past initialization
                            //  of i.

        int i = ch - '0';
Label1:
        total += i;
    } // i would be destroyed here if the
      //  goto error were not present.
    else
        // Break statement transfers control out of loop,
        //  destroying total and ch.
        break;
}
```

In the above example, the `goto` statement tries to transfer control past the initialization of `i`. However, if `i` were declared but not initialized, the transfer would be legal.

The objects `total` and `ch`, declared in the block that serves as the *statement* of the `while` statement, are destroyed when that block is exited using the `break` statement.

Declaration of Static Objects

An object can be declared with static storage class using the **static** or **extern** keyword. Local objects must be explicitly declared as **static** or **extern** to have static storage class. Global objects (objects declared outside all functions) that are declared with no storage-class specifier are assumed to be **extern**.

Initialization

Global objects of static storage class are initialized at program startup. (For more information about construction and destruction of global objects, see "Additional Startup Considerations" and "Additional Termination Considerations" in Chapter 2, on pages 43 and 44, respectively.)

Local objects declared as **static** are initialized the first time their declarations are encountered in the program flow. The following class, introduced in Chapter 2, shows how this works:

```
#include <iostream.h>
#include <string.h>

// Define a class that logs initializations and destructions.
class InitDemo
{
public:
    InitDemo( char *szWhat );
    ~InitDemo();
private:
    char *szObjName;
};

// Constructor for class InitDemo.
InitDemo::InitDemo( char *szWhat )
{
    if( szWhat != 0 && strlen( szWhat ) > 0 )
    {
        szObjName = new char[ strlen( szWhat ) + 1 ];
        strcpy( szObjName, szWhat );
    }
    else
        szObjName = 0;

    clog << "Initializing: " << szObjName << "\n";
}
```

```
// Destructor for InitDemo.
InitDemo::~InitDemo()
{
    if( szObjName != 0 )
    {
        clog << "Destroying: " << szObjName << "\n";
        delete szObjName;
    }
}

// Main function.
int main( int argc, char *argv[] )
{
    if( argc < 2 )
    {
        cerr << "Supply a one-letter argument.\n";
        return -1;
    }

    if( *argv[1] == 'a' )
    {
        cout << "*argv[1] was an 'a'\n";

        // Declare static local object.
        static InitDemo I1( "static I1" );
    }
    else
        cout << "*argv[1] was not an 'a'\n";

    return 0;
}
```

If the command-line argument supplied to this program starts with the lowercase letter "a," the declaration of `I1` is executed, the initialization takes place, and the result is:

```
*argv[1] was an 'a'
Initializing: static I1
Destroying: static I1
```

Otherwise, the flow of control bypasses the declaration of `I1`, and the result is:

```
*argv[1] was not an 'a'
```

When a static local object is declared with an initializer that does not evaluate to a constant expression, the object is given the value 0 (converted to the appropriate type) at the point before execution enters the block for the first time. However, the object is not visible and no constructors are called until the actual point of declaration.

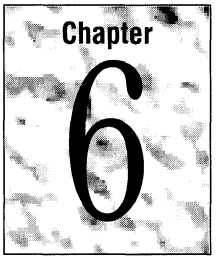
At the point of declaration, the object's constructor (if the object is of a class type) is called as expected. (Static local objects are only initialized the first time they are seen.)

Destruction

Local static objects are destroyed during processing of the termination functions specified by **atexit**.

If a static object was not constructed because the program's flow of control bypassed its declaration, no attempt is made to destroy that object.

Declarations



Declarations introduce new names into a program. They can be used to:

- Specify storage class, type, and linkage for an object
- Specify storage class, type, and linkage for a function
- Provide the definition of a function
- Provide an initial value for an object
- Associate a name with a constant (enumerated type declaration)
- Declare a new type (**class**, **struct**, or **union** declaration)
- Specify a synonym for a type (**typedef** declaration)

In addition to introducing a new name, a declaration specifies how an identifier is to be interpreted by the compiler. Declarations do not automatically reserve storage associated with the identifier—reserving storage is done by definitions.

Note Most declarations are also definitions.

Syntax

declaration:
decl-specifiers$_{opt}$ *declarator-list*$_{opt}$ **;**
function-definition
linkage-specification

The declarators in *declarator-list* contain the names being declared. Although the *declarator-list* is shown as optional, it can be omitted only in declarations or definitions of a function.

Note The declaration of a function is often called a "prototype." This declaration provides type information about arguments and the function's return type that allows the compiler to perform correct conversions and to ensure type safety.

The *decl-specifiers* part of a declaration is also shown as optional; however, it can be omitted only in declarations of class types or enumerations.

Declarations occur in a scope. This controls the visibility of the name declared and the duration of the object defined (if any). For more information about how scope rules interact with declarations, see "Scope" in Chapter 2, on page 28.

An object declaration is also a definition unless it contains the **extern** storage-class specifier described in "Storage-Class Specifiers" on page 157. A function declaration is also a definition unless it is a prototype–a function header with no defining function body. An object's definition causes the storage for that object to be allocated and appropriate initializations to take place.

6.1 Specifiers

This section explains the *decl-specifiers* portion of declarations. (The syntax for declarations is given at the beginning of this chapter.)

Syntax

decl-specifier:
  storage-class-specifier
  type-specifier
  fct-specifier
  friend
  typedef

decl-specifiers:
  decl-specifiers$_{opt}$ *decl-specifier*

The *decl-specifiers* portion of a declaration is the longest sequence of *decl-specifiers* that can be construed to be a type name. The remainder of the declaration is the name or names introduced. The examples in the following list illustrate this concept:

Declaration	*decl-specifiers*	*name*
`char __far *lpszAppName;`	`char __far *`	`lpszAppName`
`typedef char __far * LPSTR;`	`char __far *`	`LPSTR`
`LPSTR _fstrcpy( LPSTR, LPSTR );`	`LPSTR`	`_fstrcpy`
`volatile void *pvvObj;`	`volatile void *`	`pvvObj`

Because **signed**, **unsigned**, **long**, and **short** all imply **int**, a **typedef** name following one of these keywords is taken to be a member of *declarator-list,* not of *decl-specifiers.*

Note Because a name can be redeclared, its interpretation is subject to the most recent declaration in the current scope. Redeclaration can affect how names are interpreted by the compiler, particularly **typedef** names.

Storage-Class Specifiers

The C++ storage-class specifiers tell the compiler about the duration and visibility of the object or function they declare, as well as where an object should be stored.

Syntax

storage-class-specifier:
auto
register
static
extern

Automatic Storage-Class Specifiers

The **auto** and **register** storage-class specifiers can be used only to declare names used in blocks or to declare formal arguments to functions. The term "auto" comes from the fact that storage for these objects is automatically allocated at run time (normally on the program's stack).

The auto Keyword

Few programmers use the **auto** keyword in declarations because all block-scoped objects not explicitly declared with another storage class are implicitly automatic. Therefore, the following two declarations are equivalent:

```
{
auto int i;    // Explicitly declared as auto.
int      j;    // Implicitly auto.
}
```

The register Keyword

The **register** keyword is similar to the **auto** keyword except that it tells the compiler to keep the object in a machine register if one is available. If no register is available, the compiler treats the object as any other automatic object.

The benefits of the **register** storage class are increased speed and reduced demands on the application stack. The latter benefit is particularly useful in recursive algorithms.

Microsoft Specific

The global register-allocation optimization (/Oe option) instructs the compiler to ignore the **register** keyword and perform register allocation based on code analysis. For algorithms that depend on register allocation, either compile without this option or use the optimize pragma to shut off global register allocation.♦

ANSI C does not allow for taking the address of a register object; this restriction does not apply to C++. However, if the address-of operator (**&**) is used on an object, the compiler must put the object in a location for which an address can be represented—in practice, this means in memory instead of in a register.

Static Storage-Class Specifiers

The static storage-class specifiers, **static** and **extern**, can be applied to objects and functions. Table 6.1 shows where the keywords **static** and **extern** can and cannot be used.

Table 6.1 Use of static and extern

Construct	Can static Be Used?	Can extern Be Used?
Function declarations within a block	No	No
Formal arguments to a function	No	No
Objects in a block	Yes	No
Objects outside a block	Yes	Yes
Functions	Yes	Yes
Class member functions	Yes	No
Class member data	Yes	No
typedef names	No	No

A name specified using the **static** keyword has internal linkage. That is, it is not visible outside the current translation unit. A name specified using the **extern** keyword has external linkage unless previously defined as having internal linkage. For more information about the visibility of names, see "Scope" on page 28 and "Program and Linkage" on page 33 in Chapter 2.

Note Functions that are declared as **inline** and that are not class member functions are given the same linkage characteristics as functions declared as **static**.

A class name whose declaration has not yet been encountered by the compiler can be used in an **extern** declaration. The name introduced with such a declaration cannot be used until the class declaration has been encountered.

Names Without Storage-Class Specifiers

File-scope names with no explicit storage-class specifiers have external linkage unless they are:

- Declared using the **const** keyword
- Previously declared with internal linkage

Function Specifiers

The **inline** and **virtual** keywords can be used as specifiers in function declarations. This use of the **virtual** keyword differs from its use in the base-class specifier of a class definition.

inline Specifier

The **inline** specifier instructs the compiler to replace function calls with the code of the function body. This substitution is "inline expansion" (sometimes called "inlining"). Inline expansion alleviates the function-call overhead at the potential cost of larger code size.

The **inline** keyword tells the compiler that inline expansion is preferred. However, the compiler can create a separate instance of the function (instantiate) and create standard calling linkages instead of inserting the code inline. Several cases where this can happen are:

- Recursive functions
- Functions that are referred to through a pointer elsewhere in the translation unit

Functions that are declared as **inline** and that are not class member functions have internal linkage unless otherwise specified.

Microsoft Specific

The **__inline** keyword is equivalent to **inline**. ♦

As with normal functions, the order of evaluation of the arguments to an inline function is not defined. In fact, it could be different than the order in which the arguments are evaluated when passed using normal function call protocol.

Microsoft Specific

Recursive functions can be substituted inline to a depth specified by the **inline_depth** pragma. After that depth, recursive function calls are treated as calls to an instance of the function. The **inline_recursion** pragma controls the inline expansion of a function currently under expansion. See Chapter 13, "Preprocessing," for more information about these pragmas. See the *Environment and Tools* manual for information about the /Ob command-line option that controls inline recursion. ♦

Inline Class Member Functions

A function defined in the body of a class declaration is an inline function. Consider the following class declaration:

```
class Account
{
public:
    Account(double initial_balance) { balance = initial_balance; }
    double GetBalance();
    double Deposit( double Amount );
    double Withdraw( double Amount );
private:
    double balance;
};
```

The `Account` constructor is an inline function. The member functions `GetBalance`, `Deposit`, and `Withdraw` are not specified as **inline** but can be implemented as inline functions using code such as the following:

```
inline double Account::GetBalance()
{
    return balance;
}

inline double Account::Deposit( double Amount )
{
    return ( balance += Amount );
}

inline double Account::Withdraw( double Amount )
{
    return ( balance -= Amount );
}
```

Note In the class declaration, the functions were declared without the **inline** keyword. The **inline** keyword can be specified in the class declaration; the result is the same.

A given inline member function must be defined exactly the same way in every compilation unit. This constraint causes inline functions to behave exactly as if they were instantiated functions. Additionally, there must be exactly one definition of an inline function.

A class member function defaults to external linkage unless a definition for that function contains the **inline** specifier. The example above shows that these functions need not be explicitly declared with the **inline** specifier; using **inline** in the function definition causes it to be an inline function. However, it is illegal to redeclare a function as **inline** after a call to that function.

Because **inline** implies **static,** it is an error to define an inline class member function with the **static** storage-class specifier.

Inline Functions versus Macros

Although inline functions are similar to macros (because the function code is expanded at the point of the call at compile time), inline functions are parsed by the compiler whereas macros are expanded by the preprocessor. As a result, there are several important differences:

- Inline functions follow all the protocols of type safety enforced on normal functions.
- Inline functions are specified using the same syntax as any other function except that they include the **inline** keyword in the function declaration.
- Expressions passed as arguments to inline functions are evaluated exactly once. In some cases, expressions passed as arguments to macros can be evaluated more than once. The following example shows a macro that converts lowercase letters to uppercase:

```
#include <stdio.h>
#include <conio.h>

#define toupper(a) ((a) >= 'a' && ((a) <= 'z') ? ((a)-('a'-'A')):(a))

int main()
{
    char ch = toupper( getch() );
    printf( "%c", ch );

    return 0;
}
```

The intent of the expression `toupper( getch() )` is that a character should be read from the console device (**stdin**) and if necessary, converted to uppercase.

Because of the implementation, **getch** is executed once to determine if the character is greater than or equal to "a" and once to determine if it is less than or equal to "z." If it is in that range, **getch** is executed yet again to convert the character to uppercase. This means the program waits for two or three characters when the programmer intended it to wait for only one.

Inline functions remedy this problem:

```
#include <stdio.h>
#include <conio.h>

inline char toupper( char a )
{
    return ((a >= 'a' && a <= 'z') ? a-('a'-'A') : a );
}

int main()
{
    char ch = toupper( getch() );
    printf( "%c", ch );

    return 0;
}
```

When to Use Inline Functions

Inline functions are best used for small functions such as those used to access private data members. The main purpose of these one- or two-line "accessor" functions is to return state information about objects; short functions are sensitive to the overhead of function calls. Longer functions spend proportionately less time in the calling/returning sequence and benefit less from inlining.

The `Point` class, introduced in "Function Call Results" in Chapter 4, on page 87 can be optimized as follows:

```
class Point
{
public:
    // Define "accessor" functions as
    //  reference types.
    unsigned& x();
    unsigned& y();
private:
    unsigned _x;
    unsigned _y;
};

inline unsigned& Point::x()
{
    return _x;
}

inline unsigned& Point::y()
{
    return _y;
}
```

Assuming coordinate manipulation is a relatively common operation in a client of such a class, specifying the two accessor functions (`x` and `y` in the preceding example) as **inline** typically saves the overhead of these procedures:

- Function calls (including parameter passing and placing the object's address on the stack)
- Preservation of caller's stack frame
- New stack-frame setup
- Return-value communication
- Old stack-frame restore
- Return

virtual Specifier

The **virtual** keyword can be applied only to nonstatic class member functions. It signifies that binding of calls to the function is deferred until run-time. Virtual functions are covered in their own section, "Virtual Functions" in Chapter 9, on page 275.

typedef Specifier

The **typedef** specifier is used to define a name that can be used as a synonym for a type or derived type. You cannot use the **typedef** specifier inside a function definition.

Syntax

typedef-name:
identifier

A **typedef** declaration introduces a name that, within its scope, becomes a synonym for the type given by the *decl-specifiers* portion of the declaration. In contrast to the **class**, **struct**, **union**, and **enum** declarations, **typedef** declarations do not introduce new types—they introduce new names for existing types.

One use of **typedef** declarations is to make declarations more uniform and compact. For example:

```
typedef char CHAR;          // Character type.
typedef CHAR * PSTR;        // Pointer to a string (char *).
typedef CHAR __far * LPSTR  // Far pointer to a string.
typedef CHAR __near * NPSTR // Near pointer to a string.
...
LPSTR _fstrchr( LPSTR source, CHAR target );
```

The names introduced by the above declarations are synonyms for the following types:

Name	Synonymous Type
CHAR	**char**
PSTR	**char ***
LPSTR	**char __far ***
NPSTR	**char __near ***

The preceding example code declares a type name, CHAR. The CHAR type name is then used to define these derived type names: PSTR (a pointer to a string), LPSTR (a far pointer to a string), and NPSTR (a near pointer to a string). Finally, the names are used in declaring the function **_fstrchr**. To see how the **typedef**s can clarify declarations, contrast the above declaration of **_fstrchr** with the following declaration:

```
char __far * _fstrchr( char __far * source, char target );
```

To use **typedef** to specify fundamental and derived types in the same declaration, you can separate declarators with commas. For example:

```
typedef char CHAR, *PSTR,
      __far *LPSTR, __near *NPSTR;
```

A particularly complicated use of **typedef** is to define a synonym for a "pointer to a function that returns type *T*." For example, a **typedef** declaration that means "pointer to a function that takes no arguments and returns type **void**," uses this code:

```
typedef void (*PVFN)();
```

The synonym can be handy in declaring arrays of functions that are to be invoked through a pointer:

```
#include <iostream.h>
#include <stdlib.h>

extern void func1();          // Declare 4 functions.
extern void func2();          // These functions are assumed to be
extern void func3();          //  defined elsewhere.
extern void func4();
                              // Declare synonym for pointer to
typedef void (*PVFN)();       //  function that takes no arguments
                              //  and returns type void.
```

```
int main( int argc, char * argv[] )
{
    // Declare an array of pointers to functions.
    PVFN pvfn1[] = { func1, func2, func3, func4 };

    // Invoke the function specified on the command line.
    if( argc > 0 && *argv[1] > '0' && *argv[1] <= '4' )
    (*pvfn1[atoi( argv[1] ) - 1])();

    return 0;
}
```

Where **void** is used in the preceding **typedef** declaration, any set of type names can be substituted. This facilitates type checking. For example, a synonym for "pointer to a function that returns type **int** and takes two arguments, both of type **const char ***" can be written as follows:

```
typedef int (*PIFN)( const char *, const char * );

// Declare a pointer to a function to point to a string
//  comparison function. At run time this can be changed
//  to point to the stricmp function if a case-insensitive
//  comparison is required.
PIFN pifnCompare = strcmp;
...
(*pifnCompare)( string1, string2 );
```

Redeclaration of typedef Names

The **typedef** declaration can be used to redeclare the same name to refer to the same type. For example:

```
// FILE1.H
typedef char CHAR;

// FILE2.H
typedef char CHAR;

// PROG.CPP
#include "file1.h"
#include "file2.h"
...
```

The program PROG.CPP includes two header files, both of which contain **typedef** declarations for the name `CHAR`. As long as both declarations refer to the same type, such redeclaration is acceptable.

A **typedef** cannot redefine a name that was previously declared as a different type. Therefore, if FILE2.H contains

```
// FILE2.H
typedef int CHAR;
```

the compiler issues an error because of the attempt to redeclare the name `CHAR` to refer to a different type. This extends to constructs such as:

```
typedef char CHAR;
typedef CHAR CHAR;      // OK: redeclared as same type

typedef union REGS      // OK: name REGS redeclared
{                       //  by typedef name with the
    struct wordregs x;  //  same meaning.
    struct byteregs h;
} REGS;
```

Use of typedef with Class Types

Use of the **typedef** specifier with class types is supported largely because of the ANSI C practice of declaring unnamed structures in **typedef** declarations. For example, many C programmers use the following coding practice:

```
typedef struct    // Declare an unnamed structure and give it the
{                 //  typedef name POINT.
    unsigned x;
    unsigned y;
} POINT;
```

The advantage of such a declaration to the C programmer is that it enables declarations like:

```
POINT ptOrigin;
```

instead of:

```
struct point_t ptOrigin;
```

In C++, the difference between **typedef** names and real types (declared with the **class**, **struct**, **union**, and **enum** keywords) is more distinct. While the C practice of declaring a nameless structure in a **typedef** statement still works, it provides no notational benefits as it does in C.

The following code is illegal in C++ because the `POINT` function is not a type constructor; hence it must return a value.

```
typedef struct
{
    POINT();          // Error: does not return a value.
    unsigned x;
    unsigned y;
} POINT;
```

The above example declares a class named `POINT` using the unnamed class **typedef** syntax. `POINT` is treated as a class name; however, the following restrictions apply to names introduced this way:

- The name (the synonym) cannot appear after a **class**, **struct**, or **union** prefix.
- The name cannot appear in the constructor names within a class declaration.
- The name cannot appear in the destructor names within a class declaration.

In summary, this syntax does not provide any mechanism for inheritance, construction, or destruction.

Name Space of typedef Names

Names declared using **typedef** occupy the same name space as other identifiers (except statement labels). Therefore, they cannot use the same identifier as a previously declared name, except in the case of a class-type declaration. Consider the following example:

```
typedef unsigned long UL;   // Declare a typedef name, UL.
int UL;                     // Error: redefined.
```

The name-hiding rules that pertain to other identifiers also govern the visibility of names declared using **typedef**. Therefore, the following example is legal in C++:

```
typedef unsigned long UL;   // Declare a typedef name, UL.
...
long Beep
{
    unsigned int UL;        // Redeclaration hides typedef name.
    ...
}
// typedef name "unhidden" here.
```

friend Specifier

The **friend** specifier is used to designate functions or classes that have the same access privileges as class member functions. Friend functions and classes are covered in detail in "Friends" in Chapter 10, on page 290.

Type Specifiers

Type specifiers determine the type of the name being declared.

Syntax

type-specifier:
simple-type-name
class-specifier
enum-specifier
elaborated-type-specifier
:: *class-name*
const
volatile

This section discusses simple type names, elaborated type specifiers, and nested type names.

Simple Type Names

A simple type name is the name of a complete type.

Syntax

simple-type-name:
complete-class-name
qualified-type-name
char
short
int
long
signed
unsigned
float
double
void

Table 6.2 shows how the simple type names can be used together.

A name specified using the **const** keyword has internal linkage unless it is specifically given external linkage.

Table 6.2 Type Name Combinations

Type	Can Appear With	Comments
int	**long** or **short**, but not both	Type **int** can be used alone, in which case it means **short int** in 16-bit compilations and **long int** in 32-bit compilations.
long	int or **double**	Type **long** implies type **long int**.
short	**int**	Type **short** implies type **short int**.
signed	**char**, **short**, **int**, or **long**	Type **signed** implies **signed int**. The most-significant bit of objects of type **signed char** and bit fields of signed integral types is taken to be the sign bit.
unsigned	**char**, **short**, **int**, or **long**	Type **unsigned** implies **unsigned int**. The most-significant bit of objects of type **unsigned char** and bit fields of unsigned integral types is not treated as the sign bit.

Elaborated Type Specifiers

Elaborated type specifiers are used to declare user-defined types. These can be either class- or enumerated-types.

Syntax

elaborated-type-specifier:
  class-key class-name
  class-key identifier
  enum *enum-name*

class-key:
  class
  struct
  union

If *identifier* is specified, it is taken to be a class name. For example:

```
class Window;
```

The above statement declares the `Window` identifier as a class name. This syntax is used for forward declaration of classes. For more information about class names, see "Class Names" in Chapter 8, on page 232.

If a name is declared using the **union** keyword, it must be defined using the **union** keyword as well. Names that are defined using the **class** keyword can be declared using the **struct** keyword (and vice versa). Therefore, the following code samples are legal:

```
// Legal example 1
struct A;     // Forward declaration of A.

class A       // Define A.
{
public:
    int i;
};

// Legal example 2
class A;      // Forward declaration of A.

struct A      // Define A.
{
private:
    int i;
};

// Legal example 3
union A;      // Forward declaration of A.

union A       // Define A.
{
    int  i;
    char ch[2];
};
```

The next set of examples, however, are illegal:

```
// Illegal example 1
union A;      // Forward declaration of A.

struct A      // Define A.
{
    int i;
};
```

```
// Illegal example 2
union A;      // Forward declaration of A.

class A      // Define A.
{
public:
    int i;
};

// Illegal example 3
struct A;     // Forward declaration of A.

union A       // Define A.
{
    int  i;
    char ch[2];
};
```

Nested Type Names

Microsoft C++ supports declaration of nested types—both named and anonymous.

Syntax

qualified-type-name:
  typedef-name
  class-name **::** *qualified-type-name*

complete-class-name:
  qualified-class-name
  :: *qualified-class-name*

qualified-class-name:
  class-name
  class-name **::** *qualified-class-name*

In some programming situations, it makes sense to define nested types. These types are visible only to member functions of the class type in which they are defined. They can also be made visible by constructing a qualified type name using the scope-resolution operator (**::**).

Note One commonly used class hierarchy that employs nested types is iostream. In the iostream header files, the definition of class **ios** includes a series of enumerated types, which are packaged for use only with the iostream library.

The following is an example of how to define nested classes:

```
class WinSystem
{
public:
    class Window
    {
    public:
        Window();          // Default constructor.
        ~Window();         // Destructor.
        int NumberOf();    // Number of objects of class.
        int Count();       // Count number of objects of class.
    private:
        static int _Count;
    };
    class CommPort
    {
    public:
        CommPort();        // Default constructor.
        ~CommPort();       // Destructor.
        int NumberOf();    // Number of objects of class.
        int Count();       // Count number of objects of class.
    private:
        static int _Count;
    };
};

// Initialize WinSystem static members.
int WinSystem::Window::_Count = 0;
int WinSystem::CommPort::_Count = 0;
```

To access a name defined in a nested class, use the scope-resolution operator (**::**) to construct a complete class name. Use of this operator is shown in the initializations of the **static** members in the example above. To use a nested class in your program, use code such as the following:

```
WinSystem::Window Desktop;
WinSystem::Window AppWindow;

cout << "Number of active windows: " << Desktop.Count() << "\n";
```

Nested anonymous classes or structures can be defined as follows:

```
class Ledger
{
    class
    {
    public:
        double   PayableAmt;
        unsigned PayableDays;
    } Payables;

    class
    {
    public:
        double   RecvableAmt;
        unsigned RecvableDays;
    } Receivables;
};
```

An anonymous class must be an aggregate, which has no member functions and no static members.

Note While an enumerated type can be defined inside a class declaration, the reverse is not true; class types cannot be defined inside enumeration declarations.

6.2 Enumeration Declarations

An enumeration is a distinct integral type that defines named constants. Enumerations are declared using the **enum** keyword.

Syntax

enum-name:
  identifier

enum-specifier:
  enum *identifier*$_{opt}$ { *enum-list*$_{opt}$ }

enum-list:
  enumerator
  enum-list , *enumerator*

enumerator:
  identifier
  identifier = *constant-expression*

Enumerated types are valuable when an object can assume a known and reasonably limited set of values. Consider the example of the suits from a deck of cards:

```
class Card
{
public:
    enum Suit
    {
        Diamonds,
        Hearts,
        Clubs,
        Spades
    };
    // Declare two constructors: a default constructor,
    //  and a constructor that sets the cardinal and
    //  suit value of the new card.
    Card();
    Card( int CardInit, Suit SuitInit );

    // Get and Set functions.
    int   GetCardinal();     // Get cardinal value of card.
    int   SetCardinal();     // Set cardinal value of card.
    Suit  GetSuit();         // Get suit of card.
    Suit  SetSuit();         // Set suit of card.
    char *NameOf();          // Get string representation of card.
private:
    Suit  suit;
    int   cardinalValue;
};

// Define a postfix increment operator for Suit.
inline Card::Suit operator++( Card::Suit &rs, int )
{
    return (Card::Suit)(rs + 1);
}
```

The preceding example defines a class, `Card`, that contains a nested enumerated type, `Suit`. To create a pack of cards in a program, use code such as the following:

```
Card *Deck[52];
int   j = 0;

for( Card::Suit curSuit = Card::Diamonds; curSuit <= Card::Spades;
     curSuit++ )
    for( int i = 1; i <= 13; ++i )
        Deck[j++] = new Card( i, curSuit );
```

In the preceding example, the type `Suit` is nested; therefore, the class name (`Card`) must be used explicitly in public references. However, in member functions, the class name can be omitted.

In the first segment of code, the postfix increment operator for `Suit` is defined. Without a user-defined increment operator, `curSuit` could not be incremented. For more information about user-defined operators, see "Overloaded Operators" in Chapter 12, on page 351.

Consider the code for the `NameOf` member function (a better implementation is presented later):

```
char* Card::NameOf() // Get the name of a card.
{
    static char szName[20];
    static char *Numbers[] =
    { "1", "2", "3", "4", "5", "6", "7", "8", "9",
      "10", "Jack", "Queen", "King"
    };
    static char *Suits[] =
    { "Diamonds", "Hearts", "Clubs", "Spades" };

    if( GetCardinal() < 13)
        strcpy( szName, Numbers[GetCardinal()] );

    strcat( szName, " of " );

    switch( GetSuit() )
    {
    // Diamonds, Hearts, Clubs, and Spades do not need explicit
    //  class qualifier.
    case Diamonds: strcat( szName, "Diamonds" ); break;
    case Hearts:   strcat( szName, "Hearts" );   break;
    case Clubs:    strcat( szName, "Clubs" );    break;
    case Spades:   strcat( szName, "Spades" );   break;
    }

    return szName;
}
```

An enumerated type is an integral type. The identifiers introduced with the **enum** declaration can be used wherever constants appear. Normally, the first identifier's value is 0 (`Diamonds`, in the example above), and the values increase by one for each succeeding identifier. Therefore, the value of `Spades` is 3.

Any enumerator in the list, including the first one, can be initialized to a value other than its default value. Suppose the declaration of `Suit` had been the following:

```
enum Suit
{
    Diamonds = 5,
    Hearts,
    Clubs = 4,
    Spades
};
```

Then the values of `Diamonds`, `Hearts`, `Clubs`, and `Spades` would have been 5, 6, 4, and 5, respectively. Note that 5 is used more than once.

The default values for these enumerators make implementation of the `NameOf` function simpler:

```
char* Card::NameOf() // Get the name of a card.
{
    static char szName[20];
    static char *Numbers[] =
    { "1", "2", "3", "4", "5", "6", "7", "8", "9",
      "10", "Jack", "Queen", "King"
    };
    static char *Suits[] =
    { "Diamonds", "Hearts", "Clubs", "Spades" };

    if( GetCardinal() < 13)
        strcpy( szName, Numbers[GetCardinal()] );

    strcat( szName, " of " );

    strcat( szName, Suits[GetSuit()] );

    return szName;
}
```

The accessor function `GetSuit` returns type `Suit`, an enumerated type. Because enumerated types are integral types, they can be used as arguments to the array subscript operator. (For more information, see "Subscript Operator" in Chapter 4, on page 81.)

Enumerator Names

The names of enumerators must be different from any other enumerator or variable in the same scope. However, the values (as shown above) can be duplicated.

Definition of Enumerator Constants

Enumerators are considered defined immediately after their initializers; therefore, they can be used to initialize succeeding enumerators. The following example defines an enumerated type that ensures that any two enumerators can be combined with the OR operator:

```
enum FileOpenFlags
{
    OpenReadOnly  = 1,
    OpenReadWrite = OpenReadOnly  << 1,
    OpenBinary    = OpenReadWrite << 1,
    OpenText      = OpenBinary    << 1,
    OpenShareable = OpenText      << 1
};
```

In the preceding example, the preceding enumerator is used to initialize each succeeding enumerator.

Conversions and Enumerated Types

Because enumerated types are integral types, any enumerator can be converted to another integral type by integral promotion. Consider this example:

```
enum Days
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};

int  i;
Days d = Thursday;

i = d;    // Converted by integral promotion.
cout << "i = " << i << "\n";
```

However, there is no implicit conversion from any integral type to an enumerated type. Therefore, (continuing with the preceding example) the following statement is in error:

```
d = 6;    // Erroneous attempt to set d to Saturday.
```

Assignments such as the one above, where no implicit conversion exists, must use a cast to perform the conversion:

```
d = (Days)6;    // Explicit cast-style conversion to type Days.
d = Days( 4 );  // Explicit function-style conversion to type Days.
```

The example above shows conversions of values that coincide with the enumerators. There is no mechanism that protects you from converting a value that does not coincide with one of the enumerators. For example:

```
d = Days( 967 );
```

Some such conversions may work. However, there is no guarantee the resultant value will be one of the enumerators. Additionally, if the size of the enumerator is too small to hold the value being converted, the value stored may not be what you expect.

6.3 Linkage Specifications

The term "linkage specification" refers to the protocol for linking functions (or procedures) written in different languages. The following calling conventions are affected:

- Case sensitivity of names.
- Decoration of names. In C, the compiler prefixes names with an underscore. This is often called "decoration." In C++, name decoration is used to retain type information through the linkage phase. (For more information, see Appendix A, "Phases of Translation.")
- Order in which arguments are expected on the stack.
- Responsibility for adjusting the stack on function return. Either the called function or the calling function is responsible.
- Passing of hidden arguments (whether any hidden arguments are passed).

Microsoft Specific The calling conventions for Microsoft C and C++ are shown in Table 6.3.

Table 6.3 C and C++ Calling Conventions

Calling Convention	C++	C
Case sensitivity	Case sensitive	Case sensitive
Order of arguments	Left argument pushed first; **this** passed last (class-member functions only)	Right argument pushed first
Stack responsibility	Called function	Calling function
Hidden arguments	For member functions, the **this** pointer is passed as a hidden argument	Used only for structure and floating return types
Naming	C++ type-safe naming	C naming; all names prefixed with an underscore (_) ♦

Syntax

linkage-specification:
extern *string-literal* { *declaration-list*$_{opt}$ }
extern *string-literal declaration*

declaration-list:
declaration
declaration-list

Linkage specification facilitates gradually porting C code to C++ by allowing the use of existing code.

Microsoft Specific

The only linkage specifications currently supported by Microsoft C++ are **"C"** and **"C++"**. ♦

The following example declares the functions `atoi` and `atol` with C linkage:

```
extern "C"
{
    int  atoi( char *string );
    long atol( char *string );
}
```

Calls to these functions are made using C linkage. The same result could be achieved with these two declarations:

```
extern "C" int  atoi( char *string );
extern "C" long atol( char *string );
```

Microsoft Specific

All Microsoft C standard include files use conditional compilation directives to detect C++ compilation. When a C++ compilation is detected, the prototypes are enclosed in an **extern "C"** directive as follows:

```
// Sample.h
#if defined(__cplusplus)
extern "C"
{
#endif

// Function declarations

#if defined(__cplusplus)
}
#endif♦
```

There is no need for you to declare the functions in the standard include files as **extern "C"**.

If a function is overloaded, no more than one of the functions of the same name can have a linkage specifier. (For more information, see "Function Overloading" in Chapter 7, on page 205.)

Table 6.4 shows how various linkage specifications work.

Table 6.4 Effects of Linkage Specifications

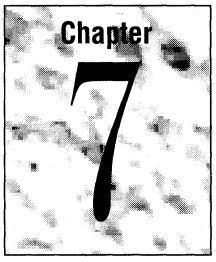
Specification	Effect
On an object	Affects linkage of that object only
On a function	Affects linkage of that function and all functions or objects declared within it
On a class	Affects linkage of all nonmember functions and objects declared within the class

If a function has more than one linkage specification, they must agree; it is an error to declare functions as having both C and C++ linkage. Furthermore, if two declarations for a function occur in a program–one with a linkage specification and one without–the declaration with the linkage specification must be first. Any redundant declarations of functions that already have linkage specification are given the linkage specified in the first declaration. For example:

```
extern "C" int CFunc1();
...
int CFunc1();              // Redeclaration is benign; C linkage is
                           //  retained.

int CFunc2();
...
extern "C" int CFunc2(); // Error: not the first declaration of
                           //  CFunc2;  cannot contain linkage
                           //  specifier.
```

Functions and objects explicitly declared as **static** within the body of a compound linkage specifier ({ }) are treated as static functions or objects; the linkage specifier is ignored. Other functions and objects behave as if declared using the **extern** keyword. (See "Storage-Class Specifiers" on page 157 for details about the **extern** keyword.)

Declarators



A "declarator" is the part of a declaration that names an object, type, or function. Declarators appear in a declaration as one or more names separated by commas; each name can have an associated initializer.

Syntax

declarator-list:
  init-declarator
  declarator-list , *init-declarator*

init-declarator:
  declarator initializer$_{opt}$

7.1 Overview

Declarators are the components of a declaration that specify names. Declarators can also modify basic type information to cause names to be functions or pointers to objects or functions. (Specifiers, discussed in Chapter 6, convey properties such as type and storage class. Modifiers, discussed in this chapter and in Appendix B, modify declarators.) Figure 7.1 shows a complete declaration of two names, `szBuf` and `strcpy`, and calls out the components of the declaration.

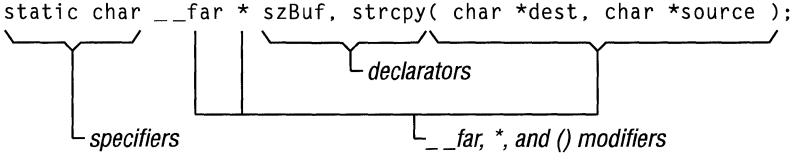


Figure 7.1 Specifiers, Modifiers, and Declarators

Microsoft Specific

Most Microsoft extended keywords can be used as modifiers to form derived types; they are not specifiers or declarators. (See Appendix B, "Microsoft-Specific Modifiers.")♦

Syntax

declarator:
  dname
  ptr-operator declarator
  declarator (*argument-declaration-list*) *cv-mod-list*
  declarator [*constant-expression*$_{opt}$]
  (*declarator*)

ptr-operator:
  * *cv-qualifier-list*$_{opt}$
  & *cv-qualifier-list*$_{opt}$
  complete-class-name :: * *cv-qualifier-list*$_{opt}$

cv-qualifier-list:
  cv-qualifier cv-qualifier-list$_{opt}$

cv-qualifier:
  const
  volatile

cv-mod-list:
  cv-qualifier cv-mod-list$_{opt}$
  pmodel cv-mod-list$_{opt}$

pmodel: one of
  __near
  __far
  __huge

dname:
  name
  class-name
  ~ *class-name*
  typedef-name
  qualified-type-name

Declarators appear in the declaration syntax after an optional list of specifiers (*decl-specifiers*). These specifiers are discussed in Chapter 6, "Declarations." A declaration can contain more than one declarator, but each declarator declares only one name. The following sample declaration shows how specifiers and declarators are combined to form a complete declaration:

```
const char *pch, ch;
```

In the above declaration, the keywords **const** and **char** make up the list of specifiers. Two declarators are listed: `*pch`, and `ch`. The simplified syntax of a declaration, then, is the following, where `const char` is the type, and `*pch` and `ch` are the identifiers:

type identifier$_1$[[, *identifier*$_2$[[...,*identifier*$_n$]]]] ;

When the binding of elements in a complicated declarator does not yield the desired result, you can use parentheses for clarification. A better technique, however, is to use a **typedef** or a combination of parentheses and **typedef**s. Consider declaring an array of pointers to functions. Each function must obey the same protocol so that the arguments and return values are known:

```
//  Function returning type int that takes one
//   argument of type char *.
typedef int (*PIFN)( char * );

//  Declare an array of 7 pointers to functions
//   returning int and taking one argument of type
//   char *.
PIFN pifnDispatchArray[7];
```

The equivalent declaration can be written without **typedef**s, but it is so complicated that the potential for error exceeds any benefits:

```
int ( *pifnDispatchArray[7] )( char * );
```

7.2 Type Names

Type names are used in some declarators in the following ways:

- In explicit conversions
- As arguments to the **sizeof** operator
- As arguments to the **new** operator
- In function prototypes
- In **typedef** statements

A type name consists of type specifiers, covered in Chapter 6, and abstract declarators, discussed in "Abstract Declarators" on page 187.

In the following example, the arguments to the function **strcpy** are supplied using their type names. In the case of the `source` argument, **const** is the specifier and `char *` is the abstract declarator:

```
static char __far *szBuf, *strcpy( char *dest, const char *source );
```

Syntax

type-name:
type-specifier-list *abstract-declarator*$_{opt}$

type-specifier-list:
type-specifier type-specifier-list $_{opt}$

abstract-declarator:
ptr-operator *abstract-declarator*$_{opt}$
abstract-declarator$_{opt}$ (*argument-declaration-list*) *cv-qualifier-list*$_{opt}$
abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
(*abstract-declarator*)

An abstract declarator is a declarator that does not declare a name—the identifier is left out. For example:

```
char *
```

declares the type "pointer to type **char**." This abstract declarator can be used in a function prototype as follows:

```
char *strcmp( char *, char * );
```

In the above prototype (declaration), the function's arguments are specified as abstract declarators. The following is a more complicated abstract declarator that declares the type "pointer to a function that takes two arguments, both of type **char ***," and returns type **char ***:

```
char * (*)( char *, char * )
```

Since abstract declarators completely declare a type, it is legal to form expressions of the form:

```
// Get the size of array of 10 pointers to type char.
size_t nSize = sizeof( char *[10] );

// Allocate a pointer to a function that has no
//  return value and takes no arguments.
typedef void (PVFN *)();
PVFN *pvfn = new PVFN;

// Allocate an array of pointers to functions that
//  return type WinStatus, and take one argument of
//  type WinHandle.
typedef WinStatus (PWSWHFN *)( WinHandle );
PWSWHFN pwswhfnArray[] = new PWSWHFN[10];
```

Ambiguity Resolution

To perform explicit conversions from one type to another, you must use casts, specifying the desired type name. Some type casts result in syntactic ambiguity. The following function-style type cast is ambiguous:

```
char *aName( String( s ) );
```

It is unclear whether it is a function declaration or an object declaration with a function-style cast as the initializer: Does the above statement declare a function returning type **char** * that takes one argument of type `String`, or does it declare the object `aName` and initialize it with the value of `s` cast to type `String`?

The rule in this case is that if a syntactic construct can be considered a valid function declaration, it is treated as such. Only if it cannot possibly be a function declaration—that is, if it would be syntactically incorrect—is a statement examined to see if it is a function-style type cast. Therefore, the compiler considers the statement to be a declaration of a function and treats the parentheses around the identifier `s` as gratuitous. On the other hand, the statements:

```
char *aName( (String)s );
```

and

```
char *aName = String( s );
```

are clearly declarations of objects, and a user-defined conversion from type `String` to type **char** * is invoked to perform the initialization of `aName`.

7.3 Abstract Declarators

An abstract declarator is a declarator in which the identifier is omitted. (For related information, see "Type Names" on page 185.)

The following abstract declarators are discussed in this section:

- Pointers
- References
- Pointers to members
- Arrays
- Functions
- Default arguments

Pointers

Pointers are declared using the declarator syntax:

* *cv-qualifier-list*$_{opt}$ *dname*

A pointer is a 16- or 32-bit quantity that holds the address of an object. The full declaration, then is:

decl-specifiers * *cv-qualifier-list*$_{opt}$ *dname* ;

A simple example of such a declaration is:

```
char *pch;
```

The preceding declaration specifies that `pch` points to an object of type **char**.

const and volatile Pointers

The **const** and **volatile** keywords change how pointers are treated. The **const** keyword specifies that the value associated with the name that follows can be set only at program startup; the data is protected from modification thereafter.

The **volatile** keyword specifies that the value associated with the name that follows can be modified by actions other than those in the user application. Therefore, the **volatile** keyword is useful for declaring objects in shared memory that can be accessed by multiple processes or global data areas used for communication with interrupt service routines.

When a name is declared as **volatile**, the compiler reloads the value from memory each time it is accessed by the program. This dramatically reduces the possible optimizations. However, when the state of an object can change unexpectedly, it is the only way to ensure predictable program performance.

To declare the object pointed to by the pointer as **const** or **volatile**, use a declaration of the form:

```
const    char *cpch;
volatile char *vpch;
```

To declare the value of the pointer—that is, the actual address stored in the pointer—as **const** or **volatile**, use a declaration of the form:

```
char * const    pchc;
char * volatile pchv;
```

By extension, both the pointer and the object can be declared as **const** or **volatile** using the declaration:

```
const char ch = 'A';

// Object and pointer const.
const    char * const cpchc = &ch;
const    char * volatile cpchv;  // Pointer volatile; object const.
volatile char * const vpchc;     // Pointer const; object volatile.
volatile char * volatile cpchc;  // Object and pointer volatile.
```

The C++ language prevents assignments that would allow an object or pointer declared as **const** to be modified. Such assignments would remove the information that the object or pointer was declared with, thereby violating the intent of the original declaration. Consider the following declarations:

```
const char cch = 'A';
char        ch = 'B';
```

Given the above declarations of two objects (`cch`, of type **const char**, and `ch` of type **char**), the following declaration/initializations are valid:

```
const char        *pch1 = &cch;
const char *const  pch4 = &cch;
const char        *pch5 = &ch;
char              *pch6 = &ch;
char       *const  pch7 = &ch;
const char *const  pch8 = &ch;
```

The following declaration/initializations are erroneous.

```
char *pch2 = &cch;
char *const pch3 = &cch;
```

The declaration of `pch2` declares a pointer through which a constant object might be modified and is therefore disallowed. The declaration of `pch3` specifies that the *pointer* is constant, not the object; the declaration is disallowed for the same reason the `pch2` declaration is disallowed.

The following eight assignments show assigning through pointer and changing of pointer value for the above declarations; for the purposes of this discussion, assume that the initialization had been correct for `pch1` through `pch8`.

```
*pch1 = 'A';  // Error: object declared const
pch1 = &ch;   // OK: pointer not declared const
*pch2 = 'A';  // OK: normal pointer
pch2 = &ch;   // OK: normal pointer
*pch3 = 'A';  // OK: object not declared const
pch3 = &ch;   // Error: pointer declared const
*pch4 = 'A';  // Error: object declared const
pch4 = &ch;   // Error: pointer declared const
```

Pointers declared as **volatile** or as a mixture of **const** and **volatile** obey the same rules.

Pointers declared as **const** are often used in function declarations as follows:

```
char *strcpy( char *szTarget, const char *szSource );
```

The above statement declares a function, **strcpy**, that takes two arguments of type "pointer to **char**" and returns a pointer to type **char**. Because the arguments are passed by reference and not by value, the function would be free to modify both `szTarget` and `szSource` if `szSource` were not declared as **const**. The declaration of `szSource` as **const** assures the caller that `szSource` cannot be changed by the called function.

Note Because there is a standard conversion from *typename* * to **const** *typename* *, it is legal to pass an argument of type **char** * to **strcpy**. However, the reverse is not true; no implicit conversion exists to remove the **const** attribute from an object or pointer.

References

References are declared using the declarator syntax:

Syntax

& *cv-qualifier-list*$_{opt}$ *dname*

A reference is a 16- or 32-bit quantity that holds the address of an object but behaves syntactically like an object. A reference declaration consists of an (optional) list of specifiers followed by a reference declarator.

Syntax

decl-specifiers & *cv-qualifier-list*$_{opt}$ *dname* ;

Consider the user-defined type `Date`:

```
struct Date
{
    short DayOfWeek;
    short Month;
    short Day;
    short Year;
};
```

The following statements declare an object of type `Date` and a reference to that object:

```
Date  Today;             // Declare the object.
Date& TodayRef = Today;  // Declare the reference.
```

Both the name of the object, `Today`, and the reference to the object, `TodayRef`, can be used identically in programs:

```
Today.DayOfWeek = 3;    // Tuesday
TodayRef.Month  = 7;    // July
```

Reference-Type Function Arguments

Often, when passing large objects to functions, it is more efficient to pass references to these objects instead. This allows the compiler to pass the address of the object while maintaining the syntax that would have been used to access the object. Consider the following example that uses the `Date` structure:

```
// Create a Julian date of the form DDDYYYY
// from a Gregorian date.
long JulianFromGregorian( Date& GDate )
{
    static int cDaysInMonth[] = {
    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };
    long JDate;

    // Add in days for months already elapsed.
    for( int i = 0; i < GDate.Month - 1; ++i )
        JDate += cDaysInMonth[i];

    // Add in days for this month.
    JDate += GDate.Day;

    // Check for leap year.
    if( GDate.Year % 100 != 0 && GDate.Year % 4 != 0 )
        JDate++;

    // Add in year.
    JDate *= 10000;
    JDate += GDate.Year;

    return JDate;
}
```

The above code illustrates the fact that members of a structure passed by reference are accessed using the member-selection operator (.) instead of the pointer member-selection operator (->).

Although arguments passed as reference types observe the syntax of nonpointer types, they retain one important characteristic of pointer types: they are modifiable unless declared as **const**. Because the intent of the above code is not to modify the object `GDate`, a more appropriate function prototype is:

```
long JulianFromGregorian( const Date& GDate );
```

This prototype guarantees that the function `JulianFromGregorian` will not change its argument.

Note that any function prototyped as taking a reference type can accept an object of the same type in its place because there is a standard conversion from *typename* to *typename&*.

Reference-Type Function Returns

Functions can be declared to return a reference type. There are two reasons to make such a declaration:

- The information being returned is a large enough object that returning a reference is more efficient than returning a copy.
- The type of the function must be an l-value.

Just as it can be more efficient to pass large objects *to* functions by reference, it also can be more efficient to return large objects *from* functions by reference. Reference return protocol eliminates the necessity of copying the object to a temporary location prior to returning.

Another case in which reference return types can be useful is when the function must evaluate to an l-value. Most overloaded operators fall into this category, particularly the assignment operator. Overloaded operators are covered in "Overloaded Operators" in Chapter 12, on page 351. Consider the `Point` example from Chapter 4:

```
class Point
{
public:
    // Define "accessor" functions as
    //  reference types.
    unsigned& x();
    unsigned& y();
private:
    unsigned obj_x;
    unsigned obj_y;
};

unsigned& Point :: x()
{
    return obj_x;
}
```

```
unsigned& Point :: y()
{
    return obj_y;
}

void main()
{
    Point ThePoint;

    // Use x() and y() as l-values.
    ThePoint.x() = 7;
    ThePoint.y() = 9;

    // Use x() and y() as r-values.
    cout << "x = " << ThePoint.x() << "\n"
         << "y = " << ThePoint.y() << "\n";
}
```

Notice that the functions `x` and `y` are declared as returning reference types. These functions can be used on either side of an assignment statement.

Declarations of reference types must contain initializers except in the following cases:

- Explicit **extern** declaration
- Declaration of a class member
- Declaration within a class
- Declaration of an argument to a function or the return type for a function

References to Pointers

References to pointers can be declared in much the same way as references to objects. Declaring a reference to a pointer yields a modifiable value that is used like a normal pointer. The following code samples illustrate the difference between using a pointer to a pointer and a reference to a pointer:

```
#include <iostream.h>
#include <string.h>

// Define a binary tree structure.
struct BTree
{
    char  *szText;
    BTree *Left;
    BTree *Right;
};
```

```
// Define a pointer to the root of the tree.
BTree *btRoot = 0;

int Add1( BTree **Root, char *szToAdd );
int Add2( BTree*& Root, char *szToAdd );
void PrintTree( BTree* btRoot );

main( int argc, char *argv[] )
{
    if( argc < 2 )
    {
        cerr << "Usage: Refptr [1 | 2]" << "\n";
        cerr << "\n\twhere:\n";
        cerr << "\t1 uses double indirection\n";
        cerr << "\t2 uses a reference to a pointer.\n";
        cerr << "\n\tInput is from stdin.\n";
        return 1;
    }

    char *szBuf = new char[132];

    // Read a text file from the standard input device and
    //  build a binary tree.
    while( !cin.eof() )
    {
        cin.get( szBuf, 132, '\n' );
        cin.get();
        if( strlen( szBuf ) )
            switch( *argv[1] )
            {
            // Method 1: Use double indirection.
            case '1':
                Add1( &btRoot, szBuf );
                break;

            // Method 2: Use reference to a pointer.
            case '2':
                Add2( btRoot, szBuf );
                break;

            default:
                cerr << "Illegal value '" << *argv[1]
                     << "' supplied for add method.\n"
                     << "Choose 1 or 2.\n";
                return -1;
            }
    }
```

```
    // Display the sorted list.
    PrintTree( btRoot );

    return 0;
}
// PrintTree: Display the binary tree in order.
void PrintTree( BTree* btRoot )
{
    // Traverse the left branch of the tree recursively.
    if( btRoot->Left )
        PrintTree( btRoot->Left );

    // Print the current node.
    cout << btRoot->szText << "\n";

    // Traverse the right branch of the tree recursively.
    if( btRoot->Right )
        PrintTree( btRoot->Right );
}

// Add1: Add a node to the binary tree.
//       Uses double indirection.
int Add1( BTree **Root, char *szToAdd )
{
    if( (*Root) == 0 )
    {
        (*Root) = new BTree;
        (*Root)->Left = 0;
        (*Root)->Right = 0;
        (*Root)->szText = new char[strlen( szToAdd ) + 1];
        strcpy( (*Root)->szText, szToAdd );
        return 1;
    }
    else if( strcmp( (*Root)->szText, szToAdd ) > 0 )
        return Add1( &((*Root)->Left), szToAdd );
    else
        return Add1( &((*Root)->Right), szToAdd );
}

// Add2: Add a node to the binary tree.
//       Uses reference to pointer
```

```
int Add2( BTree*& Root, char *szToAdd )
{
    if( Root == 0 )
    {
        Root = new BTree;
        Root->Left = 0;
        Root->Right = 0;
        Root->szText = new char[strlen( szToAdd ) + 1];
        strcpy( Root->szText, szToAdd );
        return 1;
    }
    else if( strcmp( Root->szText, szToAdd ) > 0 )
        return Add2( Root->Left, szToAdd );
    else
        return Add2( Root->Right, szToAdd );
}
```

In the preceding program, functions `Add1` and `Add2` are functionally equivalent (although they are not called the same way). The difference is that `Add1` uses double indirection whereas `Add2` uses the convenience of a reference to a pointer.

Pointers to Members

Declarations of pointers to members are special cases of pointer declarations.

Syntax

decl-specifiers class-name :: * *cv-qualifier-list*$_{opt}$ *dname* ;

A pointer to a member of a class differs from a normal pointer because it has type information for not only the type of the member, but also for the class to which the member belongs. The following example declares a class, `Window`, then some pointers to member data.

```
class Window
{
public:
    Window();                                  // Default constructor.
    Window( int x1, int y1,                    // Constructor specifying
        int x2, int y2 );                      //  window size.
    BOOL SetCaption( const char *szTitle );    // Set window caption.
    const char *GetCaption();                  // Get window caption.
    char *szWinCaption;                        // Window caption.
};

// Declare a pointer to the data member szWinCaption.
char * Window::* pwCaption = &Window::szWinCaption;
```

In the preceding example, `pwCaption` is a pointer of type **char *** that is a member of class `Window`. The next code fragment declares pointers to the `SetCaption` and `GetCaption` member functions.

```
const char * (Window::*pfnwGC)() = &Window::GetCaption;
BOOL (Window::*pfnwSC)( const char * ) = &Window::SetCaption;
```

The pointers `pfnwGC` and `pfnwSC` point to `GetCaption` and `SetCaption` of the `Window` class, respectively. To copy information to the window caption directly using the pointer to member `pwCaption`, use code such as this:

```
Window wMainWindow;
Window *pwChildWindow = new Window;
char   *szUntitled    = "Untitled -  ";
int    cUntitledLen   = strlen( szUntitled );

strcpy( wMainWindow.*pwCaption, szUntitled );
(wMainWindow.*pwCaption)[cUntitledLen - 1] = '1';

strcpy( pwChildWindow->*pwCaption, szUntitled );
(pwChildWindow->*pwCaption)[szUntitledLen - 1] = '2';
```

The difference between the **.*** and **->*** operators (the pointer-to-member operators) is that the **.*** operator is used to select members of an object, while the **->*** operator is used to select members through a pointer. (For more about these operators, see "Expressions with Pointer-to-Member Operators" in Chapter 4, on page 124.)

Note that the result of the pointer-to-member operators is the type of the member—in this case, **char ***.

The following code fragment invokes the member functions `GetCaption` and `SetCaption` using pointers to members:

```
// Allocate a buffer.
char szCaptionBase[100];

// Copy the main window caption into the buffer
//  and append " [View 1]".
strcpy( szCaptionBase, (wMainWindow.*pfnwGC)() );
strcat( szCaptionBase, " [View 1]" );

// Set the child window's caption.
(pwChildWindow->*pfnwSC)( szCaptionBase );
```

Restrictions on Pointers to Members

It is illegal to declare pointers to static class members. Because only one instance of a static member exists for all objects of a given class, the ordinary address-of (**&**) and dereference (*****) operators can be used.

Pointers to Members and Virtual Functions

Invoking a virtual function through a pointer-to-member function works exactly as if the function had been called directly: the correct function is looked up in the v-table and invoked. The following code shows how this is done:

```
class Base
{
public:
    virtual void Print();
};
void (Base ::* bfnPrint)() = &Base :: Print;

void Base :: Print()
{
    cout << "Print function for class 'Base'\n";
}

class Derived : public Base
{
public:
    void Print();  // Print is still a virtual function.
};

void Derived :: Print()
{
    cout << "Print function for class 'Derived'\n";
}

main()
{
    Base   *bPtr;
    Base    bObject;
    Derived dObject;

    bPtr = &bObject;    // Set pointer to address of bObject.
    (bPtr->*bfnPrint)();

    bPtr = &dObject;    // Set pointer to address of dObject.
    (bPtr->*bfnPrint)();

    return 0;
}
```

The output from this program is:

```
Print function for class 'Base'
Print function for class 'Derived'
```

The key to virtual functions working, as always, is invoking them through a pointer to a base class. (For more information about virtual functions, see "Virtual Functions" in Chapter 9, on page 275.)

Arrays

Arrays are collections of like objects. The simplest case of an array is a vector. C++ provides a convenient syntax for declaration of fixed-size arrays:

Syntax

decl-specifiers dname [*constant-expression*$_{opt}$] ;

The number of elements in the array is given by the *constant-expression.* The first element in the array is the 0th element, and the last element is the (*n*-1th) element, where *n* is the size of the array. The *constant-expression* must be of an integral type and must be greater than 0.

Arrays are derived types and can therefore be constructed from the following:

- Any user-defined or built-in type except **void**
- Pointers to data or functions
- Pointers to members
- Enumerated types
- Other arrays, except for arrays of references

Arrays constructed from other arrays are multidimensional arrays. These multidimensional arrays are specified by placing multiple [*constant-expression*] specifications in sequence. For example, consider this declaration:

```
int i2[5][7];
```

It specifies an array of type **int**, conceptually arranged in a two-dimensional matrix of five rows and seven columns, as shown in Figure 7.2.

0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6

Figure 7.2 Conceptual Layout of Multidimensional Array

In declarations of multidimensioned arrays that have an *initializer-list* (as described in "Initializers" on page 217), the *constant-expression* that specifies the bounds for the first dimension may be omitted. For example:

```
const int cMarkets = 4;

// Declare a float that represents the transporation costs.
double TransportCosts[][cMarkets] =
{ { 32.19, 47.29, 31.99, 19.11 },
  { 11.29, 22.49, 33.47, 17.29 },
  { 41.97, 22.09,  9.76, 22.55 }  };
```

The above declaration defines an array that is three rows by four columns. The rows represent factories and the columns represent markets to which the factories ship. The values are the transportation costs from the factories to the markets. The first dimension of the array is left out, but the compiler fills it in by examining the initializer.

The technique of omitting the bounds specification for the first dimension of a multidimensioned array can also be used in function declarations as follows:

```
#include <float.h>          // Includes DBL_MAX.
#include <iostream.h>

const int    cMkts = 4;

// Declare a float that represents the transportation costs.
double TransportCosts[][cMkts] =
{ { 32.19, 47.29, 31.99, 19.11 },
  { 11.29, 22.49, 33.47, 17.29 },
  { 41.97, 22.09,  9.76, 22.55 }  };
// Calculate size of unspecified dimension.
const int cFactories = sizeof TransportCosts /
                       sizeof( double[cMkts] );

double FindMinToMkt( int Mkt, double TransportCosts[][cMkts],
                     int cFacts );

main(int argc, char *argv[] )
{
    double MinCost;

    MinCost = FindMinToMkt( *argv[1] - '0', TransportCosts,
                            cFacts );

    cout << "The minimum cost to Market " << argv[1] << " is: "
         << MinCost << "\n";

    return 0;
}
```

```
double FindMinToMkt( int Mkt, double TransportCosts[][cMkts],
                     int cFacts )
{
    double MinCost = DBL_MAX;

    for( int i = 0; i < cFacts; ++i )
        MinCost = (MinCost < TransportCosts[i][Mkt]) ?
                    MinCost : TransportCosts[i][Mkt];

    return MinCost;
}
```

The function `FindMinToMkt` is written such that adding new factories does not require any code changes, just a recompilation.

Using Arrays

Individual elements of arrays are accessed using the array subscript operator ([ ]). If a singly dimensioned array is used in an expression with no subscript, the array name evaluates to a pointer to the first element in the array. For example:

```
char chArray[10];

...

char *pch = chArray;      // Pointer to first element.
char   ch = chArray[0];   // Value of first element.
       ch = chArray[3];   // Value of fourth element.
```

When using multidimensioned arrays, various combinations are acceptable in expressions. The following example illustrates this:

```
double multi[4][4][3]; // Declare the array.

double (*p2multi)[3];  // Pointer to two-dimensional array.
double (*p1multi);     // Pointer to four-dimensional array.

cout << multi[3][2][3] << "\n";   // Use three subscripts.
p2multi = multi[3];               // Make p2multi point to
                                  //  fourth "plane" of multi.
p1multi = multi[3][2];            // Make p1multi point to
                                  //  fourth plane, second row
                                  //  of multi.
```

In the preceding code, `multi` is a three-dimensional array of type **double**.The `p2multi` pointer points to an array of type **double** of size three, and `p1multi` is a pointer to an array of type **double**. The array is used with one, two, and three subscripts in this example. Although it is more common to specify all the subscripts, as in the **cout** statement, it is sometimes useful to select a specific subset of array elements as shown in the succeeding statements.

Arrays in Expressions

When an identifier of an array type appears in an expression other than **sizeof**, address-of (**&**), or initialization of a reference, it is converted to a pointer to the first array element. For example:

```
char szError1[] = "Error: Disk drive not ready.";
char *psz = szError1;
```

The pointer `psz` points to the first element of the array `szError1`. Note that arrays, unlike pointers, are not modifiable l-values. Therefore, the following assignment is illegal:

```
szError1 = psz;
```

Interpretation of Subscript Operator

Like other operators, the subscript operator ([ ]) can be redefined by the user. The default behavior of the subscript operator, if not overloaded, is to combine the array name and the subscript using the following method:

*((*array-name*) + (*subscript*))

As in all addition that involves pointer types, scaling is performed automatically to adjust for the size of the type. Therefore, the resultant value is not *subscript* bytes from the origin of *array-name*; rather, it is the *subscript*th element of the array. (For more information about this conversion, see "Additive Operators" in Chapter 4, on page 104.)

Similarly, for multidimensional arrays, the address is derived using the following method:

*((*array-name*) + (*subscript*$_1$ * *subscript*$_2$... * *subscript*$_n$))

Indirection on Array Types

Use of the indirection operator (*) on an *n*-dimensional array type yields an *n*–1 dimensional array. If *n* is 1, a scalar (or array element) is yielded.

Ordering of C++ Arrays

C++ arrays are stored in row-major order. Row-major order means the last subscript varies the fastest.

Functions

This section explains function declarations. It includes discussions on:

- Function declaration (prototyping) syntax.
- Declaration of functions that take a varying number of arguments.
- Declaration of functions that require no arguments.
- Overloading of functions (an introduction).
- Restrictions on function declarations.
- Specification of argument lists.
- Default arguments to functions.

Function definition is covered in "Function Definitions" on page 213.

Syntax

decl-specifiers dname (*argument-declaration-list*) *cv-mod-list*$_{opt}$

argument-declaration-list:
  arg-declaration-list , ...

arg-declaration-list:
  argument-declaration
  arg-declaration-list , *argument-declaration*

argument-declaration:
  decl-specifiers declarator
  decl-specifiers declarator = *expression*
  decl-specifiers abstract-declarator$_{opt}$
  decl-specifiers abstract-declarator$_{opt}$ = *expression*

The identifier given by *dname* has the type "*cv-mod-list* function, taking *argument-declaration-list*, and returning type *decl-specifiers*."

Note that the keywords **const**, **volatile**, and many of the Microsoft-specific keywords can appear in *cv-mod-list* and in the declaration of the name. The following example shows several simple function declarations:

```
char *strchr( char *dest, char *src );
static int atoi( const char *ascnum ) const;
```

The following syntax explains the details of a function declaration:

Syntax

argument-declaration-list:
arg-declaration-list$_{opt}$ ...$_{opt}$
arg-declaration-list , ...

arg-declaration-list:
argument-declaration
arg-declaration-list , *argument-declaration*

argument-declaration:
decl-specifiers declarator
decl-specifiers declarator = *expression*
decl-specifiers abstract-declarator$_{opt}$
decl-specifiers abstract-declarator$_{opt}$ = *expression*

Variable Argument Lists

Function declarations in which the last member of *argument-declaration-list* is the ellipsis (...) can take a variable number of arguments. In these cases, C++ provides type checking only for the explicitly declared arguments. Variable argument lists can be used when it is desirable to make a function so general that even the number of arguments can vary. The **printf** family of functions is an example of functions that use variable argument lists.

To access arguments after those declared, use the macros contained in the standard include file STDARG.H as described in "Functions with Variable Argument Lists" on page 214.

Microsoft Specific

Microsoft C++ does not allow the ellipsis to be specified as an argument other than the first if there is no comma preceding the ellipsis. Therefore, the declaration `int Func( int i, ... );` is legal, but the declaration `int Func( int i ... );` is not.♦

Declaration of a function that takes a variable number of arguments requires that at least one "placeholder" argument be supplied, even if it is not used. If this placeholder argument is not supplied, there is no way to access the remaining arguments.

When arguments of type **char** are passed as variable arguments, they are converted to type **int**. Similarly, when arguments of type **float** are passed as variable arguments, they are converted to type **double**.

Declaring Functions that Take No Arguments

A function declared with the single keyword **void** in *argument-declaration-list* takes no arguments, as long as the keyword **void** is the first and only member of

argument-declaration list. Arguments of type **void** elsewhere in *argument-declaration-list* produce errors. For example:

```
long GetTickCount( void );            // OK
long GetTickCount( int Reset, void ); // Error
long GetTickCount( void, int Reset ); // Error
```

In C++, explicitly specifying that a function requires no arguments is the same as declaring a function with no *argument-declaration-list.* Therefore, the following two statements are identical:

```
long GetTickCount();
long GetTickCount( void );
```

Note that, while it is illegal to specify a **void** argument except as outlined above, types derived from type **void** (such as pointers to **void** and arrays of **void**) can appear anywhere in *argument-declaration-list.*

Function Overloading

C++ allows specification of more than one function of the same name in the same scope. These are called "overloaded functions" and are described in detail in Chapter 12, "Overloading." The purpose of overloaded functions is to allow programmers to supply different semantics for a function depending on the types and number of arguments.

For example, a **print** function that takes a string (or **char** *) argument performs very different tasks than one that takes an argument of type **double**. Overloading permits uniform naming and prevents programmers from having to invent names such as `print_sz` or `print_d`. Table 7.1 shows what parts of a function declaration C++ uses to differentiate between groups of functions with the same name in the same scope.

Table 7.1 Overloading Considerations

Function Declaration Element	Used for Overloading?
Function return type	No
Number of arguments	Yes
Type of arguments	Yes
Presence or absence of ellipsis	Yes
Use of **typedef** names	No
Unspecified array bounds	No
const or **volatile** (in *cv-mod-list*)	Yes
__near, **__far**, or **__huge** (in *cv-mod-list*)	Yes

Note Although functions can be distinguished on the basis of return type, they cannot be overloaded on this basis.

The following example illustrates how overloading can be used. Another way to solve the same problem is presented in "Default Arguments" on page 210.

```
#include <iostream.h>
#include <math.h>
#include <stdlib.h>

// Prototype three print functions.
int print( char *s );                   // Print a string.
int print( double dvalue );             // Print a double.
int print( double dvalue, int prec );   // Print a double with a
                                        //  given precision.

main( int argc, char *argv[] )
{
    const double d = 893094.2987;

    if( argc < 2 )
    {
        // These calls to print invoke print( char *s ).
        print( "This program requires one argument." );
        print( "The argument specifies the number of" );
        print( "digits precision for the second number" );
        print( "printed." );
    }

    // Invoke print( double dvalue ).
    print( d );

    // Invoke print( double dvalue, int prec ).
    print( d, atoi( argv[1] ) );

    return 0;
}

// Print a string.
int print( char *s )
{
    cout << s << endl;

    return cout.good();
}
```

```
// Print a double in default precision.
int print( double dvalue )
{
    cout << dvalue << endl;

    return cout.good();
}

// Print a double in specified precision.
//  Positive numbers for precision indicate how many digits
//  precision after the decimal point to show. Negative
//  numbers for precision indicate where to round the number
//  to the left of the decimal point.
int print( double dvalue, int prec )
{
    // Use table-lookup for rounding/truncation.
    static const double rgPow10[] = {
        10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1, 10E0,
        10E1,  10E2,  10E3,  10E4, 10E5,  10E6
    };
    const int iPowZero = 6;

    // If precision out of range, just print the number.
    if( prec < -6 || prec > 7 )
        return print( dvalue );


    // Scale, truncate, then rescale.
    dvalue = floor( dvalue / rgPow10[iPowZero - prec] ) *
                             rgPow10[iPowZero - prec];

    cout << dvalue << endl;

    return cout.good();
}
```

The preceding code shows overloading of the `print` function in file scope.

For restrictions on overloading and information on how overloading affects other elements of C++, see Chapter 12, "Overloading."

Restrictions on Functions

Functions cannot return arrays or functions. They can, however, return references or pointers to arrays or functions. Another way to return an array is to declare a structure with only that array as a member:

```
struct Address
{ char szAddress[31]; };

Address GetAddress();
```

It is illegal to define a type in either the return-type portion of a function declaration or in the declaration of any argument to a function. The following legal C code is illegal in C++:

```
enum Weather { Cloudy, Rainy, Sunny } GetWeather( Date Today )
```

The reason the preceding code is disallowed is because the type `Weather` has function scope local to `GetWeather`, and the return value cannot be properly used. Because arguments to functions have function scope, declarations made within the argument list would have the same problem if not allowed.

C++ does not support arrays of functions. However, arrays of pointers to functions can be useful. In parsing a Pascal-like language, the code is often separated into a lexical analyzer that parses tokens and a parser that attaches semantics to the tokens. If the analyzer returns a particular ordinal value for each token, code can be written to perform appropriate processing:

```
int ProcessFORToken( char *szText );
int ProcessWHILEToken( char *szText );
int ProcessBEGINToken( char *szText );
int ProcessENDToken( char *szText );
int ProcessIFToken( char *szText );
int ProcessTHENToken( char *szText );
int ProcessELSEToken( char *szText );

int (*ProcessToken[])( char * ) = {
    ProcessFORToken, ProcessWHILEToken, ProcessBEGINToken,
    ProcessENDToken, ProcessIFToken, ProcessTHENToken,
    ProcessELSEToken };
const int MaxTokenID = sizeof ProcessToken / sizeof( int (*)() );

...

int DoProcessToken( int TokenID, char *szText )
{
    if( TokenID < MaxTokenID )
        return (*ProcessToken[TokenID])( szText );
    else
        return Error( szText );
}
```

The Argument Declaration List

The *argument-declaration-list* portion of a function declaration:

- Allows the compiler to check type consistency between the arguments the function requires and the arguments supplied in the call.
- Enables conversions, either implicit or user-defined, to be performed from the argument type supplied to the required argument type.
- Checks initializations of or assignments to pointers to functions.
- Checks initializations of or assignments to references to functions.

Argument Lists in Function Prototypes (Nondefining Declaration)

The form of *argument-declaration-list* is a list of the type names of the arguments. Consider an *argument-declaration-list* for a function, `func`, that takes these three arguments:

- Pointer to type **char**
- **char**
- **int**

The code for such an *argument-declaration-list* can be written:

```
char *, char, int
```

The function declaration (the prototype), might therefore be written:

```
void func( char *, char, int );
```

While the preceding declaration contains enough information for the compiler to perform type checking and conversions, it does not provide much information about what the arguments are. A good way to document function declarations is to include identifiers as they would appear in the function definition, as in the following:

```
void func( char *szTarget, char chSearchChar, int nStartAt );
```

These identifiers in prototypes are useful only for default arguments, as they go out of scope immediately. They do, however, provide meaningful program documentation.

Argument Lists in Function Definitions

The argument list in a function definition differs from that of a prototype only in that the identifiers, if present, represent formal arguments to the function. The identifier names need not match those in the prototype (if there are any).

Note It is possible to define functions with unnamed arguments. However, these arguments are inaccessible to the function for which they are defined.

Default Arguments

In many cases, functions have a number of arguments that are used so infrequently that a default value would suffice. To address this, the default-argument facility allows for specifying only those arguments to a function that are meaningful in a given call. To illustrate this concept, consider the example presented in "Function Overloading" on page 205.

```
// Prototype three print functions.
int print( char *s );                  // Print a string.
int print( double dvalue );            // Print a double.
int print( double dvalue, int prec );  // Print a double with a
                                       //  given precision.
```

In many applications, a reasonable default can be supplied for `prec`, eliminating the need for two functions:

```
// Prototype two print functions.
int print( char *s );                    // Print a string.
int print( double dvalue, int prec=2 );  // Print a double with a
                                         //  given precision.
```

The implementation of the `print` function is changed slightly to reflect the fact that only one such function exists for type **double**:

```
// Print a double in specified precision.
//  Positive numbers for precision indicate how many digits
//  precision after the decimal point to show. Negative
//  numbers for precision indicate where to round the number
//  to the left of the decimal point.
int print( double dvalue, int prec )
{
    // Use table-lookup for rounding/truncation.
    static const double rgPow10[] = {
        10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1, 10E0,
        10E1,  10E2,  10E3,  10E4, 10E5,  10E6
    };
    const int iPowZero = 6;
```

```
    // If precision out of range, just print the number.
    if( prec >= -6 || prec <= 7 )
        // Scale, truncate, then rescale.
        dvalue = floor( dvalue / rgPow10[iPowZero - prec] ) *
                                 rgPow10[iPowZero - prec];

    cout << dvalue << endl;

    return cout.good();
}
```

To invoke the new `print` function, use code such as the following:

```
print( d );    // Precision of 2 supplied by default argument.
print( d, 0 ); // Override default argument to achieve other
               //  results.
```

There are several points to note when using default arguments:

- Default arguments are used only in function calls where trailing arguments are omitted—they must be the last argument(s). Therefore, the following code is illegal:

  ```
  int print( double dvalue = 0.0, int prec );
  ```

- A default argument cannot be redefined in later declarations even if the redefinition is identical to the original. Therefore, the following code produces an error:

  ```
  // Prototype for print function.
  int print( double dvalue, int prec = 2 );

  ...

  // Definition for print function.
  int print( double dvalue, int prec = 2 )
  {
  ...
  }
  ```

  The problem with the preceding code is that the function declaration in the definition redefines the default argument for `prec`.

- Additional default arguments can be added by later declarations.
- Default arguments can be provided for pointers to functions. For example:

  ```
  int (*pShowIntVal)( int i = 0 );
  ```

Default Argument Expressions

The expressions used for default arguments are often constant expressions, but this is not a requirement. The expression can combine functions that are visible in the current scope, constant expressions, and global variables. The expression cannot contain local variables or nonstatic class-member variables. The following code illustrates this:

```
BOOL CreateVScrollBar( HWND hWnd, short nWidth =
                       GetSystemMetrics( SM_CXVSCROLL ) );
```

The preceding declaration specifies a function that creates a vertical scroll bar of a given width for a window. If no width argument is supplied, the Windows API function, `GetSystemMetrics`, is called to find the default width for a scroll bar.

The default expression is not evaluated until the function call, but the evaluation is completed before the function call actually takes place.

Because formal arguments to a function are in function scope, and because the evaluation of default arguments takes place prior to entry to this scope, formal arguments cannot be used in default argument expressions. Use of local variables in default argument expressions is also disallowed.

Note that any formal argument declared prior to a default argument expression can potentially hide a global name in the function scope, which can cause errors. The following code is illegal:

```
const int Categories = 9;

void EnumCategories( char *Categories[], int n = Categories );
```

In the preceding code, the global name `Categories` is hidden at function scope, making the default argument expression invalid.

Other Considerations

The default argument is not considered part of the function type. Therefore, it is not used in selecting overloaded functions. Two functions that differ only in their default arguments are considered multiple definitions rather than overloaded functions.

Default arguments cannot be supplied for overloaded operators.

7.4 Function Definitions

Function definitions differ from function declarations in that they supply function bodies—the code that makes up the function.

Syntax

function-definition:
decl-specifiers$_{opt}$ *declarator* *ctor-initializer*$_{opt}$ *fct-body*

fct-body:
compound-statement

As discussed in "Functions" on page 203, the form of the declarator in the syntax is:

dname (*argument-declaration-list*) *cv-mod-list*$_{opt}$

The formal arguments declared in *argument-declaration-list* are in the scope of the function body.

Figure 7.3 shows the parts of a function definition. The shaded area is the function body.

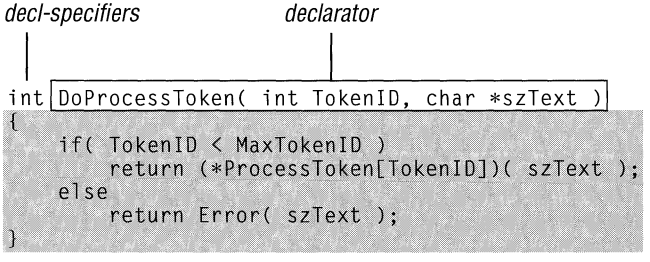


Figure 7.3 Parts of a Function Definition

The *cv-mod-list* element of the declarator syntax specifies how the **this** pointer is to be treated; it is only for use with class member functions. (For more information about the *cv-mod-list*, see "const and volatile Pointers" on page 188 and "Memory-Model Modifiers and Member Functions" in Appendix B, on page 400.)

The *ctor-initializer* element of the syntax is used only in constructors. Its purpose is to allow initialization of base classes and contained objects. (For more information about use of *ctor-initializer*, see "Initializing Bases and Members" in Chapter 11, on page 329.)

Functions with Variable Argument Lists

Functions that require variable lists are declared using the ellipsis (...) in the argument list, as described in "Variable Argument Lists" on page 204. To access arguments passed to functions using this method, use the types and macros described in the STDARG.H standard include file.

The following example shows how the **va_start**, **va_arg**, and **va_end** macros, along with the **va_list** type (declared in STDARG.H), work together:

```
#include <stdio.h>
#include <stdarg.h>

//  Declaration, but not definition, of ShowVar.
int ShowVar( char *szTypes, ... );

main()
{
    ShowVar( "fdcsi", 32.4f, 298.34E3, 'a', "Test string", 4 );

    return 0;
}
```

```
//  ShowVar takes a format string of the form
//   "ifdcs", where each character specifies the
//   type of the argument in that position.
//
//  i = int
//  f = float
//  d = double
//  c = char
//  s = string (char *)
//
//  Following the format specification is a list
//   of n arguments, where n == strlen( szTypes ).
int ShowVar( char *szTypes, ... )
{
    va_list vl;
    int i;

    //  szTypes is the last argument specified; all
    //   others must be accessed using the variable-
    //   argument macros.
    va_start( vl, szTypes );

    // Step through the list.
    for( i = 0; szTypes[i] != '\0'; ++i )
    {
        union Printable_t
        {
            int     i;
            float   f;
            double  d;
            char    c;
            char   *s;
        } Printable;
```

```
            switch( szTypes[i] )    // Type to expect.
            {
            case 'i':
                Printable.i = (int)va_arg( vl, int );
                printf( "%i\n", Printable.i );
                break;

            case 'f':
                Printable.f = (float)va_arg( vl, float );
                printf( "%f\n", Printable.f );
                break;

            case 'd':
                Printable.d = (double)va_arg( vl, double );
                printf( "%f\n", Printable.d );
                break;

            case 'c':
                Printable.c = (char)va_arg( vl, char );
                printf( "%c\n", Printable.c );
                break;

            case 's':
                Printable.s = (char *)va_arg( vl, char * );
                printf( "%s\n", Printable.s );
                break;

            default:
                break;
            }
        }
        va_end( vl );

        return 0;
}
```

The preceding example illustrates these important concepts:

- A list marker must be established as a variable of type **va_list** before any variable arguments are accessed. In the preceding example, the marker is called `vl`.
- The individual arguments are accessed using the **va_arg** macro. The **va_arg** macro needs to be told the type of argument to retrieve so it can transfer the correct number of bytes from the stack. If an incorrect type of a size different than that supplied by the calling program is specified to **va_arg**, the results are unpredictable.
- The result obtained using the **va_arg** macro should be explicitly cast to the desired type.
- The **va_end** macro must be called to terminate variable-argument processing.

7.5 Initializers

Declarators can specify the initial value for objects. The only way to specify a value for objects of **const** type is in the declarator. The part of the declarator that specifies this initial value is called the "initializer."

Syntax

initializer:
= *assignment-expression*
= { *initializer-list* ,opt }
(*expression-list*)

initializer-list:
expression
initializer-list , *expression*
{ *initializer-list* ,opt }

There are two fundamental types of initializers:

- The initializer invoked using the equal-sign syntax
- The initializer invoked using function-style syntax

Which type of initializer works best in programs is largely a matter of style. Note that using the equal-sign syntax is semantically the same as using the function-style syntax; however, it is not the same as making an assignment.

Consider the following code, which illustrates some declarators with initializers:

```
int      i = 7;                  // Uses equal-sign syntax.
Customer Cust( "Taxpayer, Joe",  // Uses function-style
               "14 Cherry Lane", //  syntax.
               "Manteca",
               "CA" );
```

The preceding code could also be written as follows, but the meaning of the initializations is not as clear:

```
int i( 7 );
Customer Cust = { "Taxpayer, Joe",
                  "14 Cherry Lane",
                  "Manteca",
                  "CA" };
```

The function-style initializer is more commonly used for objects like `Cust` that do not represent arithmetic types.

Declarations of automatic, register, static, and external variables can contain initializers. However, declarations of external variables can contain initializers only if the variables are not declared as **extern**.

These initializers can contain expressions involving constants and variables in the current scope. The initializer expression is evaluated at the point the declaration is encountered in program flow, or, for global static objects and variables, at program startup. (For more information about initialization of global static objects, see "Additional Startup Considerations" in Chapter 2, on page 43.)

Initializing Pointers to const Objects

A pointer to a **const** object can be initialized with a pointer to an object that is not **const**, but not vice versa. For example, the following initialization is legal:

```
Window StandardWindow;
const Window* pStandardWindow( &StandardWindow );
```

In the preceding code, the pointer `pStandardWindow` is declared as a pointer to a **const** object. Although `StandardWindow` is not declared as **const**, the declaration is acceptable because it does not allow non**const** access to a **const** object. The reverse of this is as follows:

```
const Window StandardWindow;
Window* pStandardWindow( &StandardWindow );
```

The above code explicitly declares `StandardWindow` as a **const** object. Initializing the non**const** pointer `pStandardWindow` with the address of `StandardWindow` generates an error because it allows non**const** access to the object through the pointer. That is, it allows removal of the **const** attribute from the object.

Uninitialized Objects

Objects and simple variables of storage class **static** that are declared with no initializer are guaranteed to be initialized to a bit pattern of zeros. No such special processing takes place for uninitialized objects of automatic or register storage classes. They have undefined values.

Initializing Static Members

Static members are considered to have class scope. Therefore, they can access other member data or functions. For example:

```
class DialogWindow
{
public:
    static short  GetTextHeight();
private:
    static short nTextHeight;
};

short DialogWindow :: nTextHeight = GetTextHeight();
```

Note that in the above definition of the static member `nTextHeight`, `GetTextHeight` is implicitly known to be `DialogWindow :: GetTextHeight`.

Initializing Aggregates

An aggregate type is a type that:

- Is an array or class type
- Has no constructors (for class types)
- Has no nonpublic members (for class types)
- Has no base classes (for class types)
- Has no virtual functions (for class types)

Initializers for aggregates can be specified as a comma-separated list of values enclosed in curly braces. For example, this code declares an **int** array of 10 and initializes it:

```
int rgiArray[10] = { 9, 8, 4, 6, 5, 6, 3, 5, 6, 11 };
```

The initializers are stored in the array elements in increasing subscript order. Therefore, `rgiArray[0]` is 9, `rgiArray[1]` is 8, and so on, until `rgiArray[9]`, which is 11. To initialize a structure, use code such as the following:

```
struct RCPrompt
{
    short nRow;
    short nCol;
    char *szPrompt;
};
RCPrompt rcContinueYN = { 24, 0, "Continue (Y/N?)" };
```

Length of Aggregate-Initializer Lists

If an aggregate initializer list is shorter than the array or class type that is being initialized, zeros are stored in the elements for which no initializer is specified. Therefore, the following two declarations are equivalent:

```
// Explicitly initialize all elements.
int rgiArray[5] = { 3, 2, 0, 0, 0 };

// Allow remaining elements to be zero-initialized.
int rgiArray[5] = { 3, 2 };
```

While initializer lists can be truncated, as shown above, supplying too many initializers generates an error.

Initializing Aggregates That Contain Aggregates

Some aggregates contain other aggregates—for example, arrays of arrays, arrays of structures, or structures that are composed of other structures. Initializers can be supplied for such constructs by initializing each one in the order it occurs with a brace-enclosed list. For example:

```
// Declare an array of type RCPrompt.
RCPrompt rgRCPrompt[4] =
{ { 4,  7, "Options Are:"  },
  { 6,  7, "1. Main Menu"  },
  { 8,  7, "2. Print Menu" },
  { 10, 7, "3. File Menu"  } };
```

Note that `rgRCPrompt` is initialized with a brace-enclosed list of brace-enclosed lists. The enclosed braces are not syntactically required, but they lend clarity to the declaration. The following example program shows how a two-dimensional array is filled by such an initializer:

```
#include <iostream.h>

main()
{
    int rgI[2][4] = { 1, 2, 3, 4, 5, 6, 7, 8 };

    for( int i = 0; i < 2; ++i )
        for( int j = 0; j < 4; ++j )
            cout << "rgI[" << i << "][" << j << "] = "
                 << rgI[i][j] << endl;

    return 0;
}
```

The output from this program is:

```
rgI[0][0] = 1
rgI[0][1] = 2
rgI[0][2] = 3
rgI[0][3] = 4
rgI[1][0] = 5
rgI[1][1] = 6
rgI[1][2] = 7
rgI[1][3] = 8
```

Short initialization lists can be used only with explicit subaggregate initializers and enclosed in braces. If `rgI` had been declared as

```
int rgI[2][4] = { { 1, 2 }, { 3, 4 } };
```

the program output would have been

```
rgI[0][0] = 1
rgI[0][1] = 2
rgI[0][2] = 0
rgI[0][3] = 0
rgI[1][0] = 3
rgI[1][1] = 4
rgI[1][2] = 0
rgI[1][3] = 0
```

Initializing Incomplete Types

Incomplete types, such as unbounded array types, can be initialized as follows:

```
char HomeRow[] = { 'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l' };
```

The compiler computes the size of the array from the number of initializers provided.

Incomplete types, such as pointers to class types that are declared but not defined, are declared as follows:

```
class DefinedElsewhere;        // Class definition elsewhere.
class DefinedHere
{
    ...
    friend class DefinedElsewhere;
};
```

Initializing Using Constructors

Objects of class type are initialized by calling the appropriate constructor for the class. For complete information about initializing class types, see "Explicit Initialization" in Chapter 11, on page 326.

Initializers and Unions

Objects of **union** type are initialized with a single value (if the union does not have a constructor). This is done in one of two ways:

- Initialize the union with another object of the same **union** type. For example:

```
struct Point
{
    unsigned x;
    unsigned y;
};

union PtLong
{
    long  l;
    Point pt;
};

...

PtLong ptOrigin;
PtLong ptCurrent = ptOrigin;
```

  In the preceding code, `ptCurrent` is initialized with the value of `ptOrigin`—an object of the same type.

- Initialize the union with a brace-enclosed initializer for the first member. For example:

```
PtLong ptCurrent = { 0x0a000aL };
```

Initializing Character Arrays

Character arrays can be initialized in one of two ways:

- Individually, as follows:

```
char chABCD[4] = { 'a', 'b', 'c', 'd' };
```

- With a string, as follows:

```
char chABCD[5] = "abcd";
```

In the second case, where the character array is initialized with a string, the compiler appends a trailing `'\0'` (end-of-string character). Therefore, the array must be at least one larger than the number of characters in the string.

Because most string handling uses the standard library functions or relies on the presence of the trailing end-of-string character, it is common to see unbounded array declarations initialized with strings:

```
char chABCD[] = "ABCD";
```

Initializing References

Variables of reference type must be initialized with an object of the type from which the reference type is derived, or with an object of a type that can be converted to the type from which the reference type is derived. For example:

```
int  iVar;
long lVar;

long& LongRef1 = lVar; // No conversion required.
long& LongRef2 = iVar; // Converted to type long.

LongRef1 = 23L;        // Change lVar through a reference.
LongRef2 = 11L;        // Change iVar through a reference.
```

Once initialized, a reference-type variable always points to the same object; it cannot be modified to point to another object.

Although the syntax can be the same, initialization of reference-type variables and assignment to reference-type variables are semantically different. In the preceding example, the assignments that change `iVar` and `lVar` look similar to the initializations but have completely different effects. The initialization specifies the object to which the reference-type variable points; the assignment assigns to the referred-to object through the reference.

Because both passing an argument of reference type to a function and returning a value of reference type from a function are initializations, the formal arguments to a function are initialized correctly, as are the references returned.

The only time reference-type variables can be declared without initializers is in the following:

- Function declarations (prototypes). For example:

```
int func( int& );
```

- Function-return type declarations. For example:

```
int& func( int& );
```

- Declaration of a reference-type class member. For example:

```
class c
{
public:
    int& i;
};
```

- Declaration of a variable explictly specified as **extern**. For example:

```
extern int& iVal;
```

When initializing a reference-type variable, the compiler uses the decision graph shown in Figure 7.4 (on the following page) to select between creating a reference to an object or creating a temporary object to which the reference points.

References to **volatile** types (declared as **volatile** *typename***&** *identifier*) can be initialized with **volatile** or non**volatile** objects of the same type. They cannot, however, be initialized with **const** objects of that type. Similarly, references to **const** types (declared as **const** *typename***&** *identifier*) can be initialized with **const** or non**const** objects of the same type (or anything that has a conversion to that type). They cannot, however, be initialized with **volatile** objects of that type.

References that are not qualified with either the **const** or **volatile** keyword can be initialized only with objects declared as neither **const** nor **volatile**.

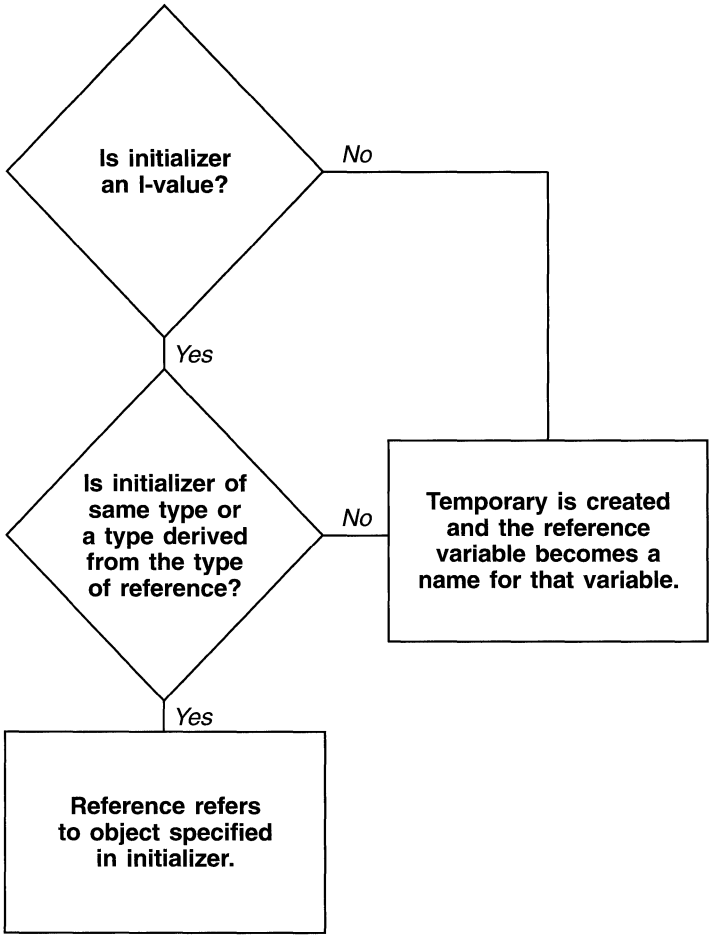
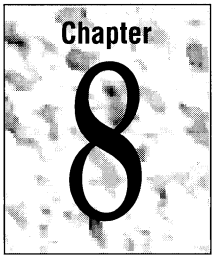


Figure 7.4 Decision Graph for Initialization of Reference Types

Classes



This chapter introduces C++ classes. Classes, which can contain data and functions, introduce user-defined types into a program. User-defined types in traditional programming languages are collections of data which, taken together, describe an object's attributes and state. Class types in C++ allow description of attributes and state, but they also allow definition of behavior.

8.1 Overview

Class types are defined using the **class**, **struct**, and **union** keywords. For simplicity, types defined with these keywords are called class declarations, except in discussions of language elements that behave differently depending on which keyword is used.

Microsoft Specific

"Local class" declarations—class declarations made in block scope—are exported to file scope in Microsoft C/C++. Classes not declared within another class always have file scope. ♦

Names of classes defined within another class ("nested") have class scope of the enclosing class.

Syntax

class-name:
    identifier

The variables and functions of a class are called members. When defining a class, it is common practice to supply the following members (although all are optional):

- Class data members, which define the state and attributes of an object of the class type.
- One or more "constructor" functions that initialize an object of the class type. Constructors are described in "Constructors" in Chapter 11, on page 300.

- One or more "destructor" functions that perform cleanup functions such as deallocating dynamically allocated memory or closing files. Destructors are described in "Destructors" in Chapter 11, on page 305.
- One or more member functions that define the object's behavior. These functions perform operations particular to objects of a specific class only.

Defining Class Types

Class types are defined using *class-specifiers.* Class types can be declared using *elaborated-type-specifiers* as shown in "Type Specifiers" in Chapter 6, on page 168.

Syntax

class-specifier:
class-head { *member-list*$_{opt}$ }

class-head:
class-key *imodel*$_{opt}$ *identifier*$_{opt}$ *base-spec*$_{opt}$
class-key *imodel*$_{opt}$ *class-name*$_{opt}$ *base-spec*$_{opt}$

class-key:
class
struct
union

imodel:
__near
__far
__export

Class names are introduced as identifiers immediately after the compiler processes them (before entry into the class body); they can be used to declare class members. This allows declaration of self-referential data structures, such as the following:

```
class Tree
{
public:
    void *Data;
    Tree *Left;
    Tree *Right;
};
```

Structures, Classes, and Unions

The three class types are structure, class, and union. They are declared using the **struct**, **class**, and **union** keywords (see *class-key* syntax above). Table 8.1 shows differences among the three class types.

Table 8.1 Access Control and Constraints of Structures, Classes, and Unions

Structures	Classes	Unions
class-key is **struct**	*class-key* is **class**	*class-key* is **union**
Default access is public	Default access is private	Default access is public
No usage constraints	No usage constraints	Use only one member at a time

Anonymous Class Types

Classes can be anonymous—that is, they can be declared without an *identifier*. This is useful in cases when you replace a class name with a **typedef** name, as in the following example:

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

Note The use of anonymous classes shown in the previous example is useful for preserving compatibility with existing C code. In some C code, the use of **typedef** in conjunction with anonymous structures is prevalent.

Anonymous classes are also useful when you want a reference to a class member to appear as though it were not contained in a separate class, as in the following example:

```
struct PTValue
{
    POINT ptLoc;
    union
    {
        int  iValue;
        long lValue;
    };
};

PTValue ptv;
```

In the preceding code, `iValue` can be accessed using the object member-selection operator (.) as follows:

```
int i = ptv.iValue;
```

Anonymous classes are subject to certain restrictions. (For more information about anonymous unions, see "Unions" on page 249.) Anonymous classes:

- Cannot have a constructor or destructor.
- Cannot be passed as arguments to functions (unless type checking is defeated using ellipses).
- Cannot be returned as return values from functions.

Point of Class Definition

A class is defined at the end of its *class-specifier*. Member functions need not be defined in order for the class to be considered defined. Consider the following example:

```
class Point                     // Point class
{                               //  considered defined.
public:
    Point()
        { cx = cy = 0; }        // Constructor defined.
    Point( int x, int y )
        { cx = X, cy = Y; }     // Constructor defined.
    unsigned &x( unsigned );    // Accessor declared.
    unsigned &y( unsigned );    // Accessor declared.
private:
    unsigned  cx, cy;
};
```

Even though the two accessor functions (`x` and `y`) are not yet defined, the class `Point` is considered defined. (Accessor functions are functions provided to give safe access to member data.)

Class-Type Objects

An object is a typed region of storage in the execution environment; in addition to retaining state information, it also defines behavior. Class-type objects are defined using *class-name*. Consider the following code fragment:

```
class Account                          // Class name is Account.
{
public:
            Account();                 // Default constructor.
            Account( double );         // Construct from double.
    double& Deposit( double );
    double& Withdraw( double, int );
    ...
};

Account CheckingAccount;               // Define object of class type.
```

The preceding code declares a class (a new type) called `Account`. It then uses this new type to define an object called `CheckingAccount`.

The following operations are defined by C++ for objects of class type:

- Assignment. One object can be assigned to another. The default behavior for this operation is a memberwise copy. This behavior can be modified by supplying a user-defined assignment operator.
- Explicit initialization of an object. For example:

  ```
  Point myPoint = thatPoint;
  ```

  declares `myPoint` as an object of type `Point` and initializes it to the value of `thatPoint`.
- Initialization caused by passing as an argument. Objects can be passed to functions either by value or by reference. If they are passed by value, a copy of each object is passed to the function. The default method for creating the copy is memberwise copy; this can be modified by supplying a user-defined copy constructor (a constructor that takes a single argument of the type "reference to class").
- Initialization caused by returning as the result of a function. Objects can be returned from functions either by value or by reference. The default method for returning an object by value is a memberwise copy; this can be modified by supplying a user-defined copy constructor. An object returned by reference (using pointer or reference types) should not be both automatic and local to the called function. If it is, the object referred to by the return value will have gone out of scope before it can be used.

"Overloaded Operators" in Chapter 12, on page 351 explains how to redefine other operators on a class-by-class basis.

Empty Classes

You can declare empty classes, but objects of such types still have nonzero size. The following example illustrates this:

```
#include <iostream.h>

class NoMembers
{
};

int main()
{
    NoMembers n;  // Object of type NoMembers.

    cout << "The size of an object of empty class is: "
         << sizeof n << endl;

    return 0;
}
```

The output of the preceding program is:

```
The size of an object of empty class is: 1.
```

The memory allocated for such objects is of nonzero size so that the objects have different addresses. Having different addresses is important because it must be possible to compare pointers to objects for identity. Without distinct addresses, such a comparison is impossible. Also, in arrays, each member array must have a distinct address.

Important An empty class is a class with no data members. Empty classes can have functions, including constructors, destructors, and so on, which define their behavior.

Microsoft Specific

An empty base class typically contributes zero bytes to the size of a derived class.♦

8.2 Class Names

Class declarations introduce new types, called class names, into programs. These class declarations also act as definitions of the class for a given translation unit. There may be only one definition for a given class type per translation unit. Using these new class types, you can declare objects, and the compiler can perform type checking to verify that no operations incompatible with the types are performed on the objects.

An example of such type checking is:

```
class Point
{
public:
    unsigned x, y;
};

class Rect
{
public:
    unsigned x1, y1, x2, y2;
};

// Prototype a function that takes two arguments, one of type
//  Point and the other of type pointer to Rect.
int PtInRect( Point, Rect & );

...

Point pt;
Rect  rect;

rect = pt;   // Error. Types are incompatible.
pt = rect;   // Error. Types are incompatible.

// Error. Arguments to PtInRect are reversed.
cout << "Point is " << PtInRect( rect, pt ) ? "" : "not"
     << " in rectangle." << endl;
```

As the preceding code illustrates, operations (such as assignment and argument passing) on class-type objects are subject to the same type checking as objects of built-in types.

Because the compiler distinguishes between class types, functions can be overloaded on the basis of class-type arguments as well as built-in type arguments. For more information about overloaded functions, see "Function Overloading" in Chapter 7, on page 205 and Chapter 12, "Overloading."

Declaring and Accessing Class Names

Class names can be declared in global or class scope. If they are declared in class scope, they are referred to as "nested" classes.

Microsoft Specific

Block-scoped class declarations, or "local" class declarations, are not permitted in Microsoft C/C++. However, use of simple aggregates such as C structures is allowed in block scope.♦

Any class name introduced in class scope hides other elements of the same name in an enclosing scope. Names hidden by such a declaration can then be referred to only by using an *elaborated-type-specifier*. The following example shows an example of using an *elaborated-type-specifier* to refer to a hidden name:

```
struct A      // Global scope definition of A.
{
    int a;
};

void main()
{
    char A = 'a'; // Redefine the name A as an object.

    struct A AObject;
    ...
}
```

Because the name `A` that refers to the structure is hidden by the `A` that refers to the `char` object, **struct** (a *class-key*) must be used to declare `AObject` as type `A`.

You can use the *class-key* to declare a class without providing a definition. This nondefining declaration of a class introduces a class name for forward reference. This technique is useful when designing classes that refer to one another in **friend** declarations. It is also useful when class names must be present in header files but the definition is not required. For example:

```
// RECT.H
class Point;   // Nondefining declaration of class Point.
class Line
{
public:
    int Draw( Point &ptFrom, Point &ptTo );
    ...
};
```

In the preceding sample, the name `Point` must be present, but it need not be a defining declaration that introduces the name.

typedef Statements and Classes

Using the **typedef** statement to name a class type causes the **typedef** name to become a *class-name*. For more information about using **typedef**, see "typedef Specifier" in Chapter 6, on page 163.

8.3 Class Members

Classes can have these kinds of members:

- Member functions. (See "Member Functions" on page 240.)
- Data members.
- Classes, which include classes, structures, and unions. (See "Nested Class Declarations" on page 254 and "Unions" on page 249.
- Enumerations. (See "Enumeration Declarations" in Chapter 6, on page 173.)
- Bit fields. (See "Bit Fields" on page 252.)
- Friends. (See "Friends" in Chapter 10, on page 290.)
- Type names. (See "Type Names in Class Scope" on page 257.)

Note Friends are included in the preceding list because they are contained in the class declaration. However, they are not true class members, because they are not in the scope of the class.

Syntax

member-list:
  member-declaration *member-list*$_{opt}$
  access-specifier **:** *member-list*$_{opt}$

member-declaration:
  decl-specifiers$_{opt}$ *member-declarator-list*$_{opt}$ **;**
  function-definition$_{opt}$ **;**
  qualified-name **;**

member-declarator-list:
  member-declarator
  member-declarator-list **,** *member-declarator*

member-declarator:
  declarator *pure-specifier*$_{opt}$
  identifier$_{opt}$ **:** *constant-expression*

pure-specifier:
  = 0

The purpose of the *member-list* is to:

- Declare the complete set of members for a given class.
- Specify the access (public, private, or protected) associated with various class members.

In the declaration of a member list, you can declare members only once; redeclaration of members produces an error message. Because a member list is a complete

set of the members, you cannot add members to a given class with subsequent declarations.

Class members can be divided into two general categories: member data and member functions. While it is acceptable to overload (reuse) function names within a member list, the same is not true for data member names. Data member names can be declared only once within a class.

Member declarators cannot contain initializers. Supplying an initializer produces an error message. The following code illustrates this error:

```
class CantInit
{
public:
    long l = 7;       // Error: attempt to initialize
                      //  class member.
    static int i = 9; // Error: must be defined and initialized
                           outside of class declaration.
};
```

Because a separate instance of nonstatic member data is created for each object of a given class type, the correct way to initialize member data is to use the class's constructor. (Constructors are covered in "Constructors," in Chapter 11, on page 300.) Only one shared copy of static data members exists for all objects of a given class type. Static data members must be defined and can be initialized at file scope. (For more information about static data members, see "Static Data Members" on page 247.) The following example shows how to perform these initializations:

```
class CanInit
{
public:
    CanInit() { l = 7; } // Initializes l when new objects of type
                         //  CanInit are created.
    long       l;
    static int i;
    static int j;
};

int CanInit::i = 15;   // i is defined at file scope and
                       //  initialized to 15. The initializer
                       //  is evaluated in the scope of CanInit.
int CanInit::j = i;    // The right side of the initializer
                       //  is in the scope of the object being
                       //  initialized.
```

Note The class name, `CanInit`, must precede `i` to specify that the `i` being defined is a member of class `CanInit`.

Class-Member Declaration Syntax

Member data cannot be declared as **auto**, **extern**, or **register** storage class. They can, however, be declared as having **static** storage class.

The *decl-specifiers* specifiers can be omitted in member-function declarations. For information on *decl-specifiers*, see "Specifiers" on page 156 and "Member Functions" on page 240; see also "Functions" in Chapter 7, on page 203. The following code is therefore legal and declares a function that returns type **int**:

```
class NoDeclSpec
{
public:
    NoSpecifiers();
};
```

When you declare a friend class in a member list, you can omit the *member-declarator-list*. For more information on friends, see "friend Specifier" in Chapter 6, on page 167 and "Friends" in Chapter 10, on page 290. Even if a class name has not been introduced, it can be used in a **friend** declaration. This **friend** declaration introduces the name. However, in member declarations for such classes, the *elaborated-type-specifier* syntax must be used, as shown in the following example:

```
class HasFriends
{
public:
    friend class NotDeclaredYet;
};
```

In the preceding example, there is no *member-declarator-list* after the class declaration. Because the declaration for `NotDeclaredYet` has not yet been processed, the *elaborated-type-specifier* form is used: `class NotDeclaredYet`. A type that has been declared can be used in a **friend** member declaration using a normal type specifier:

```
class AlreadyDeclared
{
    ...
};

class HasFriends
{
public:
    friend AlreadyDeclared;
};
```

The *pure-specifier* (shown in the following example) indicates that no implementation is supplied for the virtual function being declared. Therefore, the *pure-specifier* can be specified only on virtual functions. Consider this example:

```
class StrBase  // Base class for strings.
{
public:
    virtual int IsLessThan( StrBase& ) = 0;
    virtual int IsEqualTo( StrBase& ) = 0;
    virtual StrBase& CopyOf( StrBase& ) = 0;
    ...
};
```

The preceding code declares an abstract base class—that is, a class designed to be used only as the base class for more specific classes. Such base classes can enforce a particular protocol, or set of functionality, by declaring one or more virtual functions as "pure" virtual functions, using the *pure-specifier*.

Classes that inherit from the `StrBase` class must provide implementations for the pure virtual functions or they, too, are considered abstract base classes.

Abstract base classes cannot be used to declare objects. For example, before an object of a type inherited from `StrBase` can be declared, the functions `IsLessThan`, `IsEqualTo`, and `CopyOf` must be implemented. (For more information about abstract base classes, see "Abstract Classes" in Chapter 9, on page 280.)

Using Type Names Within Class Declarations

The type name being declared can be used within a class declaration as long as the size of the class is not used. Exceptions are inline functions and default arguments.

Inline functions behave as if they were defined immediately after the class is declared. (For more information, see "Inline Member Functions" on page 246.) Default arguments are not evaluated until the point of the function call. Therefore, the size of the class is known before the inline function definition or default argument evaluation is processed.

Declaring Unsized Arrays in Member Lists (Microsoft Specific)

Unsized arrays can be declared in class member lists as the last data member if the program is not compiled with the ANSI-compatibility option (/Za). Because this is a Microsoft extension, using unsized arrays in this way can make your code less portable. To declare an unsized array, simply omit the first dimension. For example:

```
class Symbol
{
public:
    int  SymbolType;
    char SymbolText[];
};
```

Restrictions

If a class contains an unsized array, it cannot be used as the base class for another class. In addition, a class containing an unsized array cannot be used to declare any member except the last member of another class.

The **sizeof** operator, when applied to a class containing an unsized array, returns the amount of storage required for all members except the unsized array. Implementors of classes that contain unsized arrays should provide alternate methods for obtaining the correct size of the class.

You cannot declare arrays of objects that have unsized array components. Also, performing pointer arithmetic on pointers to such objects generates an error message.

Storage of Class-Member Data

Nonstatic class-member data is stored such that items falling between access specifiers are stored at successively higher memory addresses. No ordering across access specifiers is guaranteed.

Microsoft Specific

Depending on the /Zp compilation option, or the **pack** pragma, intervening space can be allocated to align member data on word or double-word boundaries. (For more information about the /Zp compilation option or the **pack** pragma, see Chapter 13, "Preprocessing.")

In Microsoft C++, class-member data is stored at successively higher memory addresses, even though the C++ language does not require it. Basing assumptions on this ordering can lead to nonportable code. ♦

Member Naming Restrictions

A function with the same name as the class in which it is declared is a constructor. A constructor is implicitly called when an object of this class type is created. (For more information about constructors, see "Constructors" in Chapter 11, on page 300.)

The following items cannot have the same name as the classes in whose scope they are declared: data members (both static and nonstatic), enclosed enumerators, members of anonymous unions, and nested class types.

8.4 Member Functions

Classes can contain data and functions. These functions are referred to as "member functions." Any nonstatic function declared inside a class declaration is considered a member function and is called using the member-selection operators (. and –>). When calling member functions from other member functions of the same class, the object and member-selection operator can be omitted. For example:

```
class Point
{
public:
    short x() { return _x; }
    short y() { return _y; }
    void  Show() { cout << "(" << x() << ", " << y() << "\n"; }
private:
    short _x, _y;
};

void main()
{
    Point pt;

    pt.Show();
}
```

Note that in the member function, `Show`, calls to the other member functions, `x` and `y`, are made without member-selection operators. These calls implicitly mean `this->x()` and `this->y()`. However, in **main**, the member function, `Show`, must be selected using the object `pt` and the member-selection operator (.).

Static functions declared inside a class can be called using the member-selection operators or by specifying the fully qualified function name (including the class name).

Note A function declared using the **friend** keyword is not considered a member of the class in which it is declared a **friend** (although it can be a member of another

class). A friend declaration controls the access a nonmember function has to class data.

The following class declaration shows how member functions are declared:

```
class Point
{
public:
    unsigned GetX();
    unsigned GetY();
    unsigned SetX( unsigned x );
    unsigned SetY( unsigned y );
private:
    unsigned ptX, ptY;
};
```

In the preceding class declaration, four functions are declared: `GetX`, `GetY`, `SetX`, and `SetY`. The next example shows how such functions are called in a program:

```
int main()
{
    // Declare a new object of type Point.
    Point ptOrigin;

    // Member function calls use the . member-selection operator.
    ptOrigin.SetX( 0 );
    ptOrigin.SetY( 0 );

    // Declare a pointer to an object of type Point.
    Point *pptCurrent = new Point;

    // Member function calls use the -> member-selection operator.
    pptCurrent->SetX( ptOrigin.GetX() + 10 );
    pptCurrent->SetY( ptOrigin.GetY() + 10 );

    return 0;
}
```

In the preceding code, the member functions of the object `ptOrigin` are called using the member-selection operator (.). However, the member functions of the object pointed to by `pptCurrent` are called using the –> member-selection operator.

Overview of Member Functions

Member functions are either static or nonstatic. The behavior of static member functions differs from that of other member functions because static member functions have no implicit **this** argument. Member functions, whether static or nonstatic, can be defined either in the class declaration or outside the class declaration.

If a member function is defined inside a class declaration, it is treated as an inline function, and there is no need to qualify the function name with its class name. Although functions defined inside class declarations are already treated as inline functions, you can use the **inline** keyword to document code. (For more information, see "Inline Member Functions" on page 246.)

An example of declaring a function within a class declaration follows:

```
class Account
{
public:
    // Declare the member function Deposit within the declaration
    //  of class Account.
    double Deposit( double HowMuch )
    {
        balance += HowMuch;
        return balance;
    }
private:
    double balance;
};
```

If a member function's definition is outside the class declaration, it is treated as an inline function only if it is explicitly declared as **inline**. In addition, the function name in the definition must be qualified with its class name using the scope-resolution operator (**::**).

The following example is identical to the previous declaration of class `Account`, except the `Deposit` function is defined outside the class declaration:

```
class Account
{
public:
    // Declare the member function Deposit but do not define it.
    double Deposit( double HowMuch );
private:
    double balance;
};

inline double Account::Deposit( double HowMuch )
{
    balance += HowMuch;
    return balance;
}
```

Note While member functions can be defined either inside a class declaration or separately, no member functions can be added to a class after the class is defined.

Classes containing member functions can have many declarations, but the member functions themselves must have only one definition in a program. Multiple

definitions cause an error message at link time. If a class contains inline function definitions, the function definitions must be identical to observe this "one definition" rule.

Nonstatic Member Functions

Nonstatic member functions have an implied argument, **this**, that points to the object through which the function is invoked. The type of **this** is *type* * **const**. These functions are considered to have class scope and can use class data and other member functions in the same class scope directly. In the preceding example, the expression `balance += HowMuch` adds the value of `HowMuch` to the class member `balance`. Consider the following statements:

```
Account Checking;

Checking.Deposit( 57.00 );
```

The preceding code declares an object of type `Account`, then invokes the member function `Deposit` to add $57.00 to it. In the function `Account::Deposit`, balance is taken to mean `Checking.balance` (the balance member for this object).

Nonstatic member functions are intended to operate on objects of their class type. Calling such a function on objects of different types (using explicit type conversions) causes undefined behavior.

Static Member Functions

Static member functions are considered to have class scope. In contrast to nonstatic member functions, these functions have no implicit **this** argument; therefore, they can use only static data members, enumerators, or nested types directly. Static member functions can be accessed without using an object of the corresponding class type. Consider this example:

```
class WindowManager
{
public:
    static int  CountOf();              // Return count of open windows.
           void Minimize();             // Minimize current window.
           WindowManager SideEffects(); // Function with side effects.
    ...
private:
    static int wmWindowCount;
};
```

```
int WindowManager::wmWindowCount = 0;

...

// Minimize (show iconic) all windows
for( int i = 0; i < WindowManager::CountOf(); ++i )
    rgwmWin[i].Minimize();
```

In the preceding code, the class `WindowManager` contains the static member function `CountOf`. This function returns the number of windows open but is not necessarily associated with a given object of type `WindowManager`. This concept is demonstrated in the loop where the `CountOf` function is used in the controlling expression; because `CountOf` is a static member function, it can be called without reference to an object.

Static member functions have external linkage. These functions do not have **this** pointers (covered in the next section). As a result, the following restrictions apply to such functions:

- They cannot access nonstatic class member data using the member-selection operators (. or –>).
- They cannot be declared as **virtual**.
- They cannot have the same name as a nonstatic function that has the same argument types.

Note The left side of a member-selection operator (. or –>) that selects a static member function is not evaluated. This can be important if the function is used for its side effects. For example, the expression `SideEffects().CountOf()` does not call the function `SideEffects`.

The this Pointer

All nonstatic member functions can use the **this** keyword, which is a **const** (nonmodifiable) pointer to the object for which the function was called. Member data is addressed by evaluating the expression **this–>***member-name* (although this technique is seldom used). In member functions, using a member name in an expression implicitly uses **this–>***member-name* to select the correct function or data member.

Note Because the **this** pointer is nonmodifiable, assignments to **this** are not allowed. Earlier implementations of C++ allowed assignments to **this**.

Occasionally, the **this** pointer is used directly—for example, in manipulation of self-referential data structures, where the address of the current object is required.

Type of this Pointer

The **this** pointer's type can be modified in the function declaration by the **const**, **volatile**, **__near**, **__far**, and **__huge** keywords. To declare a function as having the attributes of one or more of these keywords, use the *cv-mod-list* grammar which follows. (For more information, see "Memory-Model Modifiers and Member Functions" in Appendix B, on page 400.)

Syntax

cv-mod-list:
  cv-qualifier cv-mod-list$_{opt}$
  pmodel cv-mod-list$_{opt}$

cv-qualifier:
  const
  volatile

Microsoft Specific

pmodel:
  __near
  __far
  __huge ♦

Consider this example:

```
class Point
{
    unsigned X() const;
};
```

The preceding code declares a member function, `X`, in which the **this** pointer is treated as a **const** pointer to a **const** object. Combinations of *cv-mod-list* options can be used, but they always modify the object pointed to by **this**, not the **this** pointer itself. Therefore, the following declaration declares function `X`; the **this** pointer is a **const** pointer to a **const** object that is addressed far (using intersegment addressing):

```
class Point
{
    unsigned X() __far const;
};
```

The type of **this** is described by the following syntax, where *cv-qualifier-list* can be **const** or **volatile**, *class-type* is the name of the class, and *imodel* is a memory-model option:

cv-qualifier-list$_{opt}$ *class-type imodel*$_{opt}$ *** const this**

Table 8.2 explains more about how these modifiers work.

Table 8.2 Semantics of this Modifiers

Modifier	Meaning
const	Cannot change member data; cannot invoke non**const** member functions.
volatile	Member data is loaded from memory each time it is accessed; disables certain optimizations.
__near	Addressing is intrasegment (16-bit)(Microsoft specific).
__far	Addressing is intersegment; pointer arithmetic is 16-bit. Individual objects must be smaller than 64K (Microsoft specific).
__huge	Addressing is intersegment; pointer arithmetic is 32-bit (use **__huge** for large arrays of large objects) (Microsoft specific).

For objects explicitly declared as **const**, it is an error to call non**const** member functions. Similarly, for objects explicitly declared as **volatile**, it is an error to call non**volatile** member functions. This model is followed for **__near**, **__far**, and **__huge**.

Member functions declared as **const** cannot change member data—in such functions, **this** is a pointer to a **const** object.

Note Constructors and destructors cannot be declared as **const** or **volatile**. They can, however, be invoked on **const** or **volatile** objects. Constructors and destructors can be declared as **__near**, **__far**, or **__huge**; which constructor or destructor is called depends on the addressing of the object being created or destroyed.

Inline Member Functions

Member functions defined within a class declaration are considered inline functions. Calls to these functions are usually replaced with the actual code defined in the function body.

A member or nonmember function defined outside the class declaration can be specified as inline by using the **inline** keyword. For example:

```
class Rect
{
public:
    inline void Set( unsigned x1, unsigned y1,
                     unsigned x2, unsigned y2 );
private:
    unsigned private_x1, private_y1, private_x2, private_y2;
};

void Rect::Set( unsigned x1, unsigned y1,
                  unsigned x2, unsigned y2 );
{
    private_x1 = x1;
    private_y1 = y1;
    private_x2 = x2;
    private_y2 = y2;
}
```

Notice that the definition of `Rect::Set` is outside the class declaration, but it is still an inline function.

Inline functions declared inside class declarations behave as if they were defined just after the class declaration. Therefore, because the class is considered completely defined, inline functions can access the class name and all class members.

8.5 Static Data Members

Classes can contain static member data and member functions. When a data member is declared as **static**, only one copy of the data is maintained for all objects of the class. (For more information, see "Static Member Functions" on page 243.)

Static data members are not part of objects of a given class type; they are separate objects. As a result, the declaration of a static data member is not considered a definition. The data member is declared in class scope, but definition is performed at file scope. These static members have external linkage. The following example illustrates this:

```
class BufferedOutput
{
public:
    // Return number of bytes written by any object of this class.
    short BytesWritten() { return bytecount; }

    // Reset the counter.
    static void ResetCount() { bytecount = 0; }

    // Static member declaration.
    static long bytecount;
};

// Define bytecount in file scope.
long BufferedOutput::bytecount;
```

In the preceding code, the member `bytecount` is declared in class `BufferedOutput`, but it must be defined outside the class declaration.

Static data members can be referred to without referring to an object of class type. The number of bytes written using `BufferedOutput` objects can be obtained as follows:

```
long nBytes = BufferedOutput::bytecount;
```

For the static member to exist, it is not necessary that any objects of the class type exist. Static members can also be accessed using the member-selection (. and –>) operators. For example:

```
BufferedOutput Console;

long nBytes = Console.bytecount;
```

In the preceding case, the reference to the object (`Console`) is not evaluated; the value returned is that of the static object `bytecount`.

Static data members are subject to class-member access rules, so private access to static data members is allowed only for class-member functions and friends. These rules are described in Chapter 10, "Member-Access Control." The exception is that static data members must be defined in file scope regardless of their access restrictions. If the data member is to be explicitly initialized, an initializer must be provided with the definition.

The type of a static member is not qualified by its class name. Therefore, the type of `BufferedOutput::bytecount` is `long`.

8.6 Unions

Unions are class types that can contain only one data element at a time (although the data element can be an array or a class type). The members of a union represent the kinds of data the union can contain. An object of union type requires enough storage to hold the largest member in its *member-list*. Consider the following example:

```
#include <stdlib.h>
#include <string.h>
#include <limits.h>

union NumericType        // Declare a union that can hold the following:
{
    int         iValue;  // int value
    long        lValue;  // long value
    double      dValue;  // double value
};

int main( int argc, char *argv[] )
{
    NumericType *Values = new NumericType[argc - 1];

    for( int i = 1; i < argc; ++i )
        if( strchr( argv[i], '.' ) != 0 )
           // Floating type. Use dValue member for assignment.
            Values[i].dValue = atof( argv[i] );
        else
           // Not a floating type.
        {
                // If data is bigger than largest int, store it in
                //  lValue member.
            if( atol( argv[i] ) >  INT_MAX )
                Values[i].lValue = atol( argv[i] );
            else
                // Otherwise, store it in iValue member.
                Values[i].iValue = atoi( argv[i] );
        }

    return 0;
}
```

The `NumericType` union is arranged in memory (conceptually) as shown in Figure 8.1.

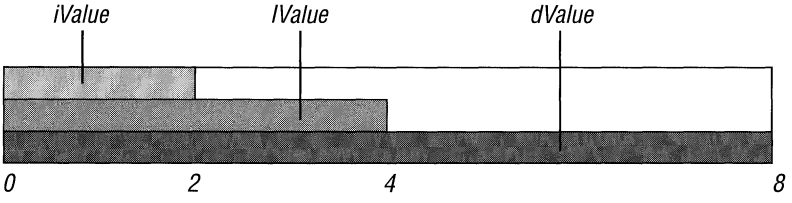


Figure 8.1 Storage of Data in NumericType Union

Member Functions in Unions

In addition to member data, unions can have member functions, as described in "Member Functions" on page 240. While unions can have special functions such as constructors and destructors, unions cannot contain virtual functions. (For more information about constructors, see page 300; for more information on destructors, see page 305 in Chapter 11.)

Unions as Class Types

Unions cannot have base classes; that is, they cannot inherit the attributes of other unions, structures, or classes. Unions also cannot be used as base classes for further inheritance.

Inheritance is covered in detail in Chapter 9, "Derived Classes."

Union Member Data

Unions can contain most types in their member lists, except for the following:

- Class types that have constructors or destructors
- Class types that have user-defined assignment operators
- Static data members

Anonymous Unions

Anonymous unions are unions that are declared without a *class-name* or *declarator-list.*

Syntax **union** { *member-list* } ;

Such union declarations do not declare types—they declare objects. The names declared in an anonymous union cannot conflict with other names declared in the same scope.

Names declared in an anonymous union are used directly, like nonmember variables. The following example illustrates this:

```
#include <iostream.h>

struct DataForm
{
    enum DataType { CharData = 1, IntData, StringData };
    DataType type;

    // Declare an anonymous union.
    union
    {
        char  chCharMem;
        char *szStrMem;
        int   iIntMem;
    };
    void print();
};

void DataForm::print()
{
    // Based on the type of the data, print the
    // appropriate data type.
    switch( type )
    {
    case CharData:
        cout << chCharMem;
        break;

    case IntData:
        cout << szStrMem;
        break;

    case StringData:
        cout << iIntMem;
        break;
    }
}
```

In the function `DataForm::print`, the three members (`chCharMem`, `szStrMem`, and `iIntMem`) are accessed as though they were declared as members (without the **union** declaration). However, the three union members share the same memory.

In addition to the restrictions listed in "Union Member Data" on page 250, anonymous unions are subject to additional restrictions:

- They must also be declared as **static** if declared in file scope.
- They can have only public members; private and protected members in anonymous unions generate errors.
- They cannot have function members.

Note Simply omitting the *class-name* portion of the syntax does not make a union an anonymous union. For a union to qualify as an anonymous union, the declaration must not declare an object.

8.7 Bit Fields

Classes and structures can contain members that occupy less storage than an integral type. These members are specified as bit fields. The syntax for bit-field *member-declarator* specification follows:

Syntax

declarator$_{opt}$ **:** *constant-expression*

The *declarator* is the name by which the member is accessed in the program. It must be an integral type (including enumerated types). The *constant-expression* specifies the number of bits the member occupies in the structure. Anonymous bit fields—that is, bit-field members with no identifier—can be used for padding.

Note An unnamed bit field of width 0 forces alignment of the next bit field to the next *type* boundary, where *type* is the type of the member.

The following example declares a structure that contains bit fields:

```
struct Date
{
    unsigned nWeekDay  : 3;    // 0..7   (3 bits)
    unsigned nMonthDay : 6;    // 0..31  (6 bits)
    unsigned nMonth    : 5;    // 0..12  (5 bits)
    unsigned nYear     : 8;    // 0..100 (8 bits)
};
```

The conceptual memory layout of an object of type `Date` is shown in Figure 8.2.

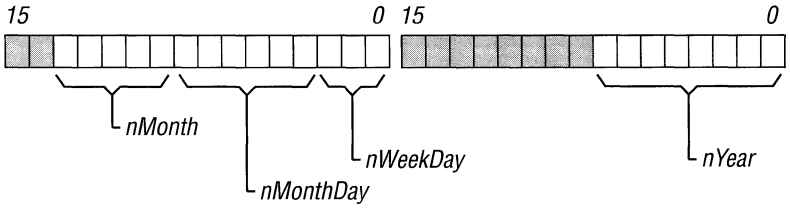


Figure 8.2 Memory Layout of Date Object

Note that `nYear` is 8 bits long and would overflow the word boundary of the declared type, **unsigned int**. Therefore, it is begun at the beginning of a new **unsigned int**. It is not necessary that all bit fields fit in one object of the underlying type; new units of storage are allocated, according to the number of bits requested in the declaration.

Microsoft Specific

The ordering of data declared as bit fields is from low to high bit, as shown in Figure 8.2.♦

If the declaration of a structure includes an unnamed field of length 0, as shown in the following example, the memory layout is as shown in Figure 8.3.

```
struct Date
{
    unsigned nWeekDay  : 3;    // 0..7   (3 bits)
    unsigned nMonthDay : 6;    // 0..31  (6 bits)
    unsigned           : 0;    // Force alignment to next boundary.
    unsigned nMonth    : 5;    // 0..12  (5 bits)
    unsigned nYear     : 8;    // 0..100 (8 bits)
};
```

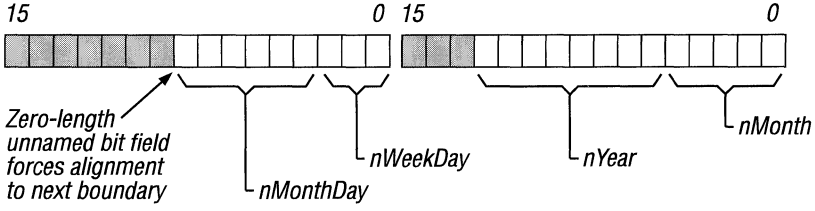


Figure 8.3 Layout of Date Object with Zero-Length Bit Field

The underlying type of a bit field must be an integral type, as described in "Fundamental Types" in Chapter 2, on page 50.

Restrictions on Use of Bit Fields

The following list details erroneous operations on bit fields:

- Taking the address of a bit field
- Declaring a pointer to a bit field
- Declaring a reference to a bit field

8.8 Nested Class Declarations

A class can be declared within the scope of another class. Such a class is called a "nested class." Nested classes are considered to be within the scope of the enclosing class and are available for use within that scope. To refer to a nested class from a scope other than its immediate enclosing scope, you must use a fully qualified name.

The following example shows how to declare nested classes:

```
class BufferedIO
{
public:
    enum IOError { None, Access, General };

    // Declare nested class BufferedInput.
    class BufferedInput
    {
    public:
        int read();
        int good() { return _inputerror == None; }
    private:
        IOError _inputerror;
    };

    // Declare nested class BufferedOutput.
    class BufferedOutput
    {
        // Member list
    };
};
```

`BufferedIO::BufferedInput` and `BufferedIO::BufferedOutput` are declared within `BufferedIO`. These class names are not visible outside the scope of class `BufferedIO`. However, an object of type `BufferedIO` does not contain any objects of types `BufferedInput` or `BufferedOutput`.

Nested classes can directly use names, type names, names of static members, and enumerators only from the enclosing class. To use names of other class members, you must use pointers, references, or object names.

In the preceding `BufferedIO` example, the enumeration `IOError` can be accessed directly by member functions in the nested classes, `BufferedIO::BufferedInput` or `BufferedIO::BufferedOutput`, as shown in function `good`.

Note Nested classes declare only types within class scope. They do not cause contained objects of the nested class to be created. The preceding example declares two nested classes, but does not declare any objects of these class types.

Access Privileges and Nested Classes

Nesting a class within another class does not give member functions of the nested class special access privileges. Similarly, member functions of the enclosing class have no special access to members of the nested class.

For more information about access privileges, see Chapter 10, "Member-Access Control."

Member Functions in Nested Classes

Member functions declared in nested classes can be defined in file scope. The preceding example could have been written:

```
class BufferedIO
{
public:
    enum IOError { None, Access, General };
    class BufferedInput
    {
    public:
        int read(); // Declare but do not define member
        int good(); //  functions read and good.
    private:
        IOError _inputerror;
    };

    class BufferedOutput
    {
        // Member list.
    };
};
```

```
// Define member functions read and good in
//  file scope.
int BufferedIO::BufferedInput::read()
{
    ...
}

int BufferedIO::BufferedInput::good()
{
    return _inputerror == None;
}
```

In the preceding example, the *qualified-type-name* syntax is used to declare the function name. The declaration:

```
BufferedIO::BufferedInput::read()
```

means "the `read` function that is a member of the `BufferedInput` class that is in the scope of the `BufferedIO` class." Because this declaration uses the *qualified-type-name* syntax, constructs of the following form are possible:

```
typedef BufferedIO::BufferedInput BIO_INPUT;

int BIO_INPUT::read()
```

The preceding declaration is equivalent to the previous one, but it uses a **typedef** name in place of the class names.

Friend Functions and Nested Classes

Friend functions declared in a nested class are considered to be in the scope of the nested class, not the enclosing class. Therefore, the friend functions gain no special access privileges to members or member functions of the enclosing class. If you want to use a name that is declared in a nested class in a friend function, and the friend function is defined in file scope, use qualified type names as follows:

```
extern char *rgszMessage[];

class BufferedIO
{
public:
    ...
```

```
    class BufferedInput
    {
    public:
        friend int GetExtendedErrorStatus();
        ...
        static char *message;
        int          iMsgNo;
    };
};
char *BufferedIO::BufferedInput::message;

int GetExtendedErrorStatus()
{
    ...
    strcpy( BufferedIO::BufferedInput::message,
            rgszMessage[iMsgNo] );
    return iMsgNo;
}
```

The preceding code shows the function `GetExtendedErrorStatus` declared as a friend function. In the function, which is defined in file scope, a message is copied from a static array into a class member. Note that a better implementation of `GetExtendedErrorStatus` is to declare it as:

```
int GetExtendedErrorStatus( char *message )
```

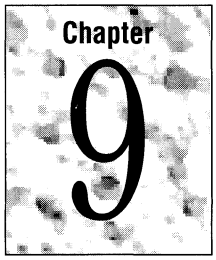
Using the preceding interface, several classes can use the services of this function by passing a memory location where they want the error message copied.

8.9 Type Names in Class Scope

Type names defined within class scope are considered local to their class. They cannot be used outside that class. The following example demonstrates this concept:

```
class Tree
{
public:
    typedef Tree * PTREE;
    PTREE  Left;
    PTREE  Right;
    void  *vData;
};

PTREE pTree;  // Error: not in class scope.
```

Derived Classes



This chapter explains how to use derived classes to produce extensible programs.

9.1 Overview

New classes can be derived from existing classes using a mechanism called "inheritance" (see the discussion beginning on this page). Classes that are used for derivation are called "base classes" of a particular derived class. A derived class is declared using the following syntax:

Syntax

base-spec:
  : *base-list*

base-list:
  base-specifier
  base-list **,** *base-specifier*

base-specifier:
  complete-class-name
  virtual *access-specifier*$_{\text{opt}}$ *complete-class-name*
  access-specifier **virtual**$_{\text{opt}}$ *complete-class-name*

access-specifier:
  private
  protected
  public

Single Inheritance

In "single inheritance" a common form of inheritance, classes have only one base class. Consider the relationship illustrated in Figure 9.1.

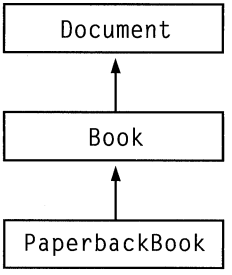


Figure 9.1 Simple Single Inheritance Graph

Note the progression from general to specific in Figure 9.1. Another common attribute found in the design of most class hierarchies is that the derived class has a "kind of" relationship with the base class. In Figure 9.1, a `Book` is a kind of a `Document`, and a `PaperbackBook` is a kind of a `book`.

One other item of note in Figure 9.1: `Book` is both a derived class (from `Document`) and a base class (`PaperbackBook` is derived from `Book`). A skeletal declaration of such a class hierarchy is shown in the following example:

```
class Document
{
    // Member list.
};

// Book is derived from Document.
class Book : public Document
{
    // Member list.
};

// PaperbackBook is derived from Book.
class PaperbackBook : public Book
{
    // Member list.
};
```

`Document` is considered a "direct base" class to `Book`; it is an "indirect base" class to `PaperbackBook`. The difference is that a direct base class appears in the base list of a class declaration; an indirect base does not.

Note that the base class from which each class is derived is completely declared before the declaration of the derived class. It is not sufficient to provide a forward-referencing declaration for a base class; it must be a complete declaration.

In the preceding example, the access specifier **public** is used. The meaning of public, protected, and private inheritance is described in Chapter 10, "Member-Access Control."

A class can serve as the base class for many more specific classes, as illustrated in Figure 9.2.

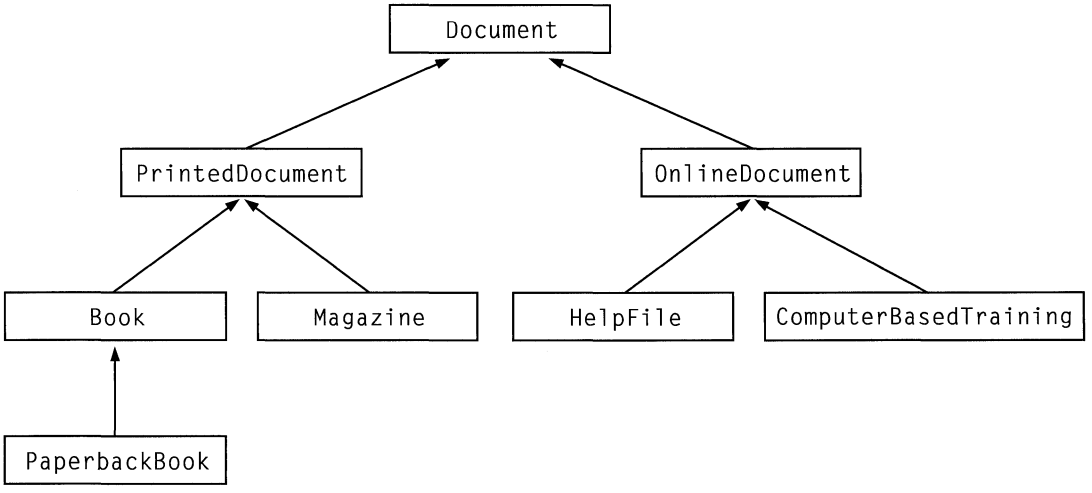


Figure 9.2 Sample of Directed Acyclic Graph

In the diagram in Figure 9.2, called a "directed acyclic graph" (or "DAG"), some of the classes are base classes for more than one derived class. However, the reverse is not true: There is only one direct base class for any given derived class. The graph in Figure 9.2 depicts a "single inheritance" structure.

Note Directed acyclic graphs are not unique to single inheritance. They are also used to depict multiple inheritance graphs. This topic is covered in "Multiple Inheritance" on page 264.

In inheritance, the derived class contains the members of the base class plus any new members you add. As a result, a derived class can refer to members of the base class (unless those members are redefined in the derived class). The scope-resolution operator (**::**) can be used to refer to members of direct or indirect base classes when those members have been redefined in the derived class. Consider this example:

```
class Document
{
public:
    char *Name;           // Document name.
    void  PrintNameOf();  // Print name.
};

// Implementation of PrintNameOf function from class Document.
void Document::PrintNameOf()
{
    cout << Name << endl;
}

class Book : public Document
{
public:
    Book( char *name, long pagecount );
private:
    long  PageCount;
};

// Constructor from class Book.
Book::Book( char *name, long pagecount )
{
    Name = new char[ strlen( name ) + 1 ];
    strcpy( Name, name );
    PageCount = pagecount;
};
```

Note that the constructor for `Book` (`Book::Book`), has access to the data member, `Name`. In a program, an object of type `Book` can be created and used as follows:

```
//  Create a new object of type Book. This invokes the
//   constructor Book::Book.
Book LibraryBook( "Programming Windows, 2nd Ed", 944 );

...

//  Use PrintNameOf function inherited from class Document.
LibraryBook.PrintNameOf();
```

As the preceding example demonstrates, class-member and inherited data and functions are used identically. If the implementation for class `Book` calls for a re-implementation of the `PrintNameOf` function, the function that belongs to the `Document` class can be called only by using the scope-resolution (`::`) operator:

```
class Book : public Document
{
    Book( char *name, long pagecount );
    void PrintNameOf();
    long  PageCount;
};
```

```
void Book::PrintNameOf()
{
    cout << "Name of book: ";
    Document::PrintNameOf();
}
```

Pointers and references to derived classes can be implicitly converted to pointers and references to their base classes if there is an accessible, unambiguous base class. The following code demonstrates this concept using pointers (the same principle applies to references):

```
#include <iostream.h>

main()
{
    Document *DocLib[10];  // Library of ten documents.

    for( int i = 0; i < 10; ++i )
    {
        cout << "Type of document: "
             << "P)aperback, M)agazine, H)elp File, C)BT"
             << endl;

        char cDocType;
        cin >> cDocType;

        switch( tolower( cDocType ) )
        {
        case 'p':
            DocLib[i] = new PaperbackBook;
            break;

        case 'm':
            DocLib[i] = new Magazine;
            break;

        case 'h':
            DocLib[i] = new HelpFile;
            break;

       case 'c':
            DocLib[i] = new ComputerBasedTraining;
            break;

       default:
            --i;
            break;
       }
    }
```

```
    for( i = 0; i < 10; ++i )
        DocLib->PrintNameOf();


    return 0;
}
```

In the `switch` statement in the preceding example, objects of different types are created, depending on what the user specified for `cDocType`. However, because these types are all derived from the `Document` class, there is an implicit conversion to `Document *`. As a result, `DocLib` is a "heterogeneous list" (a list in which not all objects are of the same type) containing different kinds of objects.

Because the `Document` class has a `PrintNameOf` function, it can print the name of each book in the library, although it may omit some of the information specific to the type of document (page count for `Book`, number of bytes for `HelpFile`, and so on).

Note Forcing the base class to implement a function such as `PrintNameOf` is often not the best design. Virtual functions, described in "Virtual Functions" on page 275, offer other design alternatives.

Multiple Inheritance

Later versions of C++ introduced a "multiple inheritance" model for inheritance. In a multiple inheritance graph, the relationships can be many-to-many between the base and derived classes, instead of one-to-many in single inheritance. Consider the graph in Figure 9.3.

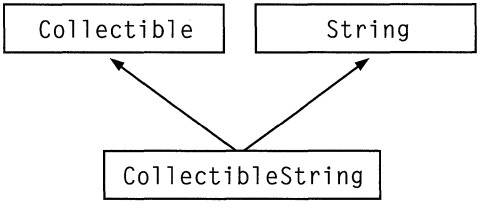


Figure 9.3 Simple Multiple-Inheritance Graph

The diagram in Figure 9.3 shows a class, `CollectibleString`. It is like a `Collectible` (something that can be contained in a collection), and it is like a `String`. Multiple inheritance is a good solution to this kind of problem (where a derived class has attributes of more than one base class) because it is easy to form a `CollectibleCustomer`, `CollectibleWindow`, and so on.

If the properties of either class are not required for a particular application, either class can be used alone or in combination with other classes. Therefore, given the hierarchy depicted in Figure 9.3 as a basis, you can form noncollectible strings and collectibles that are not strings. This flexibility is not possible using single inheritance.

Virtual Functions

Some class hierarchies are broad but have many things in common. The common code is implemented in a base class, while the specific code is in the derived classes.

It is important for the base classes to establish a protocol through which the derived classes can attain maximum functionality. These protocols are commonly implemented using virtual functions. Sometimes the base class provides a default implementation for such functions. In a class hierarchy such as the `Document` hierarchy in Figure 9.2, two useful functions are `Identify` and `WhereIs`.

When called, the `Identify` function returns a correct identification, appropriate to the kind of document: For a `Book`, a function call such as `doc->Identify()` must return the ISBN number; however, for a `HelpFile`, a product name and revision number is probably more appropriate. Similarly, `WhereIs` should return a row and shelf for a `Book`, but for a `HelpFile` it should return a disk location—perhaps a directory and filename.

It is important that all implementations of the `Identify` and `WhereIs` functions return the same kind of information. In this case, a character string is appropriate.

These functions are implemented as virtual functions, then invoked using a pointer to a base class. The binding to the actual code occurs at run time, selecting the correct `Identify` or `WhereIs` function.

Abstract Classes

Classes can be implemented to enforce a protocol. These classes are called "abstract classes" because no object of the class type can be created. They exist solely for derivation.

Classes are abstract classes if they contain pure virtual functions or if they inherit pure virtual functions and do not provide an implementation for them. Pure virtual functions are virtual functions declared with the *pure-specifier* (`= 0`) as follows:

```
virtual char *Identify() = 0;
```

The base class, `Document`, might impose the following protocol on all derived classes:

- An appropriate `Identify` function must be implemented.
- An appropriate `WhereIs` function must be implemented.

By specifying such a protocol when designing the `Document` class, the class designer can be assured that no nonabstract class can be implemented without `Identify` and `WhereIs` functions. The `Document` class, therefore, contains these declarations:

```
class Document
{
public:
    ...
    //  Requirements for derived classes: They must implement
    //   these functions.
    virtual char *Identify() = 0;
    virtual char *WhereIs() = 0;
    ...
};
```

Base Classes

As discussed previously, the inheritance process causes a new derived class to be created that is made up of the members of the base class(es) plus any new members the derived class adds. In a multiple-inheritance situation, it is possible to construct an inheritance graph where the same base class is part of more than one of the derived classes. Figure 9.4 shows such a graph.

In Figure 9.4, pictorial representations of the components of `CollectibleString` and `CollectibleSortable` are shown. However, the base class, `Collectible`, is in `CollectibleSortableString` through the `CollectibleString` path and the `CollectibleSortable` path. To eliminate this kind of redundancy, such classes can be declared as virtual base classes when they are inherited.

For information about declaring virtual base classes and how objects with virtual base classes are composed, see "Virtual Base Classes" on page 268.

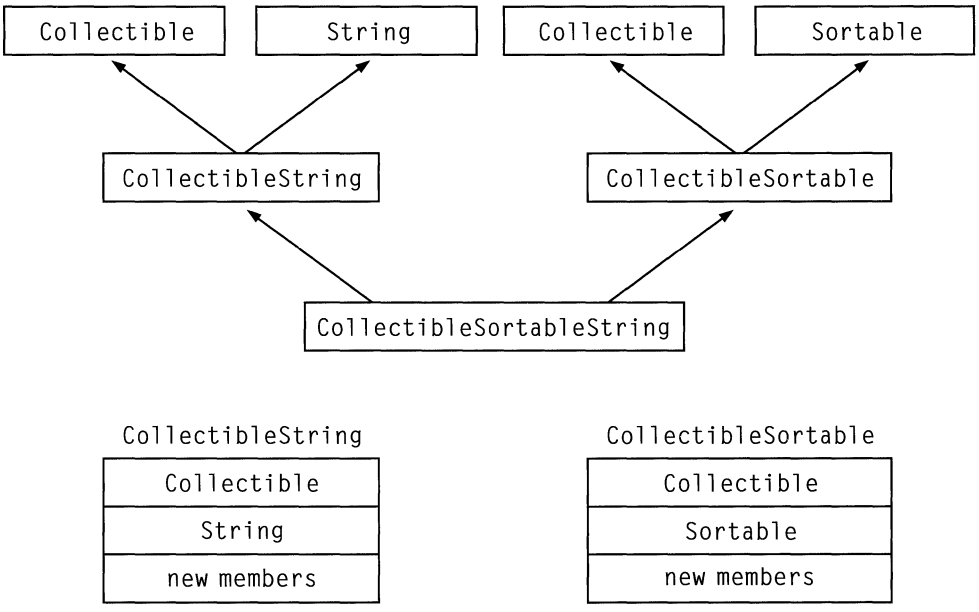


Figure 9.4 Multiple Instances of a Single Base Class

9.2 Multiple Base Classes

As described in "Multiple Inheritance" on page 264, a class can be derived from more than one base class. In a multiple-inheritance model (where classes are derived from more than one base class), the base classes are specified using the *base-list* grammar element (see "Syntax" in "Overview" on page 259). For example, the class declaration for `CollectionOfBook`, derived from `Collection` and `Book`, can be specified:

```
class CollectionOfBook : public Book, public Collection
{
    // New members
};
```

The order in which base classes are specified is not significant except in certain cases where constructors and destructors are invoked. In these cases, the order in which base classes are specified affects the following:

- The order in which initialization by constructor takes place. If your code relies on the `Book` portion of `CollectionOfBook` to be initialized before the `Collection` part, the order of specification is significant. Initialization takes place in the order the classes are specified in the *base-list.*
- The order in which destructors are invoked to clean up. Again, if a particular "part" of the class must be present when the other part is being destroyed, the order is significant. Destructors are called in the reverse order of the classes specified in the *base-list.*

Note The order of specification of base classes can affect the memory layout of the class, but it often does not. Do not make any programming decisions based on the order of base members in memory.

When specifying the *base-list*, you cannot specify the same class name more than once. However, it is possible for a class to be an indirect base to a derived class more than once.

Virtual Base Classes

Because a class can be an indirect base class to a derived class more than once, C++ provides a way to optimize the way such base classes work. Consider the class hierarchy in Figure 9.5, which illustrates a simulated lunch line.

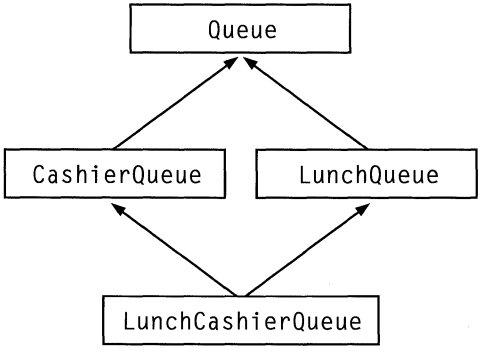


Figure 9.5 Simulated Lunch-Line Graph

In Figure 9.5, `Queue` is the base class for both `CashierQueue` and `LunchQueue`. However, when both classes are combined to form `LunchCashierQueue`, the following problem arises: the new class contains two subobjects of type `Queue`; one from `CashierQueue` and the other from `LunchQueue`. Figure 9.6 shows the conceptual memory layout (the actual memory layout might be optimized).

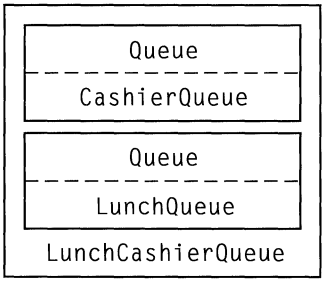


Figure 9.6 Simulated Lunch-Line Object

Note that there are two `Queue` subobjects in the `LunchCashierQueue` object. The following code declares `Queue` to be a virtual base class:

```
class Queue
{
    // Member list
};

class CashierQueue : virtual public Queue
{
    // Member list
};

class LunchQueue : virtual public Queue
{
    // Member list
};

class LunchCashierQueue : public LunchQueue, public CashierQueue
{
    // Member list
};
```

The `virtual` keyword ensures that only one copy of the subobject `Queue` is included (see Figure 9.7).

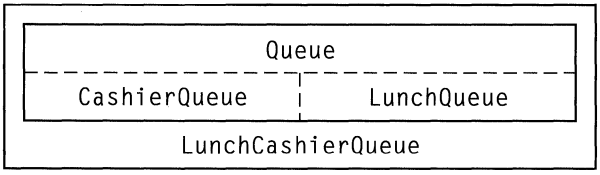


Figure 9.7 Simulated Lunch-Line Object with Virtual Base Classes

A class can have both a virtual and a nonvirtual component of a given type. This happens in the conditions illustrated in Figure 9.8.

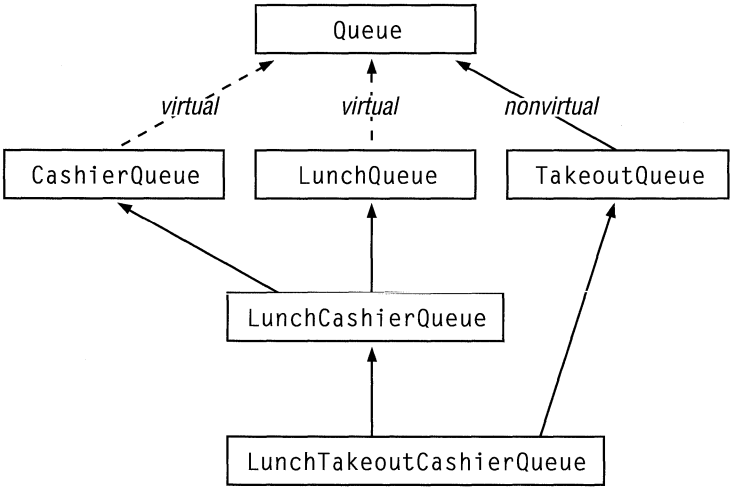


Figure 9.8 Virtual and Nonvirtual Components of the Same Class

In Figure 9.8, `CashierQueue` and `LunchQueue` use `Queue` as a virtual base class. However, `TakeoutQueue` specifies `Queue` as a base class, not a virtual base class. Therefore, `LunchTakeoutCashierQueue` has two subobjects of type `Queue`: one from the inheritance path that includes `LunchCashierQueue`, and one from the path that includes `TakeoutQueue`. This is illustrated in Figure 9.9.

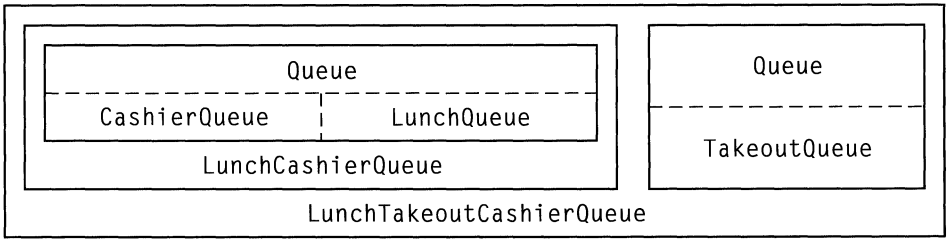


Figure 9.9 Object Layout with Virtual and Nonvirtual Inheritance

Note Virtual inheritance provides significant size benefits when compared with nonvirtual inheritance. However, it can introduce extra processing overhead.

Name Ambiguities

Multiple inheritance introduces the possibility for names to be inherited along more than one path. The class-member names along these paths are not necessarily unique. These name conflicts are called "ambiguities."

Any expression that refers to a class member must make an unambiguous reference. The following example shows how ambiguities develop:

```
// Declare two base classes, A and B.
class A
{
public:
    unsigned a;
    unsigned b();
};

class B
{
public:
    unsigned a();  // Note that class A also has a member "a"
    int b();       //  and a member "b".
    char c;
};

// Define class C as derived from A and B.
class C : public A, public B
{
};
```

Given the preceding class declarations, code such as the following is ambiguous:

```
C *pc = new C;

pc->b();
```

Consider the preceding example. Because the name `a` is a member of both class `A` and class `B`, the compiler cannot discern which `a` designates the function to be called. Access to a member is ambiguous if it can refer to more than one function, object, type, or enumerator.

The compiler detects ambiguities by performing tests in this order:

1. If access to the name is ambiguous (as just described), an error message is generated.
2. If overloaded functions are unambiguous, they are resolved. (For more information about function overloading ambiguity, see "Argument Matching" in Chapter 12, on page 344.)
3. If access to the name violates member-access permission, an error message is generated. (For more information, see Chapter 10, "Member-Access Control".)

When an expression produces an ambiguity through inheritance, you can manually resolve it by qualifying the name in question with its class name. To make the preceding example compile properly with no ambiguities, use code such as:

```
C *pc = new C;

pc->B::a();
```

Note The potential ambiguity introduced by the class declarations of classes `A` and `B` in the example shown on the previous page does not cause an error; the ambiguous accessing of a member causes the error.

Ambiguities and Virtual Base Classes

If virtual base classes are used, functions, objects, types, and enumerators can be reached through multiple-inheritance paths. Because there is only one instance of the base class, there is no ambiguity when accessing these names.

Figure 9.10 shows how objects are composed using virtual and nonvirtual inheritance.

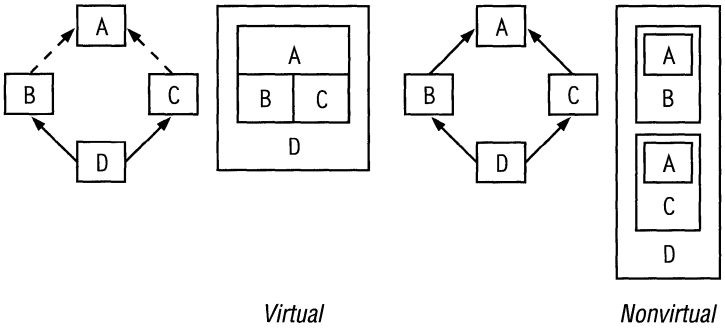


Figure 9.10 Virtual vs. Nonvirtual Derivation

In Figure 9.10, accessing any member of class `A` through nonvirtual base classes causes an ambiguity; the compiler has no information that explains whether to use the subobject associated with `B` or the subobject associated with `C`. However, when `A` is specified as a virtual base class, there is no question which subobject is being accessed.

Dominance

It is possible for more than one name (function, object, or enumerator) to be reached through an inheritance graph. Such cases are considered ambiguous with nonvirtual base classes. They are also ambiguous with virtual base classes, unless one of the names "dominates" the others.

A name dominates another name if it is defined in both classes, and one class is derived from the other. The dominant name is the name in the derived class; this name is used when an ambiguity would otherwise have arisen, as shown in the following example:

```
class A
{
public:
    int a;
};

class B : public virtual A
{
public:
    int a();
};

class C : public virtual A
{
   ...
};

class D : public B, public C
{
public:
    D() { a(); } // Not ambiguous. B::a() dominates A::a.
};
```

Ambiguous Conversions

Explicit and implicit conversions from pointers or references to class types can cause ambiguities. Figure 9.11 shows the following:

- The declaration of an object of type `D`.
- The effect of applying the address-of operator (**&**) to that object. Note that the address-of operator always supplies the base address of the object.
- The effect of explicitly converting the pointer obtained using the address-of operator to the base-class type `A`. Note that coercing the address of the object to type `A*` does not always provide the compiler with enough information as to which subobject of type `A` to select; in this case, two subobjects exist.

The conversion to type `A*` (pointer to `A`) is ambiguous because there is no way to discern which subobject of type `A` is the correct one. Note that you can avoid the ambiguity by explicitly specifying which subobject you mean to use, as follows:

```
(A *)(B *)&d        // Use B subobject.
(A *)(C *)&d        // Use C subobject.
```

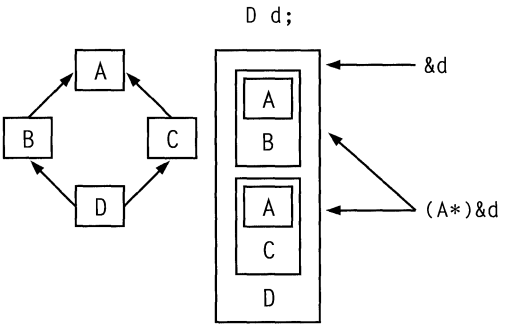


Figure 9.11 Ambiguous Conversion of Pointers to Base Classes

9.3 Virtual Functions

"Virtual functions" are functions that ensure the correct function is called for an object, regardless of the expression used to make the function call.

Suppose a base class contains a function declared as **virtual** and a derived class defines the same function. The function from the derived class is invoked for objects of the derived class, even if it is called using a pointer or reference to the base class. The following example shows a base class that provides an implementation of the `PrintBalance` function:

```
class Account
{
public:
                    Account( double d );  // Constructor.
    virtual double GetBalance();          // Obtain balance.
    virtual void   PrintBalance();        // Default implementation.
private:
    double _balance;
};

// Implementation of constructor for Account.
double Account::Account( double d )
{
    _balance = d;
}
```

```
// Implementation of GetBalance for Account.
double Account::GetBalance()
{
    return _balance;
}

// Default implementation of PrintBalance.
void Account::PrintBalance()
{
    cerr << "Error. Balance not available for base type."
         << endl;
}
```

Two derived classes, `CheckingAccount` and `SavingsAccount`, can be created as follows:

```
class CheckingAccount : public Account
{
public:
    void PrintBalance();
};

// Implementation of PrintBalance for CheckingAccount.
void CheckingAccount::PrintBalance()
{
    cout << "Checking account balance: " << GetBalance();
}

class SavingsAccount : public Account
{
public:
    void PrintBalance();
};

// Implementation of PrintBalance for SavingsAccount.
void SavingsAccount::PrintBalance()
{
    cout << "Savings account balance: " << GetBalance();
}
```

The `PrintBalance` function in the derived classes is virtual because it is declared as virtual in the base class, `Account`. To call virtual functions such as `PrintBalance`, code such as the following can be used:

```
// Create objects of type CheckingAccount and SavingsAccount.
CheckingAccount *pChecking = new CheckingAccount( 100.00 );
SavingsAccount  *pSavings  = new SavingsAccount( 1000.00 );

// Call PrintBalance using a pointer to Account.
Account *pAccount = pChecking;
pAccount->PrintBalance();

// Call PrintBalance using a pointer to Account.
pAccount = pSavings;
pAccount->PrintBalance();
```

In the preceding code, the calls to `PrintBalance` are identical, except for the object `pAccount` points to. Because `PrintBalance` is virtual, the version of the function defined for each object is called. The `PrintBalance` function in the derived classes `CheckingAccount` and `SavingsAccount` "override" the function in the base class `Account`.

If a class is declared that does not provide an overriding implementation of the `PrintBalance` function, the default implementation from the base class `Account` is used.

Functions in derived classes override virtual functions in base classes only if their type is the same. A function in a derived class cannot differ from a virtual function in a base class in its return type only; the argument list must differ as well.

When calling a function using pointers or references, the following rules apply:

- A call to a virtual function is resolved according to the underlying type of object for which it is called.
- A call to a nonvirtual function is resolved according to the type of the pointer or reference.

The following example shows how virtual and nonvirtual functions behave when called through pointers:

```
#include <iostream.h>

// Declare a base class.
class Base
{
public:
    virtual void NameOf();          // Virtual function.
            void InvokingClass();   // Nonvirtual function.
};

// Implement the two functions.
void Base::NameOf()
{
    cout << "Base::NameOf\n";
}

void Base::InvokingClass()
{
    cout << "Invoked by Base\n";
}

// Declare a derived class.
class Derived : public Base
{
public:
    void NameOf();          // Virtual function.
    void InvokingClass();   // Nonvirtual function.
};

// Implement the two functions.
void Derived::NameOf()
{
    cout << "Derived::NameOf\n";
}

void Derived::InvokingClass()
{
    cout << "Invoked by Derived\n";
}

void main()
{
    // Declare an object of type Derived.
    Derived aDerived;
    // Declare two pointers, one of type Derived * and the other
    //  of type Base *, and initialize them to point to aDerived.
    Derived *pDerived = &aDerived;
    Base    *pBase    = &aDerived;
```

```
    // Call the functions.
    pBase->NameOf();           // Call virtual function.
    pBase->InvokingClass();    // Call nonvirtual function.
    pDerived->NameOf();        // Call virtual function.
    pDerived->InvokingClass(); // Call nonvirtual function.
}
```

The output from this program is:

```
Derived::NameOf
Invoked by Base
Derived::NameOf
Invoked by Derived
```

Note that regardless of whether the `NameOf` function is invoked through a pointer to `Base` or a pointer to `Derived`, it calls the function for `Derived`. It calls the function for `Derived` because `NameOf` is a virtual function, and both `pBase` and `pDerived` point to an object of type `Derived`.

Because virtual functions are called only for objects of class types, you cannot declare global or static functions as **virtual**. However, a virtual function can be declared as a friend in another class.

The **virtual** keyword can be used when declaring overriding functions in a derived class, but it is optional; these functions are always virtual.

Virtual functions in a base class must be defined unless they are declared using the *pure-specifier*. (For more information about pure virtual functions, see "Abstract Classes" on page 265.)

The virtual function-call mechanism can be suppressed by explicitly qualifying the function name using the scope-resolution operator (**::**). Consider the preceding example. To call `PrintBalance` in the base class, use code such as the following:

```
CheckingAccount *pChecking = new CheckingAccount( 100.00 );

pChecking->Account::PrintBalance();  //  Explicit qualification.

Account *pAccount = pChecking;

pAccount->Account::PrintBalance();   //  Explicit qualification.
```

Both calls to `PrintBalance` in the preceding example suppress the virtual function-call mechanism.

9.4 Abstract Classes

Abstract classes act as expressions of general concepts from which more specific classes can be derived. You cannot create an object of an abstract class type; however, you can use pointers and references to abstract class types.

A class that contains at least one pure virtual function is considered an abstract class. Classes derived from the abstract class must implement the pure virtual function or they, too, are abstract classes.

A virtual function is declared as "pure" by using the *pure-specifier* syntax (described in "Abstract Classes" on page 265). Consider the example presented in "Virtual Functions" on page 265. The intent of class `Account` is to provide general functionality, but objects of type `Account` have insufficient specificity to be meaningful. Therefore, `Account` is a good candidate for an abstract class:

```
class Account
{
public:
                    Account( double d );  // Constructor.
    virtual double GetBalance();          // Obtain balance.
    virtual void   PrintBalance() = 0;    // Pure virtual function.
private:
    double _balance;
};
```

The only difference between this declaration and the previous one is that `PrintBalance` is declared with the pure specifier (`= 0`).

Restrictions on Using Abstract Classes

Abstract classes cannot be used for:

- Variables or member data
- Argument types
- Function return types
- Types of explicit conversions

Another restriction is that if the constructor for an abstract class calls a pure virtual function, either directly or indirectly, the result is undefined. However, constructors for abstract classes can call other member functions.

Pure virtual functions can be defined for abstract classes, but they can be called directly only by using the syntax:

abstract-class-name::*function-name*( )

An implementation can be provided for pure virtual functions. This helps when designing class hierarchies whose base class(es) include pure virtual destructors, because base class destructors are always called in the process of destroying an object. Consider the following example:

```
#include <iostream.h>

// Declare an abstract base class with a pure virtual destructor.
class base
{
public:
    base() {}
    virtual ~base()=0;
};

// Provide a definition for destructor.
base::~base()
{
}

class derived:public base
{
public:
    derived() {}
    ~derived(){}
};

void main()
{
    derived *pDerived = new derived;

    delete pDerived;
}
```

When the object pointed to by `pDerived` is deleted, the destructor for class `derived` is called, then the destructor for class `base` is called. The empty implementation for the pure virtual function ensures that at least some implementation exists for the function.

Note In the example above, the pure virtual function `base::~base` is called implicitly from `derived::~derived`. It is also possible to call pure virtual functions explicitly using a fully qualified member-function name.

9.5 Summary of Scope Rules

This section supplements "Scope" in Chapter 2, on page 28 by adding the concepts pertaining to classes.

Ambiguity

The use of a name must be unambiguous within its scope (up to the point where overloading is determined). If the name denotes a function, the function must be unambiguous with respect to number and type of arguments. If the name remains unambiguous, member-access rules are applied. (Member-access control is described in Chapter 10.)

Global Names

A name of an object, function, or enumerator is global if it is introduced outside any function or class or prefixed by the global unary scope operator (::), and it is not used in conjunction with any of these binary operators:

- Scope-resolution (::)
- Member-selection for objects and references (.)
- Member-selection for pointers (->)

Names and Qualified Names

Names used with the binary scope-resolution operator (::) are called "qualified names." The name specified after the binary scope-resolution operator must be a member of the class specified on the left of the operator or a member of its base class(es).

Names specified after the member-selection operator (. or ->) must be members of the class type of the object specified on the left of the operator or members of its base class(es). Names specified on the right of the member-selection operator (->) can also be objects of another class type, provided that class defines an overloaded member-selection operator (->) that evaluates to a pointer to the original class type. (This provision is discussed in more detail in "Class Member Access" in Chapter 12, on page 363.)

The compiler searches for names in the following order:

1. Current block scope if name is used inside a function; otherwise global scope.
2. If the name is not found in the current block scope, the compiler searches outward through each enclosing block scope, including the outermost function scope (which includes function arguments).
3. If the name is still not found, and the name is used inside a member function, class's scope is searched for the name.
4. If the name is still not found, the class's base classes are searched for the name.
5. If the name is still not found, the enclosing nested class scope (if any) and its bases are searched. The search continues until the outermost enclosing class scope is searched.
6. If the name is still not found, global scope is searched.

However, you can make modifications to this search order as follows:

1. Names preceded by **::** force the search to begin at global scope.
2. Names preceded by the **class**, **struct**, and **union** keywords force the compiler to search only for **class**, **struct**, or **union** names.
3. Names on the left side of the scope-resolution operator (**::**) can be only **class**, **struct**, or **union** names.

If the name refers to a nonstatic member but is used in a static member function, an error message is generated. If the name refers to any nonstatic member in an enclosing class, an error message is generated because enclosed classes do not have enclosing-class **this** pointers.

Function Argument Names

Function argument names in function definitions are considered to be in the scope of the outermost block of the function. Therefore, they are local names and go out of scope when the function is exited.

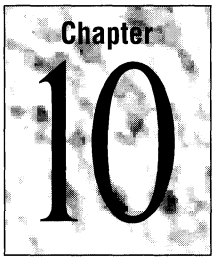
Function argument names in function declarations (prototypes) are in local scope of the declaration and go out of scope at the end of the declaration.

Default arguments are in the scope of the argument for which they are the default, as described in the preceding two paragraphs. However, they cannot access local variables or nonstatic class members. Default arguments are evaluated at the point of the function call, but they are evaluated in the function declaration's original scope. Therefore, the default arguments for member functions are always evaluated in class scope.

Constructor Initializers

Constructor initializers (described in "Initializing Bases and Members" in Chapter 11, on page 329) are evaluated in the scope of the outermost block of the constructor for which they are specified. Therefore, they can use the constructor's argument names.

Member-Access Control



C++ allows programmers to specify the level of access to member data and functions. There are three levels of access: public, protected, and private. This chapter explains how access control applies to objects of class type and to derived classes.

10.1 Controlling Access to Class Members

As a programmer, you can increase the integrity of software built with C++ by controlling access to class member data and functions. Class members can be declared as having private, protected, or public access, as shown in Table 10.1.

Table 10.1 Member-Access Control

Type of Access	Meaning
private	Class members declared as **private** can be used only by member functions and friends (classes or functions) of the class.
protected	Class members declared as **protected** can be used by member functions and friends (classes or functions) of the class. Additionally, they can be used by classes derived from the class.
public	Class members declared as **public** can be used by any function.

Access control prevents you from using objects in ways they were not intended to be used. This protection is lost when explicit type conversions (casts) are performed.

Note Access control is equally applicable to all names: member functions, member data, nested classes, and enumerators.

The default access to class members (members of a class type declared using the **class** keyword) is private; to change the access in a class declaration, you must use the **public** or **protected** keyword. The default access to **struct** and **union** members is public; to change the access in a **struct** or **union** declaration, you must use the **private** or **protected** keyword.

10.2 Access Specifiers

In class declarations, members can have access specifiers.

Syntax

access-specifier **:** *member-list*$_{opt}$

The *access-specifier* determines the access to the names that follow it, up to the next *access-specifier* or the end of the class declaration. Figure 10.1 illustrates this concept.

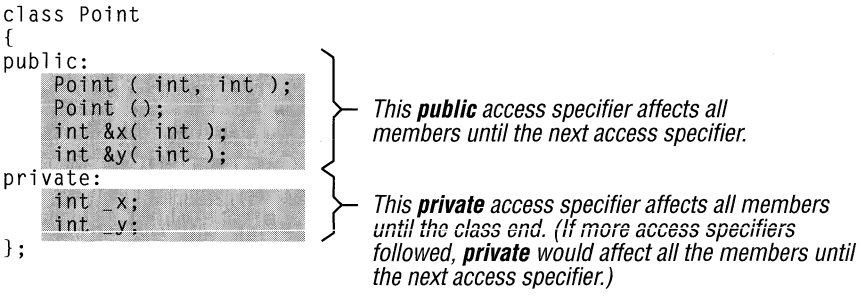


Figure 10.1 Access Control in Classes

Although only two access specifiers are shown in Figure 10.1, there is no limit to the number of access specifiers in a given class declaration. For example, the Point class in Figure 10.1 could just as easily be declared using multiple access specifiers as follows:

```
class Point
{
public:                  // Declare public constructor.
    Point( int, int );
private:                 // Declare private state variable.
    int _x;
public:                  // Declare public constructor.
    Point();
public:                  // Declare public accessor.
    int &x( int );
private:                 // Declare private state variable.
    int _y;
public:                  // Declare public accessor.
    int &y( int );
};
```

Note that there is no specific order required for member access, as shown in the example above. The allocation of storage for objects of class types is implementation dependent, but members are guaranteed to be assigned successively higher memory addresses between access specifiers.

10.3 Access Specifiers for Base Classes

Two factors control which members of a base class are accessible in a derived class; these same factors control access to the inherited members in the derived class:

- Whether the derived class declares the base class using the **public** access specifier in the *class-head* (*class-head* is described in "Syntax" in Chapter 8, on page 228).
- What the access to the member is in the base class.

Table 10.2 shows the interaction between these factors.

Table 10.2 Determining Base-Class Member Access

Member Access in Base Class		
private	protected	**public**
Inaccessible in derived class if you use private derivation	Private in derived class if you use private derivation	Private in derived class if you use private derivation
Inaccessible in derived class if you use protected derivation	Protected in derived class if you use protected derivation	Protected in derived class if you use protected derivation
Inaccessible in derived class if you use public derivation	Protected in derived class if you use public derivation	Public in derived class if you use public derivation

The following example illustrates this:

```
class BaseClass
{
public:
    int PublicFunc();    // Declare a public member.
protected:
    int ProtectedFunc(); // Declare a protected member.
private:
    int PrivateFunc();   // Declare a private member.
};

// Declare two classes derived from BaseClass.
class DerivedClass1 : public BaseClass
{ };

class DerivedClass2 : private BaseClass
{ };
```

In `DerivedClass1`, the member function `PublicFunc` is a public member and `ProtectedFunc` is a protected member because `BaseClass` is a public base class. `PrivateFunc` is private to `BaseClass`, and it is inaccessible to any derived classes.

In `DerivedClass2`, the functions `PublicFunc` and `ProtectedFunc` are considered private members because `BaseClass` is a private base class. Again, `PrivateFunc` is private to `BaseClass`, and it is inaccessible to any derived classes.

You can declare a derived class without a base-class access specifier. In such a case, the derivation is considered private if the derived class declaration uses the **class** keyword. The derivation is considered public if the derived class declaration uses the **struct** keyword. For example, the following code:

```
class Derived : Base
...
```

is equivalent to:

```
= class Derived : private Base
...
```

Similarly, the following code:

```
struct Derived : Base
...
```

is equivalent to:

```
struct Derived : public Base
...
```

Note that members declared as having private access are not accessible to functions or derived classes unless those functions or classes are declared using the **friend** declaration in the base class.

A **union** type cannot have a base class.

Note When specifying a private base class, it is advisable to explicitly use the **private** keyword so users of the derived class understand the member access.

Access Control and Static Members

When you specify a base class as **private**, it affects only nonstatic members. Public static members are still accessible in the derived classes. However, accessing members of the base class using pointers, references, or objects can require a conversion, at which time access control is again applied. Consider the following example:

```
class Base
{
public:
    int Print();             // Nonstatic member.
    static int CountOf();    // Static member.
};

// Derived1 declares Base as a private base class.
class Derived1 : private Base
{
};
```

```
// Derived2 declares Derived1 as a public base class.
class Derived2 : public Derived1
{
    int ShowCount();    // Nonstatic member.
};

// Define ShowCount function for Derived2.
int Derived2::ShowCount()
{
   // Call static member function CountOf explicitly.
    int cCount = Base::CountOf();     // OK.

   // Call static member function CountOf using pointer.
    cCount = this->CountOf();  // Error. Conversion of
                               //  Derived2 * to Base * not
                               //  permitted.
    return cCount;
}
```

In the preceding code, access control prohibits conversion from a pointer to `Derived2` to a pointer to `Base`. The **this** pointer is implicitly of type `Derived2 *`. To select the `CountOf` function, **this** must be converted to type `Base *`. Such a conversion is not permitted because `Base` is a private indirect base class to `Derived2`. Conversion to a private base class type is acceptable only for pointers to immediate derived classes. Therefore, pointers of type `Derived1 *` can be converted to type `Base *`.

Note that calling the `CountOf` function explicitly, without using a pointer, reference, or object to select it, implies no conversion. Therefore, the call is allowed.

Members and friends of a derived class, *T*, can convert a pointer to *T* to a pointer to a private direct base class of *T*.

10.4 Friends

In some circumstances, it is more convenient to grant member-level access to functions that are not members of a class or to all functions in a separate class. The

friend keyword allows programmers to designate either the specific functions or the classes whose functions can access not only **public** members, but also **protected** and **private** members.

Friend Functions

Friend functions are not considered class members; they are normal external functions that are given special access privileges. Friends are not in the class's scope, and they are not called using the member-selection operators (. and –>) unless they are members of another class. The following example shows a `Point` class and an overloaded operator, `operator+`. (This example primarily illustrates friends, not overloaded operators. For more information about overloaded operators, see "Overloaded Operators" in Chapter 12, on page 351.)

```
#include <iostream.h>

// Declare class Point.
class Point
{
public:
    // Constructors
    Point() { _x = _y = 0; }
    Point( unsigned x, unsigned y ) { _x = x; _y = y; }

    // Accessors
    unsigned x() { return _x; }
    unsigned y() { return _y; }
    void     Print() { cout << "Point(" << _x << ", " << _y << ")"
                             << endl; }

    // Friend function declarations
    friend Point operator+( Point& pt, int nOffset );
    friend Point operator+( int nOffset, Point& pt );

private:
    unsigned _x;
    unsigned _y;
};
```

```
// Friend-function definitions
//
// Handle Point + int expression.
Point operator+( Point& pt, int nOffset )
{
    Point ptTemp = pt;

    // Change private members _x and _y directly.
    ptTemp._x += nOffset;
    ptTemp._y += nOffset;

    return ptTemp;
}

// Handle int + Point expression.
Point operator+( int nOffset, Point& pt )
{
    Point ptTemp = pt;

    // Change private members _x and _y directly.
    ptTemp._x += nOffset;
    ptTemp._y += nOffset;

    return ptTemp;
}

// Test overloaded operator.
main()
{
    Point pt( 10, 20 );
    pt.Print();

    pt = pt + 3;      // Point + int
    pt.Print();

    pt = 3 + pt;      // int + Point
    pt.Print();

    return 0;
}
```

When the expression `pt + 3` is encountered in the `main` function, the compiler determines if an appropriate user-defined `operator+` exists. In this case, the function `operator+( Point pt, int nOffset )` matches the operands, and a call to

the function is issued. In the second case (the expression `3 + pt`), the function `operator+( Point pt, int nOffset )` matches the supplied operands. Therefore, supplying these two forms of `operator+` preserves the commutative properties of the `+` operator.

A user-defined `operator+` can be written as a member function, but it takes only one argument: the value to be added to the object. As a result, the commutative properties of addition cannot be correctly implemented with member functions; they must use **friend** functions instead.

Notice that both versions of the overloaded `operator+` function are declared as friends in class `Point`. Both declarations are necessary—when friend declarations name overloaded functions or operators, only the particular functions specified by the argument types become friends. Suppose a third `operator+` function were declared as follows:

```
Point &operator+( Point &pt, Point &pt );
```

The `operator+` function in the preceding example is not a friend of class `Point` simply because it has the same name as two other functions that are declared as friends.

Because friend declarations are unaffected by access specifiers, they can be declared in any section of the class declaration.

Class Member Functions and Classes as Friends

Class member functions can be declared as friends in other classes. Consider the following example:

```
class A
{
private:
    int _a;
    friend int B::Func1( A );   //  Grant friend access to one
                                //   function in class B.
};

class B
{
public:
    int Func1( A a ) { return a._a; } //  OK: this is a friend.
    int Func2( A a ) { return a._a; } //  Error: _a is a private
                                      //   member.
};
```

In the preceding example, only the function `B::Func1( A )` is granted friend access to class `A`. Therefore, access to the private member `_a` is correct in function `b` of class `B`, but not in function `c`.

Suppose the friend declaration in class `A` had been:

```
friend class B;
```

In that case, all member functions in class `B` would have been granted friend access to class `A`. Note that "friendship" cannot be inherited, nor is there any "friend of a friend" access. Figure 10.2 shows four class declarations: `Base`, `Derived`, `aFriend`, and `anotherFriend`. Only class `aFriend` has direct access to the private members of `Base` (and to any members `Base` might have inherited).

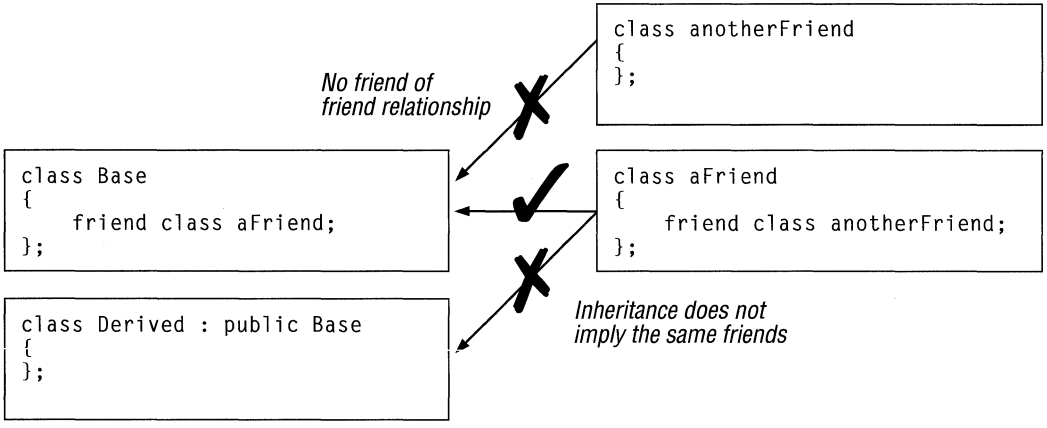


Figure 10.2 Implications of friend Relationship

Friend Declarations

If you declare a friend function that was not previously declared, that function is exported to the enclosing nonclass scope.

Functions declared in a friend declaration are treated as if they had been declared using the **extern** keyword. (For more information about **extern**, see "Static Storage-Class Specifiers" in Chapter 6, on page 158.)

While functions with global scope can be declared as friends prior to their prototypes, member functions cannot be declared as friends before the appearance of their complete class declaration. The following code shows why this fails:

```
class ForwardDeclared;   // Class name is known.

class HasFriends
{
    friend int ForwardDeclared::IsAFriend();   // Error.
};
```

The preceding example enters the class name `ForwardDeclared` into scope, but the complete declaration—specifically, the portion that declares the function `IsAFriend`—is not known. Therefore, the friend declaration in class `HasFriends` generates an error.

To declare two classes that are friends of one another, the entire second class must be specified as a friend of the first class. The reason for this restriction is that the compiler has enough information to declare individual friend functions only at the point where the second class is declared.

Note Although the entire second class must be a friend to the first class, you can select which functions in the first class will be friends of the second class.

Defining Friend Functions In Class Declarations

Friend functions can be defined inside class declarations. These functions are inline functions, and like member inline functions they behave as though they were defined immediately after all class members have been seen but before the class scope is closed (the end of the class declaration).

Friend functions defined inside class declarations are not considered in the scope of the enclosing class; they are in file scope.

10.5 Protected Member Access

Class members declared as **protected** can be used only by the following:

- Member functions of the class that originally declared these members.
- Friends of the class that originally declared these members.
- Classes publicly derived from the class that originally declared these members.

Protected members are not as private as **private** members, which are accessible only to members of the class in which they are declared, but they are not as public as **public** members, which are accessible in any function.

Protected members that are also declared as **static** are accessible to any friend or member function of a derived class. Protected members that are not declared as **static** are accessible to friends and member functions in a derived class only through a pointer to, reference to, or object of the derived class.

10.6 Access to Virtual Functions

The access control applied to virtual functions is determined by the type used to make the function call. Overriding declarations of the function do not affect the access control for a given type. For example:

```
class VFuncBase
{
public:
    virtual int GetState() { return _state; }
protected:
    int _state;
};

class VFuncDerived : public VFuncBase
{
private:
    int GetState() { return _state; }
};

...

VFuncDerived vfd;              // Object of derived type.
VFuncBase *pvfb = &vfd;        // Pointer to base type.
VFuncDerived *pvfd = &vfd;     // Pointer to derived type.
int State;

State = pvfb->GetState();      // GetState is public.
State = pvfd->GetState();      // GetState is private; error.
```

In the preceding example, calling the virtual function `GetState` using a pointer to type `VFuncBase` calls `VFuncDerived::GetState`, and `GetState` is treated as public. However, calling `GetState` using a pointer to type `VFuncDerived` is an access-control violation because `GetState` is declared private in class `VFuncDerived`.

Warning The virtual function `GetState` can be called using a pointer to the base class `VFuncBase`. This does not mean that the function called is the base-class version of that function.

10.7 Multiple Access

In multiple-inheritance lattices involving virtual base classes, a given name can be reached through more than one path. Because different access control can be applied along these different paths, the compiler chooses the path that gives the most access. See Figure 10.3.

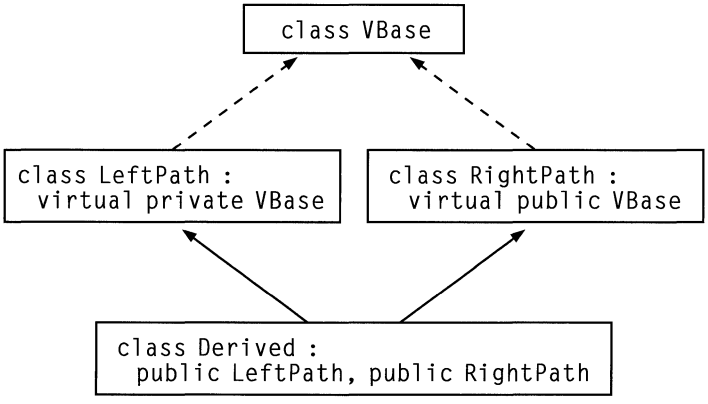
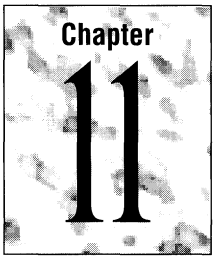


Figure 10.3 Access Along Paths of an Inheritance Graph

In Figure 10.3, a name declared in class `VBase` is always reached through class `RightPath`. The right path is more accessible because `RightPath` declares `VBase` as a public base class, whereas `LeftPath` declares `VBase` as private.

Special Member Functions



C++ defines several kinds of functions that can be declared only as class members—these are called "special member functions." These functions affect the way objects of a given class are created, destroyed, copied, and converted into objects of other types. Another important property of many of these functions is that they can be called implicitly (by the compiler).

The special member functions are described briefly in the following list:

- Constructors. These functions enable automatic initialization of objects. See "Constructors" on page 300.
- Destructors. These functions perform cleanup after objects are explicitly or implicitly destroyed. See "Destructors" on page 305.
- Conversion functions. These are used to convert between class types and other types. See "Conversion Functions" on page 315.
- The **new** operator. This is used to dynamically allocate storage. See "The operator new Function" on page 318.
- The **delete** operator. This is used to release storage allocated using the **new** operator. See "The operator delete Function" on page 323.
- The assignment operator (**operator=**). This operator is used when an assignment takes place. See "Compiler-Generated Copying" on page 334.

All of the items in the preceding list can be user-defined for each class.

Special member functions obey the same access rules as other member functions. The access rules are described in Chapter 10, "Member-Access Control." Table 11.1 is a summary of how member and friend functions behave.

Table 11.1 Summary of Function Behavior

Function Type	Is Function Inherited from Base Class?	Can Function Be Virtual?	Can Function Return a Value?	Is Function a Member or Friend?	Is Function Generated by the Compiler?
Constructor	No	No	No	Member	Yes
Destructor	No	Yes	No	Member	Yes
Conversion	Yes	Yes	No	Member	No
Assignment (**operator=**)	No	Yes	Yes	Member	Yes
Function call (**operator( )**)	Yes	Yes	Yes	Member	No
Subscript (**operator[ ]**)	Yes	Yes	Yes	Member	No
Member selection (**->**)	Yes	Yes	Yes	Member	No
Compound Assignment (**+=**, **-=**, and so on)	Yes	Yes	Yes	Either	No
new	Yes	No	**void***	Static member	No
delete	Yes	No	**void**	Static member	No
Operators not listed above	Yes	Yes	Yes	Either	No
Other member functions	Yes	Yes	Yes	Member	No
Friend functions	No	No	Yes	Friend	No

11.1 Constructors

A member function with the same name as its class is a constructor function. Constructors cannot return values, even if they have **return** statements. Specifying a constructor with a return type is an error, as is taking the address of a constructor.

If a class has a constructor, each object of that type is initialized with the constructor prior to use in a program. (For more information about initialization, see "Initialization Using Special Member Functions" on page 325.)

Constructors are called at the point an object is created. Objects are created as:

- Global (file-scoped or externally linked) objects.
- Local objects, within a function or smaller enclosing block.
- Dynamic objects, using the **new** operator. The **new** operator allocates an object on the program heap or "free store."
- Temporary objects created by explicitly calling a constructor. (For more information, see "Temporary Objects" on page 311.)
- Temporary objects created implicitly by the compiler. (For more information, see "Temporary Objects" on page 311.)
- Data members of another class. Creating objects of class type, where the class type is composed of other class-type variables, causes each object in the class to be created.
- Base class subobject of a class. Creating objects of derived class type causes the base class components to be created.

What a Constructor Does

A constructor performs various tasks that are not visible to you as the programmer, even if you write no code for the constructor. These tasks are all associated with building a complete and correct instance of class type.

In Microsoft C++ (and some other implementations of C++), a constructor:

- Initializes the object's virtual base pointer(s) (vbptr). This step is performed if the class has virtual base classes in the inheritance graph.
- Calls base class and member constructors in the order of declaration.
- Initializes the object's virtual function pointers (vfptr). This step is performed if the class has or inherits virtual functions. Virtual function pointers point to the class's virtual function table(s) (v-table) and allow correct binding of virtual function calls to code.
- Executes optional code in the body of the constructor function.

When the constructor is finished, the allocated memory is an object of a given class type. Because of the steps the constructor performs, "late binding" in the form of virtual functions can be resolved at the point of a virtual function call. The constructor has also constructed base classes and has constructed composed objects (objects included as data members). Late binding is the mechanism by which C++ implements polymorphic behavior for objects. (For a discussion of polymorphism, see Chapter 7 of the *C++ Tutorial* manual.)

Rules for Declaring Constructors

A constructor has the same name as its class. Any number of constructors can be declared, subject to the rules of overloaded functions. (For more information, see Chapter 12, "Overloading.")

Syntax

class-name (*argument-declaration-list*$_{opt}$) *cv-mod-list*$_{opt}$

C++ defines two special kinds of constructors, default and copy constructors, described in Table 11.2.

Table 11.2 Default and Copy Constructors

Kind of Construction	Arguments	Purpose
Default constructor	Can be called with no arguments	Construct a default object of the class type
Copy constructor	Can accept a single argument of reference to same class type	Copy objects of the class type

Default constructors can be called with no arguments. However, you can declare a default constructor with an argument list, provided all arguments have defaults. Similarly, copy constructors must be declared such that they can accept a single argument of reference to the same class type. More arguments can be supplied, provided all subsequent arguments have defaults.

If you do not supply any constructors, the compiler attempts to generate a default constructor. If you do not supply a copy constructor, the compiler attempts to generate one. These compiler-generated constructors are considered public member functions. Specifying a copy constructor with a first argument that is an object, and not a reference, generates an error.

A compiler-generated default constructor sets up the object (initializes vftables, and vbtables, as described previously), and it calls the default constructors for base classes and members, but it takes no other action. Base class and member constructors are called only if they exist, if they are accessible, and if they are unambiguous.

A compiler-generated copy constructor sets up a new object and performs a memberwise copy of the contents of the object to be copied. If base class or member constructors exist, they are called; otherwise, bitwise copying is performed.

If all base and member classes of a class *type* have copy constructors that accept a **const** argument, the compiler-generated copy constructor accepts a single argument of type **const** *type***&**. Otherwise, the compiler-generated copy constructor accepts a single argument of type *type***&**.

You can use a constructor to initialize a **const** or **volatile** object, but the constructor itself cannot be declared as either **const** or **volatile**. In addition, constructors cannot be declared as **virtual** or **static**.

Constructors of base classes are not inherited by derived classes. When an object of derived class type is created, it is constructed starting with the base class components; then it moves to the derived class components. The compiler uses each base class's constructor as that part of the complete object is initialized (except in cases of virtual derivation, as described in "Initializing Base Classes" on page 332).

Explicitly Called Constructors

Constructors can be explicitly called in a program to create objects of a given type. For example, to create two `Point` objects that describe the ends of a line, the following code can be written:

```
DrawLine( Point( 13, 22 ), Point( 87, 91 ) );
```

Two objects of type `Point` are created, passed to the function `DrawLine`, and destroyed at the end of the expression (the function call).

Another context in which a constructor is explicitly called is in an initialization:

```
Point pt = Point( 7, 11 );
```

An object of type `Point` is created and initialized using the constructor that accepts two arguments of type **int**.

Objects that are created by calling constructors explicitly, as in the preceding two examples, are unnamed and have a lifetime of the expression in which they are created. This is discussed in greater detail in "Temporary Objects" on page 311.

Calling Member Functions and Virtual Functions from Within Constructors

Because the object has been completely set up (virtual tables have been initialized and so on) prior to the execution of the first line of user code, it is usually safe to call any member function from within a constructor. A potentially unsafe member function call has to do with calling a virtual member function for an abstract base class during construction or destruction.

Constructors can call virtual functions. When virtual functions are called, the function invoked is the function defined for the constructor's own class (or inherited from its bases). The following example shows what happens when a virtual function is called from within a constructor:

```
#include <iostream.h>

class Base
{
    Base();             // Default constructor.
    virtual void f();   // Virtual member function.
};

Base::Base()
{
    cout << "Constructing Base sub-object\n";
    f();                // Call virtual member function
}                       //  from inside constructor.

void Base::f()
{
    cout << "Called Base::f()\n";
}

class Derived : public Base
{
    Derived();          // Default constructor.
    void f();           // Implementation of virtual
};                      //  function f for this class.

Derived::Derived()
{
    cout << "Constructing Derived object\n";
}

void Derived::f()
{
    cout << "Called Derived::f()\n";
}

void main()
{
    Derived d;
}
```

When the preceding program is run, the declaration `Derived d` causes the following sequence of events to occur:

1. The constructor for class `Derived` (`Derived::Derived`) is called.
2. Prior to entering the body of the `Derived` class's constructor, the constructor for class `Base` (`Base::Base`) is called.
3. `Base::Base` calls the function `f`, which is a virtual function. Ordinarily, `Derived::f` would be called because the object `d` is of type `Derived`. Because the `Base::Base` function is a constructor, the object is not yet of the `Derived` type, and `Base::f` is called.

Constructors and Arrays

Arrays are constructed only using the default constructor. Constructors that either accept no arguments or constructors for which all arguments have a default are considered "default constructors." Arrays are always constructed in ascending order. This means that the initialization for each member of the array is done using the same constructor.

Order of Construction

For derived classes and classes that have class-type member data, the order in which construction occurs helps you understand what portions of the object you can use in any given constructor.

Construction and Inheritance

An object of derived type is constructed from the base class to the derived class by calling the constructors for each class in order. Each class's constructor can rely on its base class being completely constructed.

For a complete description of initialization, including the order of initialization, see "Initializing Bases and Members" on page 329.

Construction and Composed Classes

Classes that contain class-type data members are called "composed classes." When an object of a composed class type is created, the constructors for the contained classes are called before the class's own constructor.

For a more information about this kind of initialization, see "Initializing Bases and Members" on page 329.

11.2 Destructors

"Destructor" functions are the inverse of constructor functions. They are called when objects are destroyed (deallocated). Designate a function as a class's destructor by preceding the class name with a tilde (~). For example, the destructor for class `String` is declared: `~String()`.

The destructor is commonly used to "clean up" when an object is no longer necessary. Consider the following declaration of a `String` class:

```
#include <string.h>

class String
{
public:
    String( char *ch );  // Declare constructor
    ~String();           //  and destructor.
private:
    char *_text;
};

// Define the constructor.
String::String( char *ch )
{
    // Dynamically allocate the correct amount of memory.
    _text = new char[strlen( ch ) + 1];

    // If the allocation succeeds, copy the initialization string.
    if( _text )
       strcpy( _text, ch );
}

// Define the destructor.
String::~String()
{
    // Deallocate the memory that was previously reserved
    //  for this string.
    delete[] _text;
}
```

In the preceding example, the destructor `String::~String` uses the **delete** operator to deallocate the space dynamically allocated for text storage.

Declaring Destructors

Destructors are functions with the same name as the class but preceded by a tilde (~).

Syntax

~class-name()

or

class-name :: *~class-name*()

The first form of the syntax is used for destructors declared or defined inside a class declaration; the second form is used for destructors defined outside a class declaration.

Several rules govern the declaration of destructors. Destructors:

- Do not accept arguments.
- Cannot specify any return type (including **void**).
- Cannot return a value using the **return** statement.
- Cannot be declared as **const** or **volatile**, nor can they be declared as **static**. However, they can be invoked for the destruction of objects declared as **const**, **volatile**, or **static**.
- Can be declared as **virtual**. Using virtual destructors, you can destroy objects without knowing their type—the correct destructor for the object is invoked using the virtual function mechanism. Note that destructors can also be declared as pure virtual functions for abstract classes.

Using Destructors

Destructors are called when one of the following events occurs:

- An object allocated using the **new** operator is explicitly deallocated using the **delete** operator. When objects are deallocated using the **delete** operator, memory is freed for the "most derived object," or the object that is a complete object and not a subobject representing a base class. This "most-derived object" deallocation is guaranteed to work only with virtual destructors. Deallocation may fail in multiple inheritance situations where the type information does not correspond to the underlying type of the actual object.
- A local (automatic) object with block scope goes out of scope.
- The lifetime of a temporary object ends.
- A program ends and global or static objects exist.
- The destructor is explicitly called using the destructor function's fully qualified name. (For more information, see "Explicit Destructor Calls" on page 310.)

The cases described in the preceding list ensure that all objects can be destroyed with user-defined methods.

If a base class or data member has an accessible destructor, and a derived class does not declare a destructor, the compiler generates one. This compiler-generated destructor calls the base class destructor and the destructors for members of the derived type. Default destructors are public. (For more information about accessibility, see "Access Specifiers for Base Classes" in Chapter 10, on page 287.)

Destructors can freely call class member functions and access class member data. When a virtual function is called from a destructor, the function called is the function for the class currently being destroyed. (For more information, see "Order of Destruction" on page 308.)

There are two restrictions on the use of destructors. The first restriction is that you cannot take the address of a destructor. The second is that derived classes do not inherit their base class's destructors.

Order of Destruction

When an object goes out of scope or is deleted, the sequence of events in its complete destruction is as follows:

1. The class's destructor is called, and the body of the destructor function is executed.
2. Destructors for nonstatic member objects are called in the reverse order in which they appear in the class declaration. The optional member initialization list used in construction of these members does not affect the order of construction or destruction. (For more information about initializing members, see "Initializing Bases and Members" on page 329.)
3. Destructors for nonvirtual base classes are called in the reverse order of declaration.
4. Destructors for virtual base classes are called in the reverse order of declaration.

Nonvirtual Base Classes

When calling destructors for nonvirtual base classes, the destructors are called in the reverse order in which the base class names are declared. Consider the following class declaration:

```
class MultInherit : public Base1, public Base2
...
```

In the preceding example, the destructor for `Base2` is called before the destructor for `Base1`.

Virtual Base Classes

Destructors for virtual base classes are called in the reverse order of their appearance in a directed acyclic graph (depth-first, left-to-right, postorder traversal). Figure 11.1 depicts an inheritance graph.

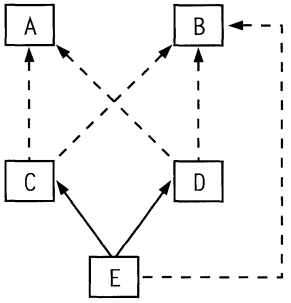


Figure 11.1 Inheritance Graph Showing Virtual Base Classes

The following lists the class heads for the classes shown in Figure 11.1.

```
class A
class B
class C : virtual public A, virtual public B
class D : virtual public A, virtual public B
class E : public C, public D, virtual public B
```

To determine the order of destruction of the virtual base classes of an object of type `E`, the compiler builds a list by applying the following algorithm:

1. Traverse the graph left, starting at the deepest point in the graph (in this case, `E`).
2. Perform leftward traversals until all nodes have been visited. Note the name of the current node.
3. Revisit the previous node (down and to the right) to find out if the node being remembered is a virtual base class.
4. If the remembered node is a virtual base class, scan the list to see if it has already been entered. If it is not a virtual base class, ignore it.
5. If the remembered node is not yet in the list, add it to the bottom of the list.
6. Traverse the graph up and along the next path to the right.
7. Go to step 2.
8. When the last upward path is exhausted, note the name of the current node.
9. Go to step 3.
10. Continue this process until the bottom node is again the current node.

So, for class `E`, the order of destruction is:

1. The nonvirtual base class `E`.
2. The nonvirtual base class `D`.
3. The nonvirtual base class `C`.
4. The virtual base class `B`.
5. The virtual base class `A`.

This process produces an ordered list of unique entries. No class name appears twice. Once the list is constructed, it is walked in reverse order, and the destructor for each of the classes in the list from the last to the first are called.

The order of construction or destruction is primarily important when constructors or destructors in one class rely on the other component being created first or persisting longer—for example, if the destructor for `A` (in the graph in Figure 11.1) relied on `B` still being present when its code executed, or vice versa.

Such interdependencies between classes in an inheritance graph are inherently dangerous because classes derived later can alter which is the leftmost path, thereby changing the order of construction and destruction.

Explicit Destructor Calls

Calling a destructor explicitly is seldom necessary. However, it can be useful to perform cleanup of objects placed at absolute addresses. These objects are commonly allocated using a user-defined **new** operator that takes a placement argument. The **delete** operator cannot deallocate this memory because it is not allocated from the free store (for more information, see "The new and delete Operators" on page 318). A call to the destructor, however, can perform appropriate cleanup. To explicitly call the destructor for an object, `s`, of class `String`, use one of the following statements:

```
s.String::~String();
```

or

```
ps->String::~String();
```

The notation for explicit calls to destructors, shown above, can be used regardless of whether the type defines a destructor. This allows the programmer to make such explicit calls without knowing if a destructor is defined for the type. An explicit call to a destructor where none is defined has no effect.

Note Explicit calls to destructors call only the destructor for the class specified. When destructors are declared as virtual, the virtual function calling mechanism is disabled for these explicit calls.

11.3 Temporary Objects

In some cases, it is necessary for the compiler to create temporary objects. These temporary objects can be created for the following reasons:

- To initialize a **const** reference with an initializer of a type different from that of the underlying type of the reference being initialized.
- To store the return value of a function that returns a user-defined type. These temporaries are created only if your program does not copy the return value to an object. For example:

```
UDT Func1();     //  Declare a function that returns a user-defined
                 //   type.

...

Func1();         //  Call Func1, but discard return value.
                 //  A temporary object is created to store the return
                 //   value.
```

  Because the return value is not copied to another object, a temporary object is created. A more common case where temporaries are created is during the evaluation of an expression where overloaded operator functions must be called. These overloaded operator functions return a user-defined type which often is not copied to another object.

  Consider the expression `ComplexResult = Complex1 + Complex2 + Complex3`. The expression `Complex1 + Complex2` is evaluated, and the result is stored in a temporary object. Next, the expression *temporary* `+ Complex3` is evaluated, and the result is copied to `ComplexResult` (assuming the assignment operator is not overloaded).
- To store the result of a cast to a user-defined type. When an object of a given type is explicitly converted to a user-defined type, that new object is constructed as a temporary object.

Temporary objects have a lifetime that is defined by their point of creation and the point at which they are destroyed. Any expression that creates more than one temporary object eventually destroys them in the reverse order in which they were created. The points at which destruction occurs are shown in Table 11.3.

Table 11.3 Destruction Points for Temporary Objects

Reason Temporary Created	Destruction Point
Result of expression evaluation	All temporaries created as a result of expression evaluation are destroyed at the end of the expression statement (that is, at the semicolon), or at the end of the controlling expressions for **for**, **if**, **while**, **do**, and **switch** statements.
Result of expressions using the built-in (not overloaded) logical operators (|| and &&)	Immediately after the right operand. At this destruction point, all temporary objects created by evaluation of the right operand are destroyed.
Initializing **const** references	If an initializer is not an l-value of same type as reference being initialized, a temporary of the underlying object type is created and initialized with the initialization expression. This temporary object is destroyed immediately after the reference object to which it is bound is destroyed.

11.4 Conversions

Objects of a given class type can be converted to objects of another type. This is done by constructing an object of the target class type from the source class type and copying the result to the target object. This process is called conversion by constructor. Objects can also be converted by user-supplied conversion functions.

When standard conversions (described in Chapter 3) cannot completely convert from a given type to a class type, the compiler can select user-defined conversions to help complete the job. In addition to explicit type conversions, conversions take place when:

- An initializer expression is not the same type as the object being initialized.
- The type of argument used in a function call does not match the type of argument specified in the function declaration.
- The type of the object being returned from a function does not match the return type specified in the function declaration.
- Two expression operands must be of the same type.
- An expression controlling an iteration or selection statement requires a different type from the one supplied.

A user-defined conversion is applied only if it is unambiguous; otherwise, an error message is generated. Ambiguity is checked at the point of usage. Hence, if the features that cause ambiguity are not used, a class can be designated with potential ambiguities and not generate any errors. Although there are many situations in which ambiguities arise, these are two leading causes of ambiguities:

- A class type is derived using multiple inheritance, and it is unclear from which base class to select the conversion (see "Ambiguity" in Chapter 9, on page 282).
- An explicit type-conversion operator and a constructor for the same conversion exist (see "Conversion Functions" on page 315).

Both conversion by constructor and conversion by conversion functions obey access control rules, as described in Chapter 10. Access control is tested only after the conversion is found to be unambiguous.

Conversion Constructors

A constructor that can be called with a single argument is used for conversions from the type of the argument to the class type. Such a constructor is called a conversion constructor. Consider the following example:

```
class Point
{
public:
    Point();
    Point( int );
    ...
};
```

The preceding example declares two constructors: a default constructor that takes no arguments, and a constructor that converts from type **int**. In code that follows, any conversion from type **int** can use the `Point( int )` constructor to convert from type **int** to type `Point`.

Sometimes a conversion is required but no conversion constructor exists in the class. These conversions cannot be performed by constructors. The compiler does not look for intermediate types through which to perform the conversion. For example, suppose a conversion exists from type `Point` to type `Rect`, and a conversion exists from type **int** to type `Point`. The compiler does not supply a conversion from type **int** to type `Rect` by constructing an intermediate object of type `Point`.

Conversions and Constants

While constants for built-in types such as **int**, **long**, and **double** can appear in expressions, no constants of class types are allowed (this is partly because classes usually describe an object complicated enough to make notation inconvenient). However, if conversion constructors from built-in types are supplied, constants of these built-in types can be used in expressions, and the conversions cause correct behavior. For example, a `Money` class can have conversions from types **long** and **double**:

```
class Money
{
public:
    Money( long );
    Money( double );
   ...
    Money operator+( const Money& );  // Overloaded addition operator.
};
```

Therefore, expressions such as the following can specify constant values:

```
Money AccountBalance = 37.89;
Money NewBalance = AccountBalance + 14L;
```

The second example involves the use of an overloaded addition operator, which is covered in the next chapter. Both examples cause the compiler to convert the constants to type `Money` before using them in the expressions.

Drawbacks of Conversion Constructors

Because the compiler can select a conversion constructor implicitly, you relinquish control over what functions are called when. If it is essential to retain full control, do not declare any constructors that take a single argument; instead, define "helper" functions to perform conversions, as in the following example:

```
#include <stdio.h>
#include <stdlib.h>

// Declare Money class.
class Money
{
public:
    Money();
```

```
// Define conversion functions that can only be called explicitly.
    static Money Convert( char * ch ) { return Money( ch ); }
    static Money Convert( double d )    { return Money( d ); };
    void  Print() { printf( "\n%f", _amount ); }
private:
    Money( char *ch ) { _amount = atof( ch ); }
    Money( double d ) { _amount = d; }
    double _amount;
};

main()
{
    // Perform a conversion from type char * to type Money.
    Money Acct = Money::Convert( "57.29" );
    Acct.Print();
    // Perform a conversion from type double to type Money.
    Acct = Money::Convert( 33.29 );
    Acct.Print();

    return 0;
}
```

In the preceding code, the conversion constructors are private and cannot be used in type conversions. However, they can be invoked explicitly by calling the `Convert` functions. Because the `Convert` functions are static, they are accessible without referencing a particular object.

Conversion Functions

In conversion by constructors, described in the previous section, objects of one type can be implicitly converted to a particular class type. This section describes a means by which you can provide explicit conversions from a given class type to another type. Conversion from a class type is often accomplished using conversion functions. Conversion functions use the following syntax:

Syntax

conversion-function-name:
operator *conversion-type-name* ()

conversion-type-name:
type-specifier-list ptr-operator$_{opt}$

The following example specifies a conversion function that converts type `Money` to type `double`:

```
class Money
{
public:
    Money();
    operator double() { return _amount; }
private:
    double _amount;
};
```

Given the preceding class declaration, the following code can be written:

```
Money Account;
...
double CashOnHand = Account;
```

The initialization of `CashOnHand` with `Account` causes a conversion from type `Account` to type `double`.

Conversion functions are often called "cast operators" because they (along with constructors) are the functions called when a cast is used. The following example uses a cast, or explicit conversion, to print the current value of an object of type `Money`:

```
cout << (double)Account << endl;
```

Conversion functions are inherited in derived classes. Conversion operators hide only base-class conversion operators that convert to exactly the same type. Therefore, a user-defined **operator int** function does not hide a user-defined **operator short** function in a base class.

Only one user-defined conversion function is applied when performing implicit conversions. If there is no explicitly defined conversion function, the compiler does not look for intermediate types into which an object can be converted.

If a conversion is required that causes an ambiguity, an error is generated. Ambiguities arise when more than one user-defined conversion is available or when a user-defined conversion and a built-in conversion exist.

The following example illustrates a class declaration with a potential ambiguity:

```
#include <string.h>

class String
{
public:
    // Define constructor that converts from type char *.
    String( char *s ) { strcpy( _text, s ); }
    // Define conversion to type char *.
    operator char *() { return _text; }
    int operator==( const String &s )
    { return !strcmp( _text, s._text ); }
private:
    char _text[80];
};

main()
{
    String s( "abcd" );
    char  *ch = "efgh";

    // Cause the compiler to select a conversion.
    return s == ch;
}
```

In the expression `s == ch`, the compiler has two choices, and no way of determining which is correct. It can convert `ch` to an object of type `String` using the constructor, then perform the comparison using the user-defined `operator==`. Or, it can convert `s` to a pointer of type `char *` using the conversion function, then perform a comparison of the pointers.

Because neither choice is "more correct" than the other, the compiler cannot determine the meaning of the comparison expression, and it generates an error.

Rules for Declaring Conversion Functions

The following four rules are used when declaring conversion functions (see page 315 for syntax):

- Classes, enumerations, and **typedef** names cannot be specified in the *type-specifier-list*. Therefore, the following code generates an error:

  ```
  operator struct String{ char string_storage; }();
  ```

  Instead, declare the `String` structure prior to the conversion function.
- Conversion functions take no arguments. Specifying arguments generates an error.

- Conversion functions have the return type specified by the *conversion-type-name*; specifying any return type for a conversion function generates an error.
- Conversion functions can be declared as **virtual**.

11.5 The new and delete Operators

C++ supports dynamic allocation and deallocation of objects using the **new** and **delete** operators. These operators allocate memory for objects from a pool called the "free store." The new operator calls the special function **operator new**, and the **delete** operator calls the special function **operator delete**.

The operator new Function

When a statement such as the following is encountered in a program, it translates into a call to the function **operator new**:

```
char *pch = new char[BUFFER_SIZE];
```

If there is insufficient memory for the allocation request, **operator new** returns **NULL**. However, if the request is for zero bytes of storage, **operator new** returns a pointer to a distinct object (that is, repeated calls to **operator new** return different pointers).

There are two scopes for **operator new** functions. They are described in Table 11.4.

Table 11.4 Scope for operator new Functions

Operator	Scope
::operator new	Global
class-name::**operator new**	Class

The first argument to **operator new** must be of type **size_t** (a type defined in STDDEF.H), and the return type is always **void ***.

The global **operator new** function is called when the **new** operator is used to allocate objects of built-in types, objects of class type that do not contain user-defined **operator new** functions, and arrays of any type. When the **new** operator is used to allocate objects of a class type where an **operator new** is defined, that class's **operator new** is called.

An **operator new** function defined for a class is a static member function (which cannot, therefore, be virtual) that hides the global **operator new** function for objects of that class type. Consider the case where **new** is used to allocate and set memory to a given value:

```
#include <malloc.h>
#include <memory.h>

class Blanks
{
public:
    Blanks(){}
    void *operator new( size_t stAllocateBlock, char chInit );
};

void *Blanks::operator new( size_t stAllocateBlock, char chInit )
{
    void *pvTemp = malloc( stAllocateBlock );
    if( pvTemp != 0 )
        memset( pvTemp, chInit, stAllocateBlock );
    return pvTemp;
}
```

For discrete objects of type `Blanks`, the global **operator new** function is hidden. Therefore, the following code allocates an object of type `Blanks` and initializes it to `0xa5`:

```
main()
{
    Blanks *a5 = new( 0xa5 ) Blanks;

    return a5 != 0;
}
```

The argument supplied in parentheses to **new** is passed to `Blanks::operator new` as the `chInit` argument. However, the global **operator new** function is hidden, causing code such as the following to generate an error:

```
Blanks *SomeBlanks = new Blanks;
```

Nonclass types and all arrays (regardless of whether they are of class type) allocated using the **new** operator always use the global **operator new** function.

Microsoft Specific

The **operator new** function can be overloaded on its return type to allow specification of different **new** operators for different memory models. Table 11.5 shows the various function declarations for **operator new**. (For more information about overloading, see Chapter 12, "Overloading.")♦

Table 11.5 Declarations for new Operator

Heap	Declaration
Default	`void *operator new( size_t );` The argument is the amount of storage (in bytes) to allocate.
Near	`void __near *operator new( size_t );` The argument is the amount of storage (in bytes) to allocate.
Far	`void __far *operator new( size_t );` The argument is the amount of storage (in bytes) to allocate.
Huge	`void __huge *operator new( unsigned long, size_t );` The first argument is the number of elements to allocate, and the second argument is the size of a given element (in bytes).

Based	`void __based( void ) *operator new( __segment, size_t );`
	The first argument is the segment specified in the **new** expression, and the second argument is the amount of storage (in bytes) to allocate. For example, the expression:
	`new __based( __segname( "_TEXT" ) ) int`
	calls **operator new** with the segment value corresponding to `_TEXT` and a size equal to **sizeof(int)**.

Note The **operator new** function is the only function that can be overloaded solely on the basis of return type. It can be overloaded only in the forms shown in Table 11.5.

Handling Insufficient Memory Conditions

Testing for failed memory allocation can be done with code such as the following:

```
int *pi = new int[BIG_NUMBER];

if( pi == 0 )
{
    cerr << "Insufficient memory" << endl;
    return -1;
}
```

In some circumstances, corrective action can be taken during memory allocation and the request can be fulfilled. To gain control when the global **operator new** function fails, use the **_set_new_handler** function (defined in NEW.H) as follows:

```
#include <stdio.h>
#include <new.h>

// Define a function to be called if new fails to allocate memory.
int MyNewHandler( size_t size )
{
    clog << "Allocation failed. Coalescing heap. << endl;

    // Call a fictitious function to recover some heap space.
    return CoalesceHeap();
}

main()
{
    // Set the failure handler for new to be MyNewHandler.
    _set_new_handler( MyNewHandler );

    int *pi = new int[BIG_NUMBER];

    return 0;
}
```

In the preceding example, the first statement in **main** sets the new handler to `MyNewHandler`. The second statement tries to allocate a large block of memory using the **new** operator. When the allocation fails, control is transferred to `MyNewHandler`. The argument passed to `MyNewHandler` is the number of bytes requested. The value returned from `MyNewHandler` is a flag indicating whether allocation should be retried: a nonzero value indicates that allocation should be retried, and a zero value indicates that allocation has failed.

`MyNewHandler` prints a warning message and takes corrective action. If `MyNewHandler` returns a nonzero value, the **new** operator retries the allocation. Only when `MyNewHandler` returns a 0 does the **new** operator stop trying and return a zero value to the program.

The **_set_new_handler** function returns the address of the previous new handler. Therefore, if a new handler needs to be installed for a short time, the previous new handler can be reinstalled using code such as the following:

```
#include <new.h>

...

_PNH old_handler = _set_new_handler( MyNewHandler );

// Code that requires MyNewHandler.
...

// Reinstall previous new handler.
_set_new_handler( old_handler );
```

A call to **_set_new_handler** with an argument of 0 causes the default new handler to be reinstalled.

The new handler you specify can have any name, but it must be a function returning type **int** (nonzero indicates the new handler succeeded, and zero indicates that it failed). The argument list is described in Tables 11.6 and 11.7.

If a user-defined **operator new** is provided, the new handler functions are not automatically called on failure.

Microsoft Specific

In Microsoft C++, objects can be allocated on the default heap, the near heap, the far heap, the huge heap, or a based heap. New handlers for these heaps can be set using the functions declared in NEW.H, which are listed in Table 11.6.

Table 11.6 Functions Used to Set New Handlers

Function Name	New Handler That Is Set	Prototype
_set_new_handler	Default new handler	**_PNH _set_new_handler (_PNH);**
set_nnew_handler	New handler for objects allocated on the near heap	**_PNH _set_nnew_handler (_PNH);**

Table 11.6 *(continued)*

Function Name	New Handler That Is Set	Prototype
_set_fnew_handler	New handler for objects allocated on the far heap	**_PNH _set_fnew_handler (_PNH);**
_set_hnew_handler	New handler for objects allocated on the huge heap	**_PNHH _set_hnew_handler (_PNHH);**
_set_bnew_handler	New handler for objects allocated on the based heap	**_PNHB _set_bnew_handler (_PNHB);**

To facilitate easy declaration of the new handlers, three types are defined. See Table 11.7.

Table 11.7 List of New-Handler Types

Type	Meaning
_PNH	Pointer to a function that returns type **int** and takes a single argument of type **size_t**.
_PNHH	Pointer to a function that returns type **int** and takes arguments of type **unsigned long** (the number of elements) and type **size_t** (the size of a given element).
_PNHB	Pointer to a function that returns type **int** and takes arguments of type **__segment** (the segment base) and of type **size_t**.

By using the appropriate "set new handler" function, you can trap memory-allocation failures for default, near, far, huge, and based objects.♦

The operator delete Function

Memory that is dynamically allocated using the **new** operator can be freed using the **delete** operator. The delete operator causes the **operator delete** function to be called, which frees memory back to the available pool. Using the **delete** operator also causes the class destructor (if there is one) to be called.

There are global and class-scoped **operator delete** functions. Only one **operator delete** function can be defined for a given class; if defined, it hides the global **operator delete** function. The global **operator delete** function is always called for arrays of any type.

The global **operator delete** function, if declared, takes a single argument of type **void ***, which contains a pointer to the object to deallocate. The return type is **void** (**operator delete** cannot return a value). Two forms exist for class-member **operator delete** functions:

```
void operator delete( void * );

void operator delete( void *, size_t );
```

Only one of the preceding two variants can be present for a given class. The first form works as described for global **operator delete**. The second form takes two arguments, the first of which is a pointer to the memory block to deallocate, and the second of which is the number of bytes to deallocate. The second form is particularly useful when an **operator delete** function from a base class is used to delete an object of a derived class.

The **operator delete** function is static; therefore, it cannot be virtual. The **operator delete** function obeys access control, as described in Chapter 10.

The following example shows user-defined **operator new** and **operator delete** functions designed to log allocations and deallocations of memory:

```
#include <iostream.h>
#include <stdlib.h>

int _fLogMemory = 0;        // Perform logging (0=no; nonzero=yes)?
int cBlocksAllocated = 0;   // Count of blocks allocated.

// User-defined operator new.
void *operator new( size_t stAllocateBlock )
{
    static fInOpNew = 0     // Guard flag.

    if( _fLogMemory && !fInOpNew )
    {
        fInOpNew = 1;
        clog << "Memory block "   << ++cBlocksAllocated
             << " allocated for " << stAllocateBlock
             << " bytes\n";
        fInOpNew = 0;
    }

    return malloc( stAllocateBlock );
}

// User-defined operator delete.
void operator delete( void *pvMem )
{
    static fInOpDelete = 0     // Guard flag.
```

```
    if( _fLogMemory && !fInOpDelete )
    {
        fInOpDelete = 1;
        clog << "Memory block " << --cBlocksAllocated
             << " deallocated\n";
        fInOpDelete = 0;
    }

    free( pvMem );
}

main( int argc, char *argv[] )
{
    _fLogMemory = 1;  // Turn logging on.
    if( argc > 1 )
    for( int i = 0; i < atoi( argv[1] ); ++i )
    {
        char *pMem = new char[10];
        delete pMem;
    }

    return cBlocksAllocated;
}
```

The preceding code can be used to detect "memory leakage"—that is, memory that is allocated on the free store, but never freed. To perform this detection, the global **new** and **delete** operators are redefined to count allocation and deallocation of memory.

Using **delete** on a null pointer has no effect. (For more information about the **delete** operator, see "delete Operator" in Chapter 4, on page 101.)

11.6 Initialization Using Special Member Functions

This section describes initialization using special member functions. It expands on the following discussions of initialization:

- "Initializing Aggregates" in Chapter 7, on page 219, which describes how to initialize arrays of nonclass types and objects of simple class types. These simple class types cannot have private or protected members, and they cannot have base classes.
- "Constructors" on page 300, which explains how to initialize class-type objects using special constructor functions.

The default method of initialization is to perform a bit-for-bit copy from the initializer into the object to be initialized. This technique is applicable only to:

- Objects of built-in types. For example:

```
int i = 100;
```

- Pointers. For example:

```
int i;
int *pi = &i;
```

- References. For example:

```
String sFileName( "FILE.DAT" );
String &rs = &sFileName;
```

- Objects of class type, where the class has no private or protected members, no virtual functions, and no base classes. For example:

```
struct Point
{
    int x, y;
};

Point pt = { 10, 20 };
```

Classes can specify more refined initialization by defining constructor functions. (For more information about declaring such functions, see "Constructors" on page 300.) If an object is of a class type that has a constructor, the object must be initialized, or there must be a default constructor. Objects that are not specifically initialized invoke the class's default constructor.

Explicit Initialization

Two forms of explicit initialization are supported in C++:

- Supplying an initializer list in parentheses:

```
String sFileName( "FILE.DAT" );
```

  The items in the parenthesized list are considered arguments to the class constructor. This form of initialization enables initialization of an object with more than one value and can also be used in conjunction with the **new** operator. For example:

```
Rect *pRect = new Rect( 10, 15, 24, 97 );
```

- Supplying a single initializer using the equal-sign initialization syntax. For example:

  ```
  String sFileName = "FILE.DAT";
  ```

  While the preceding example works the same way as the example shown for `String` in the first list item, the syntax is not adaptable to use with objects allocated on the free store.

  The single expression on the right of the equal sign is taken as the argument to the class's copy constructor; therefore, it must be a type that can be converted to the class type.

  Note that because the equal sign (=) in the context of initialization is different from an assignment operator, overloading **operator=** has no effect on initialization.

The equal-sign initialization syntax is different from the function-style syntax, even though the generated code is identical in most cases. The difference is that when the equal-sign syntax is used, the compiler has to behave as if the following sequence of events were taking place:

- Create a temporary object of the same type as the object being initialized.
- Copy the temporary object to the object.

The compiler must perform accessibility checking on the copy constructor before performing these steps. Even though the compiler can eliminate the temporary creation and copy steps in most cases, an inaccessible copy constructor causes equal-sign initialization to fail. Consider the following example:

```
class anInt
{
    anInt( const anInt& );     //  Private copy constructor.
public:
    anInt( int );              //  Public int constructor.
};
...
anInt myInt = 7;               //  Access-control violation. Attempt to
                               //   reference private copy constructor.
anInt myInt( 7 );              //  Correct; no copy constructor called.
```

When a function is called, class-type arguments passed by value and objects returned by value are conceptually initialized using the form:

type-name name = *value*

For example:

```
String s = "C++";
```

Therefore, it follows that the argument type must be a type that can be converted to the class type being passed as an argument. The class's copy constructor, as well as user-defined conversion operators or constructors that accept the type of the actual argument, must be public.

In expressions that use the **new** operator, the objects allocated on the free store are conceptually initialized using the form:

type-name name(*initializer* $_1$, *initializer* $_2$, ... *initializer* $_n$)

For example:

```
String *ps = new String( "C++" );
```

Initializers for base-class components, and member objects of a class are also conceptually initialized this way. (For more information, see "Initializing Bases and Members" on page 329.)

Initializing Arrays

If a class has a constructor, arrays of that class are initialized by a constructor. If there are fewer items in the initializer list than elements in the array, the default constructor is used for the remaining elements. If no default constructor is defined for the class, the initializer list must be complete—that is, there must be one initializer for each element in the array.

Consider the `Point` class that defines two constructors:

```
class Point
{
public:
    Point();           // Default constructor.
    Point( int, int ); // Construct from two ints.
    ...
};
```

An array of `Point` objects can be declared as follows:

```
Point aPoint[3] = {
    Point( 3, 3 )     // Use int, int constructor.
};
```

The first element of `aPoint` is constructed using the constructor `Point( int, int )`; the remaining two elements are constructed using the default constructor.

Static member arrays (whether **const** or not) can be initialized in their definitions (outside the class declaration). For example:

```
class WindowColors
{
public:
    static const char *rgszWindowPartList[7];
    ...
};
const char *WindowColors::rgszWindowPartList[7] = {
    "Active Title Bar", "Inactive Title Bar", "Title Bar Text",
    "Menu Bar", "Menu Bar Text", "Window Background", "Frame"   };
```

Initializing Static Objects

Global static objects are initialized in the order they occur in the source. They are destroyed in the reverse order. However, across translation units, the order of initialization is dependent on how the object files are arranged by the linker; destruction still takes place in the reverse order that objects were constructed.

Local static objects are initialized when they are first encountered in the program flow, and they are destroyed in the reverse order at program termination. Destruction of local static objects occurs only if the object was encountered in the program flow and initialized.

Initializing Bases and Members

An object of a derived class is made up of a component that represents each base class and a component that is unique to the particular class. Objects of classes that have member objects also contain components of other class types. This section describes how these component objects are initialized when an object of the class type is created.

To perform the initialization, the constructor-initializer, or *ctor-initializer*, syntax is used.

Syntax

ctor-initializer:
mem-initializer-list

mem-initializer-list:
mem-initializer
mem-initializer , *mem-initializer-list*

mem-initializer:
complete-class-name (*expression-list*$_{opt}$)
identifier (*expression-list*$_{opt}$)

This syntax, used in constructors, is described more fully in the next section, "Initializing Member Objects" and "Initializing Base Classes" on page 332.

Initializing Member Objects

Classes can contain member objects of class type, but to ensure that initialization requirements for the member objects are met, one of the following conditions must be met:

- The contained object's class requires no constructor.
- The contained object's class has an accessible default constructor.
- The containing class's constructors all explicitly initialize the contained object.

The following example shows how to perform such an initialization:

```
// Declare a class Point.
class Point
{
public:
    Point( int x, int y ) { _x = x; _y = y; }
private:
    int _x, _y;
};
```

```
// Declare a rectangle class that contains objects of type Point.
class Rect
{
public:
    Rect( int x1, int y1, int x2, int y2 );
private:
    Point _topleft, _bottomright;
};

//  Define the constructor for class Rect. This constructor
//   explicitly initializes the objects of type Point.
Rect::Rect( int x1, int y1, int x2, int y2 ) :
_topleft( x1, y1 ), _bottomright( x2, y2 )
{
}
```

The `Rect` class, shown in the preceding example, contains two member objects of class `Point`. Its constructor explicitly initializes the objects `_topleft` and `_bottomright`. Note that a colon follows the closing parenthesis of the constructor (in the definition). The colon is followed by the member names and arguments with which to initialize the objects of type `Point`.

Warning The order in which the member initializers are specified in the constructor does not affect the order in which the members are constructed; the members are constructed in the order they are declared in the class.

Reference and **const** member objects must be initialized using the member initialization syntax shown in "Syntax" in "Initializing Bases and Members" on page 329. There is no other way to initialize these objects.

Initializing Base Classes

Direct base classes are initialized in much the same way as member objects. Consider the following example:

```
//  Declare class Window.
class Window
{
public:
    Window( Rect rSize );
    ...
};

//  Declare class DialogBox, derived from class Window.
class DialogBox : public Window
{
public:
    DialogBox( Rect rSize );
    ...
};

//  Define the constructor for DialogBox. This constructor
//   explicitly initializes the Window subobject.
DialogBox::DialogBox( Rect rSize ) : Window( rSize )
{
}
```

Note that in the constructor for `DialogBox`, the `Window` base class is initialized using the argument `rSize`. This initialization consists of the name of the base class to initialize, followed by a parenthesized list of arguments to the class's constructor.

In initialization of base classes, the object that is not the subobject representing a base class's component is considered a "complete object." The complete object's class is considered the "most derived" class for the object.

The subobjects representing virtual base classes are initialized by the constructor for the most derived class. That means that where virtual derivation is specified, the most derived class must explicitly initialize the virtual base class, or the virtual base class must have a default constructor. Initializations for virtual base classes that appear in constructors for classes other than the most derived class are ignored.

Microsoft Specific

Although initialization of base classes is usually restricted to direct base classes, in Microsoft C++, a class constructor can initialize an indirect virtual base class. ♦

Initialization Order of Bases and Members

Base classes and member objects are initialized in the following order:

1. Virtual base classes are initialized in the order in which they appear in the directed acyclic graph. For information about using the directed acyclic graph to construct a list of unique subobjects, see "Virtual Base Classes" on page 308 in Chapter 9. (Note that these subobjects are destroyed by walking the same list in reverse.) For more information about how the directed acyclic graph is traversed, see "Order of Destruction" on page 308.
2. Nonvirtual base classes are initialized in the order in which they are declared in the class declaration.
3. Member objects are initialized in the order in which the objects are declared in the class.

The order in which base classes and member objects are initialized is not affected by the order in which the member initializers or base-class initializers appear in the *member-initializer-list* of the constructor.

Scope of Initializers

Initializers for base classes and member objects are evaluated in the scope of the constructor with which they are declared. Therefore, they can refer implicitly to class-member data.

11.7 Copying Class Objects

Two operations cause objects to be copied:

- Assignment. When one object's value is assigned to another object, the first object is copied to the second object. Therefore:

```
Point a, b;
...
a = b;
```

  causes the value of `b` to be copied to `a`.

- Initialization. Initialization occurs at the point of declaration of a new object, when arguments are passed to functions by value, and when values are returned from functions by value.

The programmer can define the semantics of "copy" for objects of class type. For example, consider the following code:

```
TextFile a, b;
a.Open( "FILE1.DAT" );
b.Open( "FILE2.DAT" );
b = a;
```

The preceding code could mean "copy the contents of FILE1.DAT to FILE2.DAT," or it could mean "ignore FILE2.DAT and make `b` a second handle to FILE1.DAT." The programmer is responsible for attaching appropriate copying semantics to each class.

Copying is done in one of two ways:

- Assignment (using the assignment operator, **operator=**).
- Initialization (using the copy constructor). (For more information about the copy constructor, see "Rules for Declaring Constructors" on page 302.)

Any given class can implement one or both copy methods. If neither method is implemented, assignment is handled as a member-by-member ("memberwise") assignment, and initialization is handled as a member-by-member initialization. Memberwise assignment is covered in more detail in "Memberwise Assignment and Initialization" on page 335.

The copy constructor takes a single argument of type *class-name***&**, where *class-name* is the name of the class for which the constructor is defined. For example:

```
class Window
{
public:
    Window( const Window& ); // Declare copy constructor.
    ...
};
```

Note The type of the copy constructor's argument should be *const class-name***&** whenever possible. This prevents the copy constructor from accidentally changing the object from which it is copying. It also allows copying from **const** objects.

Compiler-Generated Copying

Compiler-generated copy constructors, like user-defined copy constructors, have a single argument of type "reference to *class-name*." An exception is when all base classes and member classes have copy constructors declared as taking a single argument of type **const** *class-name***&**. In such a case, the compiler-generated copy constructor's argument is also **const**.

When the argument type to the copy constructor is not **const**, initialization by copying a **const** object generates an error. The reverse is not true: If the argument is **const**, initialization by copying a non**const** object is allowed.

Compiler-generated assignment operators follow the same pattern with regard to **const.** They take a single argument of type *class-name***&** unless the assignment operators in all base and member classes take arguments of type **const** *class-name***&.** In this case, the class's generated assignment operator takes a **const** argument.

Note When virtual base classes are initialized by copy constructors, compiler-generated or user-defined, they are initialized only once: at the point when they are constructed.

The implications are similar to those of the copy constructor. When the argument type is not **const**, assignment from a **const** object generates an error. The reverse is not true: If a **const** value is assigned to a non**const** value, the assignment succeeds.

For more information about overloaded assignment operators, see "Assignment" in Chapter 12, on page 360.

Memberwise Assignment and Initialization

The methods for default assignment and initialization are "memberwise assignment," and "memberwise initialization," respectively. Memberwise assignment consists of copying one object to the other, a member at a time, as if assigning each member individually. Memberwise initialization consists of copying one object to the other, a member at a time, as if initializing each member individually. The primary difference between the two is that memberwise assignment invokes each member's assignment operator (**operator=**), whereas memberwise initialization invokes the copy constructor.

Memberwise assignment is performed only by the assignment operator declared in the form:

*type***& *type*::operator=([[const | volatile]] *type*&)**

Default assignment operators for memberwise assignment cannot be generated if any of the following conditions exist:

- A member class has **const** members.
- A member class has reference members.
- A member class or its base class has a private assignment operator (**operator=**).
- A base class or member class has no assignment operator (**operator=**).

Default copy constructors for memberwise initialization cannot be generated if the class or one of its base classes has a private copy constructor, or if any of the following conditions exist:

- A member class has **const** members.
- A member class has reference members.
- A member class or its base class has a private assignment operator (**operator**).
- A base class or member class has no assignment operator (**operator**).

The default assignment operators and copy constructors for a given class are always declared, but they are not defined unless both of the following conditions are met:

- The class does not provide a user-defined function for this copy.
- The program requires that the function be present. This requirement exists if an assignment or initialization is encountered that requires memberwise copying, or if the address of the class's **operator=** function is taken.

If both of the conditions in the preceding list are not met, the compiler is not required to generate code for the default assignment operator and copy constructor functions (elimination of such code is an optimization performed by the Microsoft C++ compiler). Specifically, if the class declares a user-defined **operator=** that takes an argument of type "reference to *class-name*," no default assignment operator is generated. If the class declares a copy constructor, no default copy constructor is generated.

Therefore, for a given class `A`, the following declarations are always present:

```
//  Implicit declarations of copy constructor
//   and assignment operator.
A::A( const A& );
A& A::operator=( const A& );
```

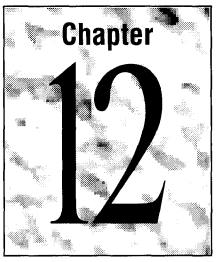
The definitions are supplied only if required (according to the preceding criteria). The copy constructor functions shown in the preceding example are considered public member functions of the class.

Default assignment operators allow objects of a given class to be assigned to objects of a public base-class type. Consider the following code:

```
class Account
{
public:
    // Public member functions
    ...
private:
    double _balance;
};

class Checking : public Account
{
private:
    int _fOverdraftProtect;
};

...

Account account;
Checking checking;

account = checking;
```

In the preceding example, the assignment operator chosen is `Account::operator=`. Because the default `operator=` function takes an argument of type `Account&` (reference to `Account`), the `Account` subobject of `checking` is copied to `account`; `fOverdraftProtect` is not copied.

Overloading



This chapter explains how to use C++ overloaded functions and overloaded operators.

12.1 Overview

The C++ language allows you to overload functions and operators. Overloading is the practice of supplying more than one definition for a given function name in the same scope. The compiler is left to pick the appropriate version of the function or operator based on the arguments with which it is called. For example:

```
double max( double d1, double d2 )
{
    return ( d1 > d2 ) ? d1 : d2;
}

int max( int i1, int i2 )
{
    return ( i1 > i2 ) ? i1 : i2;
}
```

The function `max` is considered an overloaded function. It can be used in code such as the following:

```
main()
{
    int    i = max( 12, 8 );
    double d = max( 32.9, 17.4 );

    return i + (int)d;
}
```

In the first case, where the maximum value of two variables of type `int` is being requested, the function `max( int, int )` is called. However, in the second case, the arguments are of type `double`, so the function `max( double, double )` is called.

Argument Type Differences

Overloaded functions differentiate between argument types that take different initializers. Therefore, arguments of a given type, and a reference to that type are considered the same for the purposes of overloading. They are considered the same because they take the same initializers. For example, `max( double, double )` is considered the same as `max( double &, double & )`. It is an error for overloaded functions to differ only in this manner.

For the same reason, function arguments of a type modified by **const** or **volatile** are not treated differently than the base type for the purposes of overloading.

However, the function overloading mechanism can distinguish between references that are qualified by **const** and **volatile**, and references to the base type. This makes code such as the following possible:

```
#include <iostream.h>

class Over
{
public:
    Over() { cout << "Over default constructor\n"; }
    Over( Over &o ) { cout << "Over&\n"; }
    Over( const Over &co ) { cout << "const Over&\n"; }
    Over( volatile Over &vo ) { cout << "volatile Over&\n"; }
};

main()
{
    Over o1;            // Calls default constructor.
    Over o2( o1 );      // Calls Over( Over& ).
    const Over o3;      // Calls default constructor.
    Over o4( o3 );      // Calls Over( const Over& ).
    volatile Over o5;   // Calls default constructor.
    Over o6( o5 );      // Calls Over( volatile Over& ).

    return 0;
}
```

Pointers to **const** and **volatile** objects are also considered different from pointers to the base type for the purposes of overloading.

Restrictions on Overloaded Functions

There are a number of restrictions that govern what makes up an acceptable set of overloaded functions:

- Any two functions in a set of overloaded functions must have different argument lists.
- Overloading functions with argument lists of the same types, based on return type alone, is an error.

Microsoft Specific

You can overload **operator new** solely on the basis of return type—specifically, on the basis of the memory-model modifier specified.♦

- Member functions cannot be overloaded solely on the basis of one being static and the other nonstatic.
- **typedef** declarations do not define new types; they introduce synonyms for existing types. They do not affect the overloading mechanism. Consider the following code:

```
typedef char * PSTR;

void Print( char *szToPrint );
void Print( PSTR szToPrint );
```

The preceding two functions have identical argument lists. `PSTR` is a synonym for type **char** *. Compiling this code generates an error message.

- Enumerated types are distinct types and can be used to distinguish between overloaded functions.
- The types "array of" and "pointer to" are considered identical for the purposes of distinguishing between overloaded functions. This is true only for singly dimensioned arrays. Therefore, the following overloaded functions conflict and generate an error message:

```
void Print( char *szToPrint );
void Print( char szToPrint[] );
```

For multiply dimensioned arrays, the second and all succeeding dimensions are considered part of the type. Therefore, they are used in distinguishing between overloaded functions:

```
void Print( char szToPrint[] );
void Print( char szToPrint[][7] );
void Print( char szToPrint[][9][42] );
```

12.2 Declaration Matching

Any two function declarations of the same name in the same scope can refer to two discrete functions that are overloaded or to the same function. If the argument lists of the declarations contain arguments of equivalent types (as described in the previous section), the function declarations refer to the same function. Otherwise, they refer to two different functions that are selected using overloading.

Class scope is strictly observed; therefore, a function declared in a base class is not in the same scope as a function declared in a derived class. If a function in a derived class is declared with the same name and type as a function in the base class, the derived-class function hides the base-class function instead of causing overloading.

Block scope is strictly observed; therefore, a function declared in file scope is not in the same scope as a function declared locally. If a locally declared function has the same name and type as a function declared in file scope, the locally declared function hides the file-scoped function instead of causing overloading. For example:

```
#include <iostream.h>

void func( int i )
{
    cout << "Called file-scoped func : " << i << endl;
}

void func( char *sz )
{
    cout << "Called locally-declared func : " << sz << endl;
}

main()
{
    // Declare func local to main.
    extern void func( char *sz );

    func( 3 );    // Error. func( int ) is hidden.
    func( "s" );

    return 0;
}
```

The preceding code shows two definitions from the function `func`. The definition that takes an argument of type `char *` is local to `main` because of the `extern` statement. Therefore, the definition that takes an argument of type `int` is hidden, and the first call to `func` is in error.

For overloaded member functions, different versions of the function can be given different access privileges. They are still considered to be in the scope of the enclosing class, and thus are overloaded functions. Consider the following code in which the member function `Deposit` is overloaded; one version is public, the other, private:

```
class Account
{
public:
    Account();
    double Deposit( double dAmount, char *szPassword );
private:
    double Deposit( double dAmount );
    int    Validate( char *szPassword );
};
```

The intent of the preceding code is to provide an `Account` class in which a correct password is required to perform deposits. This is accomplished using overloading. The following code shows how this class can be used and also shows an erroneous call to the private member, `Deposit`:

```
main()
{
    // Allocate a new object of type Account.
    Account *pAcct = new Account;

    // Deposit $57.22. Error: calls a private function.
    pAcct->Deposit( 57.22 );

    // Deposit $57.22 and supply a password. OK: calls a
    //  public function.
    pAcct->Deposit( 52.77, "pswd" );

    return 0;
}

double Account::Deposit( double dAmount, char *szPassword )
{
    if( Validate( szPassword ) )
        return Deposit( dAmount );
    else
        return 0.0;
}
```

Note that the call to `Deposit` in `Account::Deposit` calls the private member function. This call is correct because `Account::Deposit` is a member function and therefore has access to the private members of the class.

12.3 Argument Matching

Overloaded functions are selected on a best-match basis. That is, the set of overloaded functions is scanned to see which function declaration in the current scope best matches the arguments supplied in the function call. If a suitable function is found, that function is called. "Suitable" in this context means one of the following:

- A standard conversion to the desired argument type exists.
- A user-defined conversion (either conversion operator or constructor) to the desired argument type exists.

The compiler creates a list of candidate functions for each argument. Candidate functions are functions in which the argument in that position exactly matches the type of the supplied argument, or for which a suitable conversion can be found.

A list of "best matching functions" is built for each argument, and the selected function is the intersection of all the lists. If the intersection contains more than one function, the overloading is ambiguous and generates an error. The function that is eventually selected is always a better match than every other function in the group for at least one argument. If this is not the case (if there is no clear winner), the function call generates an error.

Consider the following declarations (the functions are marked `Variant 1`, `Variant 2`, and `Variant 3`, for identification in the following discussion):

```
Fraction &Add( Fraction &f, long l );       // Variant 1
Fraction &Add( long l, Fraction &f );       // Variant 2
Fraction &Add( Fraction &f, Fraction &f );  // Variant 3

Fraction F1, F2;
```

Consider the following statement:

```
F1 = Add( F2, 23 );
```

The preceding statement builds two lists:

List 1: Candidate Functions That Have First Argument of Type Fraction	List 2: Candidate Functions That Have Second Argument of Type int
Variant 1	Variant 1 (**int** can be converted to **long** using a standard conversion)
Variant 3	

The intersection of these two lists is Variant 1. An example of an ambiguous function call is:

```
F1 = Add( 3, 6 );
```

The preceding function call builds the following lists:

List 1: Candidate Functions That Have First Argument of Type int	List 2: Candidate Functions That Have Second Argument of Type int
Variant 2 (**int** can be converted to **long** using a standard conversion)	Variant 1 (**int** can be converted to **long** using a standard conversion)

Note that the intersection between these two lists is empty. Therefore, the compiler generates an error message.

For argument matching, a function with *n* default arguments is treated as *n*+1 separate functions, each with a different number of arguments.

The ellipsis (...) acts as a wildcard; it matches any actual argument. This can lead to many ambiguous lists, if you do not design your overloaded function lists with extreme care.

Note Ambiguity of overloaded functions cannot be determined until a function call is encountered. At that point, the lists are built for each argument in the function call, and it can be determined whether an unambiguous overload exists. This means that ambiguities can remain in your code until they are evoked by a particular function call.

Argument Matching and the this Pointer

Class member functions are treated differently, depending on whether they are declared as **static**. Because nonstatic functions have an implicit argument that supplies the **this** pointer, nonstatic functions are considered to have one more argument than static functions; otherwise, they are declared identically.

These nonstatic member functions require that the implied **this** pointer match the object type through which the function is being called, or, for overloaded operators, they require that the first argument match the object on which the operator is being applied. (For more information about overloaded operators, see "Overloaded Operators" on page 351.)

Unlike other arguments in overloaded functions, no temporary objects are introduced and no conversions are attempted when trying to match the **this** pointer argument.

When the –> member-selection operator is used to access a member function, the **this** pointer argument has a type of *class-name* * **const**. If the members are declared as **const** or **volatile**, the types are **const** *class-name* * **const** and **volatile** *class-name* * **const**, respectively.

The . member-selection operator works exactly the same way, except that an implicit & (address-of) operator is prefixed to the object name. The following example shows how this works:

```
// Expression encountered in code
obj.name

// How the compiler treats it
(&obj)->name
```

The left operand of the –>* and .* (pointer to member) operators are treated the same way as the . and –> (member-selection) operators with respect to argument matching.

Argument Matching and Conversions

When the compiler tries to match actual arguments against the arguments in function declarations, it can supply standard or user-defined conversions to obtain the correct type if no exact match can be found. The application of conversions is subject to these rules:

- Sequences of conversions that contain more than one user-defined conversion are not considered.
- Sequences of conversions that can be shortened by removing intermediate conversions are not considered.

The resultant sequence of conversions, if any, is called the best matching sequence. There are several ways to convert an object of type **int** to type **unsigned long** using standard conversions (described in Chapter 3, "Standard Conversions"):

- Convert from **int** to **long**, then from **long** to **unsigned long**.
- Convert from **int** to **unsigned long**.

The first sequence, while it achieves the desired goal, is not the best matching sequence—a shorter sequence exists.

Table 12.1 shows a group of conversions, called trivial conversions, that have a limited effect on the determination of which sequence is the best matching. The instances in which trivial conversions affect choice of sequence are discussed in the list following the table.

Table 12.1 Trivial Conversions

Convert from Type	Convert to Type
type-name	type-name**&**
*type-name***&**	*type-name*
type-name[]	*type-name**
type-name(*argument-list*)	(**type-name*)(*argument-list*)
type-name	**const** *type-name*
type-name	**volatile** *type-name*
*type-name**	const *type-name**
*type-name**	volatile *type-name**

The sequence in which conversions are attempted is as follows:

1. Exact match. An exact match between the types with which the function is called and the types declared in the function prototype is always the best match. Sequences of trivial conversions are classified as exact matches. However, sequences that do not make any of these conversions are considered better than sequences that convert:
   - From pointer, to pointer to **const** (*type* * to **const** *type* *).
   - From pointer, to pointer to **volatile** (*type* * to **volatile** *type* *).
   - From reference, to reference to **const** (*type* **&** to **const** *type* **&**).
   - From reference, to reference to **volatile** (*type* **&** to **volatile** *type* **&**).
2. Match using promotions. Any sequence not classified as an exact match that contains only integral promotions, conversions from **float** to **double**, and trivial conversions is classified as a match using promotions. While not as good a match as any exact match, a match using promotions is better than a match using standard conversions.
3. Match using standard conversions. Any sequence not classified as an exact match or a match using promotions that contains only standard conversions and trivial conversions is classified as a match using standard conversions. Within this category, the following rules are applied:
   - Conversion from a pointer to a derived class, to a pointer to a direct or indirect base class is preferable to converting to **void** * or **const void** *.
   - Conversion from a pointer to a derived class, to a pointer to a base class produces a better match the closer the base class is to a direct base class. Suppose the class hierarchy is as shown in Figure 12.1.

Figure 12.1 Graph Illustrating Preferred Conversions

Conversion from type `D*` to type `C*` is preferable to conversion from type `D*` to type `B*`. Similarly, conversion from type `D*` to type `B*` is preferable to conversion from type `D*` to type `A*`.

This same rule applies to reference conversions. Conversion from type `D&` to type `C&` is preferable to conversion from type `D&` to type `B&`, and so on.

This same rule applies to pointer-to-member conversions. Conversion from type `T D::*` to type `T C::*` is preferable to conversion from type `T D::*` to type `T B::*`, and so on (where `T` is the type of the member).

The preceding rule applies only along a given path of derivation. Consider the graph shown in Figure 12.2.

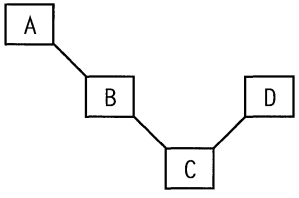


Figure 12.2 Multiple Inheritance Graph Illustrating Preferred Conversions

Conversion from type `C*` to type `B*` is preferable to conversion from type `C*` to type `A*`. The reason is that they are on the same path, and `B*` is closer. However, conversion from type `C*` to type `D*` is not preferable to conversion to type `A*`; there is no preference because the conversions follow different paths.

4. Match with user-defined conversions. This sequence cannot be classified as an exact match, a match using promotions, or a match using standard conversions. The sequence must contain only user-defined conversions, standard conversions, or trivial conversions to be classified as a match with user-defined conversions. A match with user-defined conversions is considered a better match than a match with an ellipsis, but not as good a match as a match with standard conversions.
5. Match with an ellipsis. Any sequence that matches an ellipsis in the declaration is classified as a match with an ellipsis. This is considered the weakest match.

User-defined conversions are applied if no built-in promotion or conversion exists. These conversions are selected on the basis of the type of the argument being matched. Consider the following code:

```
class UDC
{
public:
    operator int();
    operator long();
};

void Print( int i );
...
UDC udc;
Print( udc );
```

The available user-defined conversions for class `UDC` are from type **int** and type **long**. Therefore, the compiler considers conversions for the type of the object being matched: `UDC`. A conversion to **int** exists, and it is selected.

During the process of matching arguments, standard conversions can be applied to both the argument and the result of a user-defined conversion. Therefore, the following code works:

```
void LogToFile( long l );
...
UDC udc;
LogToFile( udc );
```

In the preceding example, the user-defined conversion, **operator long**, is invoked to convert `udc` to type **long**. If no user-defined conversion to type **long** had been defined, the conversion would have proceeded as follows: Type `UDC` would have been converted to type **int** using the user-defined conversion. Then the standard conversion from type **int** to **long** would have been applied to match the argument in the declaration.

If any user-defined conversions are required to match an argument, the standard conversions are not used when evaluating the best match. This is true even if more than one candidate function requires a user-defined conversion; in such a case, the functions are considered equal. For example:

```
class UDC1
{
public:
    UDC1( int );  // User-defined conversion from int.
};

class UDC2
{
public:
    UDC2( long ); // User-defined conversion from long.
};

...

void Func( UDC1 );
void Func( UDC2 );

...

Func( 1 );
```

Both versions of `Func` require a user-defined conversion to convert type **int** to the class type argument. The possible conversions are:

- Convert from type **int** to type `UDC1` (a user-defined conversion).
- Convert from type **int** to type **long**, then convert to type `UDC2` (a two-step conversion).

Even though the second of these requires a standard conversion, as well as the user-defined conversion, the two conversions are still considered equal.

Note User-defined conversions are considered conversion by construction or conversion by initialization (conversion function). Both methods are considered equal when considering the best match.

12.4 Address of Overloaded Functions

Use of a function name without arguments returns the address of that function. For example:

```
int Func( int i, int j );
int Func( long l );

...

int (*pFunc) ( int, int ) = Func;
```

In the preceding example, the first version of `Func` is selected, and its address is copied into `pFunc`.

The compiler determines which version of the function to select by finding a function with an argument list that exactly matches that of the target. The arguments in the overloaded function declarations are matched against one of the following:

- An object being initialized (as shown in the preceding example)
- The left side of an assignment statement
- A formal argument to a function
- A formal argument to a user-defined operator
- A function return type

If no exact match is found, the expression that takes the address of the function is ambiguous and an error is generated.

Note that although a nonmember function, `Func`, was used in the preceding example, the same rules are applied when taking the address of overloaded member functions.

12.5 Overloaded Operators

C++ allows the function of most built-in operators to be redefined. These operators can be redefined, or "overloaded," globally or on a class-by-class basis. Overloaded operators are implemented as functions, and can be class-member or global functions.

The name of an overloaded operator is **operator***x*, where *x* is the operator as it appears in Table 12.2. For example, to overload the addition operator, you define a function called **operator+**. Similarly, to overload the addition/assignment operator, +=, define a function called **operator+=**.

While these operators are usually called implicitly by the compiler when they are encountered in code, they can be invoked explicitly the same way as any member or nonmember function is called:

```
Point pt;

pt.operator+( 3 );  // Call addition operator to add 3 to pt.
```

Table 12.2 shows a list of operators that can be overloaded.

Table 12.2 Redefinable Operators

Operator	Name	Type
,	Comma	Binary
!	Logical negation	Unary
!=	Inequality	Binary
%	Modulus	Binary
%=	Modulus/assignment	Binary
&	Bitwise AND	Binary
&	Address-of	Unary
&&	Logical AND	Binary
&=	Bitwise AND/assignment	Binary
()	Function call	—
*	Multiplication	Binary
*	Pointer dereference	Unary
*=	Multiplication/assignment	Binary
+	Addition	Binary
+	Unary Plus	Unary
++	Increment[1]	Unary
+=	Addition/assignment	Binary
-	Subtraction	Binary
-	Unary negation	Unary
--	Decrement[1]	Unary
-=	Subtraction/assignment	Binary
->	Member selection	Binary
->*	Pointer-to-member selection	Binary
/	Division	Binary
/=	Division/assignment	Binary
<	Less than	Binary
<<	Left shift	Binary
<<=	Left shift/assignment	Binary

Table 12.2 *(continued)*

Operator	Name	Type
<=	Less than or equal to	Binary
=	Assignment	Binary
==	Equality	Binary
>	Greater than	Binary
>=	Greater than or equal to	Binary
>>	Right shift	Binary
>>=	Right shift/assignment	Binary
[]	Array subscript	—
^	Exclusive OR	Binary
^=	Exclusive OR/assignment	Binary
|	Bitwise inclusive OR	Binary
|=	Bitwise inclusive OR/assignment	Binary
||	Logical OR	Binary
~	One's complement	Unary
delete	delete	—
new	new	—

[1]Two versions of the unary increment and decrement operators exist: preincrement and postincrement.

The constraints on the various categories of overloaded operators are described in "Unary Operators" on page 355, "Binary Operators" on page 358, "Assignment" on page 360, "Function Call" on page 361, "Subscripting" on page 362, "Class Member Access" on page 363, and "Increment and Decrement" on page 356.

The operators shown in Table 12.3 cannot be overloaded.

Table 12.3 Nonredefinable Operators

Operator	Name
.	Member selection
.*	Pointer-to-member selection
::	Scope resolution
:>	Base operator
? :	Conditional
#	Preprocessor symbol
##	Preprocessor symbol

General Rules for Operator Overloading

The following rules constrain how overloaded operators are implemented. However, they do not apply to the **new** and **delete** operators, which are covered separately on pages 97 and 101 in Chapter 4, respectively.

- Operators must either be class member functions, or they must take an argument that is of class or enumerated type or arguments that are references to class or enumerated types. For example:

```
class Point
{
public:
    Point operator<( Point & );  // Declare a member operator
                                 //  overload.
    ...
    // Declare addition operators.
    friend Point operator+( Point&, int );
    friend Point operator+( int, Point& );
};
```

  The preceding code sample declares the less-than operator as a member function; however, the addition operators are declared as global functions that have friend access. Note that more than one implementation can be provided for a given operator. In the case of the preceding addition operator, the two implementations are provided to facilitate commutativity. It is just as likely that operators that add a `Point` to a `Point`, and `int` to a `Point`, and so on, might be implemented.

- Operators obey the precedence, grouping, and number of operands dictated by their typical use with built-in types. Therefore, there is no way to express the concept "add 2 and 3 to an object of type `Point`," expecting 2 to be added to the *x* coordinate and 3 to be added to the *y* coordinate.
- Unary operators, if declared as member functions take no arguments; if declared as global functions, they take one argument.
- Binary operators, if declared as member functions take one argument; if declared as global functions, they take two arguments.
- Overloaded operators cannot have default arguments.
- All overloaded operators except assignment (**operator=**) are inherited by derived classes.
- The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked (the class in which the operator is declared, or a class derived from that class). No conversions are supplied for the first argument.

Note that the meaning of any of the operators can be changed completely. That includes the meaning of the address-of (&), assignment (=), and function-call operators. Also, identities that can be relied upon for built-in types can be changed using operator overloading. For example, the following four statements are usually equivalent when completely evaluated:

```
var = var + 1;
var += 1;
var++;
++var;
```

This identity cannot be relied upon for class types that overload operators. Moreover, some of the requirements implicit in the use of these operators for basic types are relaxed for overloaded operators. For example, the addition/assignment operator, +=, requires the left operand to be an l-value when applied to basic types; there is no such requirement when the operator is overloaded.

Note For consistency, it is often best to follow the model of the built-in types when defining overloaded operators. If the semantics of an overloaded operator differ wildly from its meaning in other contexts, it can be more confusing than useful.

Unary Operators

The unary operators are shown in Table 12.4.

Table 12.4 Redefinable Unary Operators

Operator	Name
!	Logical negation
&	Address-of
*	Pointer dereference
+	Unary Plus
++	Increment
–	Unary negation
– –	Decrement

Of the operators shown in Figure 12.4, the postfix increment and decrement operators (++ and – –) are treated separately in "Increment and Decrement" on page 356.

To declare a unary operator function as a nonstatic member, it must be declared in the form:

ret-type **operator***op*()

where *ret-type* is the return type, and *op* is one of the operators listed in Table 12.4.

To declare a unary operator function as a global function, it must be declared in the form:

ret-type **operator***op*(*arg*)

where *ret-type* and *op* are as described for member operator functions, and the *arg* is an argument of class type on which to operate.

Note There is no restriction on the return types of the unary operators. For example, it makes sense for logical negation (!) to return an integral value, but this is not enforced.

Increment and Decrement

The increment and decrement operators fall into a special category because there are two variants of each:

- Preincrement and postincrement
- Predecrement and postdecrement

When you write overloaded operator functions, it can be useful to implement separate versions for the prefix and postfix versions of these operators. To distinguish between the two, the following rule is observed: the prefix form of the operator is declared exactly the same way as any other unary operator; the postfix form accepts an additional argument of type **int**.

Important When specifying an overloaded operator for the postfix form of the increment or decrement operator, the additional argument must be of type **int**; specifying any other type generates an error.

The following example shows how to define prefix and postfix increment and decrement operators for the `Point` class:

```
class Point
{
public:
    // Declare prefix and postfix increment operators.
    Point& operator++();       // Prefix increment operator.
    Point& operator++(int);    // Postfix increment operator.

    // Declare prefix and postfix decrement operators.
    Point& operator--();       // Prefix decrement operator.
    Point& operator--(int);    // Postfix decrement operator.

    // Define default constructor.
    Point() { _x = _y = 0; }
```

```
    // Define accessor functions.
    int x() { return _x; }
    int y() { return _y; }
private:
    int _x, _y;
};

// Define prefix increment operator.
Point& Point::operator++()
{
    _x++;
    _y++;
    return *this;
}

// Define postfix increment operator.
Point& Point::operator++(int)
{
    _x++;
    _y++;
    return *this;
}

// Define prefix decrement operator.
Point& Point::operator--()
{
    _x--;
    _y--;
    return *this;
}

// Define postfix decrement operator.
Point& Point::operator--(int)
{
    _x--;
    _y--;
    return *this;
}
```

The same operators can be defined in file scope (globally) using the following function heads:

```
friend Point& operator++( Point& )      // Prefix increment
friend Point& operator++( Point&, int ) // Postfix increment
friend Point& operator--( Point& )      // Prefix decrement
friend Point& operator--( Point&, int ) // Postfix decrement
```

The argument of type **int** that denotes the postfix form of the increment or decrement operator is not commonly used to pass arguments. It usually contains the value 0. However, it can be used as follows:

```
class Int
{
public:
    Int &operator++( int n );
private:
    int _i;
};

Int& Int::operator++( int n )
{
    if( n != 0 )    // Handle case where an argument is passed.
        _i += n;
    else
        _i++;       // Handle case where no argument is passed.

    return *this;
}

...

Int i;
i.operator++( 25 ); // Increment by 25.
```

No syntax exists for using the increment or decrement operators to pass these values other than explicit invocation, as shown in the preceding code. A more straightforward way to implement this functionality is to overload the addition/assignment operator (+=).

Binary Operators

Table 12.5 shows a list of operators that can be overloaded.

Table 12.5 Redefinable Binary Operators

Operator	Name
,	Comma
!=	Inequality
%	Modulus
%=	Modulus/assignment
&	Bitwise AND
&&	Logical AND
&=	Bitwise AND/assignment
*	Multiplication

Table 12.5 (*continued*)

Operator	Name
*=	Multiplication/assignment
+	Addition
+=	Addition/assignment
-	Subtraction
-=	Subtraction/assignment
->	Member selection
->*	Pointer-to-member selection
/	Division
/=	Division/assignment
<	Less than
<<	Left shift
<<=	Left shift/assignment
<=	Less than or equal to
=	Assignment
==	Equality
>	Greater than
>=	Greater than or equal to
>>	Right shift
>>=	Right shift/assignment
^	Exclusive OR
^=	Exclusive OR/assignment
|	Bitwise inclusive OR
|=	Bitwise inclusive OR/assignment
||	Logical OR

To declare a binary operator function as a nonstatic member, it must be declared in the form:

ret-type **operator***op*(*arg*)

where *ret-type* is the return type, and *op* is one of the operators listed in Table 12.5, and *arg* is an argument of any type.

To declare a binary operator function as a global function, it must be declared in the form:

ret-type **operator***op*(*arg1*, *arg2*)

where *ret-type* and *op* are as described for member operator functions, and *arg1* and *arg2* are arguments. At least one of the arguments must be of class type.

Note There is no restriction on the return types of the binary operators; however, most user-defined binary operators return either a class type or a reference to a class type.

Assignment

The assignment operator (=) is, strictly speaking, a binary operator. Its declaration is identical to any other binary operator with the following exceptions:

- It must be a nonstatic member function. No **operator=** can be declared as a non-member function.
- It is not inherited by derived classes.
- A default **operator=** function can be generated by the compiler for class types if none exists. (For more information about default **operator=** functions, see "Memberwise Assignment and Initialization" in Chapter 11, on page 335.)

The following example illustrates how to declare an assignment operator:

```
class Point
{
public:
    Point &operator=( Point & );  // Right side is the argument.
    ...
};

// Define assignment operator.
Point &Point::operator=( Point &ptRHS )
{
    _x = ptRHS._x;
    _y = ptRHS._y;

    return *this;  // Assignment operator returns left side.
}
```

Note that the supplied argument is the right side of the expression. The operator returns the object to preserve the behavior of the assignment operator, which returns the value of the left side after the assignment is complete. This allows writing statements such as:

```
pt1 = pt2 = pt3;
```

Function Call

The function-call operator, invoked using parentheses, is a binary operator. The syntax for a function call is:

Syntax

primary-expression (*expression-list*$_{opt}$)

In this context, *primary-expression* is the first operand and *expression-list*, a possibly empty list of arguments, is the second operand. The function-call operator is used for operations that require a number of parameters. This works because *expression-list* is a list instead of a single operand. The function-call operator must be a nonstatic member function.

The function-call operator, when overloaded, does not modify how functions are called; rather, it modifies how the operator is to be interpreted when applied to objects of a given class type. For example, the following code would usually be meaningless:

```
Point pt;
pt( 3, 2 );
```

Given an appropriate overloaded function-call operator, however, this syntax can be used to offset the *x* coordinate 3 units and the *y* coordinate 2 units. The following code shows such a definition:

```
class Point
{
public:
    Point() { _x = _y = 0; }
    Point &operator()( int dx, int dy )
        { _x += dx; _y += dy; return *this; }
private:
    int _x, _y;
};

...

Point pt;
pt( 3, 2 );
```

Note that the function-call operator is applied to the name of an object, not the name of a function.

Subscripting

The subscript operator ([ ]), like the function-call operator, is considered a binary operator. The subscript operator must be a nonstatic member function that takes a single argument. This argument can be of any type and designates the desired array subscript.

The following example demonstrates how to create a vector of type **int** that implements bounds checking:

```
#include <iostream.h>

class IntVector
{
public:
    IntVector( int cElements );
    ~IntVector() { delete _iElements; }
    int& operator[]( int nSubscript );
private:
    int *_iElements;
    int _iUpperBound;
};

// Construct an IntVector.
IntVector::IntVector( int cElements )
{
    _iElements = new int[cElements];
    _iUpperBound = cElements;
}

// Subscript operator for IntVector.
int& IntVector::operator[]( int nSubscript )
{
    static int iErr = -1;

    if( nSubscript >= 0 && nSubscript < _iUpperBound )
        return _iElements[nSubscript];
    else
    {
        clog << "Array bounds violation." << endl;
        return iErr;
    }
}
```

```
// Test the IntVector class.
main()
{
    IntVector v( 10 );

    for( int i = 0; i <= 10; ++i )
        v[i] = i;

    v[3] = v[9];

    for( i = 0; i <= 10; ++i )
        cout << "Element: [" << i << "] = " << v[i]
             << endl;

    return v[0];
}
```

When `i` reaches 10 in the preceding program, `operator[]` detects that an out-of-bounds subscript is being used and issues an error message.

Note that the function `operator[]` returns a reference type. This causes it to be an l-value, allowing you to use subscripted expressions on either side of assignment operators.

Class-Member Access

Class-member access can be controlled by overloading the member-selection operator (**–>**). This operator is considered a unary operator, and the overloaded operator function must be a class member function. Therefore, the declaration for such a function is:

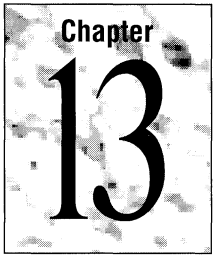
class-type ***operator–>()**

where *class-type* is the name of the class to which this operator belongs. The member-selection operator function must be a nonstatic member function.

This operator is used (often in conjunction with the pointer-defererence operator) to implement "smart pointers" that validate pointers prior to dereference or count usage.

The **.** member-selector operator cannot be overloaded.

Preprocessing



A "preprocessor directive" is an instruction to the C++ preprocessor. Preprocessing takes place during the first phase of compilation. This chapter describes the preprocessing translation phase.

This chapter also discusses macros, the **#define** directive, the **#undef** directive, and the four preprocessor operators. A "preprocessor operator" is an operator that is only recognized as an operator within the context of preprocessor directives.

Include files (text files inserted into a program at the location of a **#include** directive), conditional compilation, the **#error** and **#line** directives, and pragmas are the topics of sections at the end of this chapter. A "pragma" is a "pragmatic," or practical, instruction to the C++ compiler. Pragmas in C++ source files are typically used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole.

13.1 The Preprocessor

The C++ preprocessor is a text processor that manipulates the text of a source file as part of the first phase of translation. (For more information, see Appendix A, "Phases of Translation.") The preprocessor does not parse the source text, but it does break it up into tokens for the purpose of locating macro calls. Although the compiler ordinarily invokes the preprocessor in its first pass, the preprocessor can also be invoked separately to process text without compiling.

Microsoft Specific

You can obtain a listing of your source code after preprocessing by using the /E or /EP compilation options. Both options cause the preprocessor to be invoked and the resulting text to be output to the standard output device which, in most cases, is the console. The difference between the two options is that /E includes **#line** directives, and /EP strips these directives out. ♦

Preprocessor directives are typically used to make source programs easy to change and easy to compile in different execution environments. Directives in the source file tell the preprocessor to perform specific actions. For example, the

preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text. Preprocessor lines are recognized and carried out before macro expansion. Therefore, if a macro expands into something that looks like a preprocessor command, that command is not recognized by the preprocessor.

Preprocessor statements use the same character set as source file statements with the exception that escape sequences are not supported. The character set used in preprocessor statements is the same as the execution character set. (For information on the execution character set, see "Character Constants" in Chapter 1, on page 16.) The preprocessor also recognizes negative character values.

The C++ preprocessor recognizes the following directives:

#define	**#error**	**#include**
#elif	**#if**	**#line**
#else	**#ifdef**	**#pragma**
#endif	**#ifndef**	**#undef**

The number sign (#) must be the first nonwhite-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Any text that follows a directive (except an argument or value that is part of the directive) must be preceded by the single-line comment delimiter (//) or enclosed in comment delimiters (/* */). Lines containing preprocessor directives can be continued by immediately preceding the end-of-line marker with a backslash (\).

Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file.

13.2 Macros

Preprocessing expands macros in all lines that are not preprocessor directives (lines that do not have a # as the first nonwhite-space character) and in parts of some directives that are not skipped as part of a conditional compilation. (For more information, see "Conditional Compilation" on page 379.) The **#define** directive is typically used to associate meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers that represent constants are called "symbolic constants." Identifiers that represent statements or expressions are called "macros." Macros have their own name space. For information, see "Name Spaces" in Chapter 2, on page 61.

When the name of the macro is recognized in the program source text or in the arguments of certain other preprocessor commands, it is treated as a call to that macro. The macro name is replaced by a copy of the macro body. If the macro accepts arguments, the actual arguments following the macro name are substituted for formal parameters in the macro body. The process of replacing a macro call with the processed copy of the body is called "expansion" of the macro call.

In practical terms, there are two types of macros. "Object-like" macros take no arguments, while "function-like" macros can be defined to accept arguments so that they look and act like function calls. Because macros do not generate actual function calls, you can make programs run faster by replacing function calls with macros (although C++ inline functions are often a preferred method). However, macros can create problems if you do not define and use them with care. You may have to use parentheses in macro definitions with arguments to preserve the proper precedence in an expression. Also, macros may not correctly handle expressions with side effects. See the `getrandom` example in "The #define Directive" on page 368 for more information.

Once you have defined an identifier, you cannot redefine it to a different value without first removing the original definition. However, you can redefine the identifier with exactly the same definition. Thus, the same definition can appear more than once in a program.

The **#undef** directive removes the definition of an identifier. Once you have removed the definition, you can redefine the identifier to a different value. "The #define Directive" on page 368 and "The #undef Directive" on page 373 discuss the **#define** and **#undef** directives, respectively.

The Role of Preprocessing in C++

C++ offers new capabilities, some of which supplant those offered by the ANSI C preprocessor. These new capabilities enhance the type safety and predictability of the language:

- In C++, objects declared as **const** can be used in constant expressions. This allows programs to declare constants that have type and value information. Using the preprocessor **#define** directive to define constants is not as precise. No storage is allocated for a **const** object unless an expression that takes its address is found in the program.

- The C++ inline function capability supplants function-type macros. The advantages of using inline functions over macros are:
  - Type-safety. Inline functions are subject to the same type checking as normal functions. Macros are not type safe.
  - Correct handling of arguments that have side effects. Inline functions evaluate the expressions supplied as arguments prior to entering the function body. Therefore, there is no chance that an expression with side effects will be unsafe.

For backward compatibility, all preprocessor facilities that existed in ANSI C and in earlier C++ specifications are preserved for Microsoft C++.

The #define Directive

You can use the **#define** directive to give a meaningful name to a constant in your program. The two forms of the syntax are

Syntax

#define *identifier token-string*$_{opt}$

#define *identifier* (*identifier* , ... , *identifier*) *token-string*$_{opt}$

The **#define** directive substitutes *token-string* for all subsequent occurrences of an *identifier* in the source file. The *identifier* is replaced only when it forms a token. (For information on tokens, see "Tokens" in Chapter 1, on page 2.) For instance, *identifier* is not replaced if it appears in a comment, within a string, or as part of a longer identifier.

A **#define** without a *token-string* removes occurrences of *identifier* from the source file. The *identifier* remains defined and can be tested using the **#if defined** and **#ifdef** directives.

The *token-string* argument consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate *token-string* from *identifier*. This white space is not considered part of the substituted text, nor is any white space following the last token of the text.

Formal parameter names appear in *token-string* to mark the places where actual values are substituted. Each parameter name can appear more than once in *token-string*, and the names can appear in any order. The number of arguments in the call must match the number of parameters in the macro definition. Liberal use of parentheses ensures that complicated actual arguments are not interpreted correctly.

With the second syntax form, an optional list of parameters for a macro appears in parentheses. References to the *identifier* after the original definition replace each

occurrence of *identifier*(*identifier*$_{opt}$, ..., *identifier*$_{opt}$) with a version of the *token-string* argument that has actual arguments substituted for formal parameters.

The formal parameters in the list are separated by commas. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces can separate *identifier* and the opening parenthesis. Use line concatenation—place a backslash (\) before the newline character—for long directives on multiple source lines. The scope of a formal parameter name extends to the new line that ends *token-string*.

When a macro has been defined in the second syntax form, subsequent textual instances followed by an argument list constitute a macro call. The actual arguments following an instance of *identifier* in the source file are matched to the corresponding formal parameters in the macro definition. Each formal parameter in *token-string* that is not preceded by a stringizing (#), charizing (#@), or token-pasting (##) operator, or followed by a ## operator, is replaced by the corresponding actual argument. Any macros in the actual argument are expanded before it replaces the formal parameter. (The operators are described in "Preprocessor Operators" on page 370.)

The following examples of macros with arguments illustrate the second form of the **#define** syntax:

```
// Macro to define cursor lines
#define CURSOR(top, bottom) ((top) << 8) | bottom))

// Macro to get a random integer with a specified range
#define getrandom(min, max) \
    ((rand()%(int)(((max) + 1)-(min)))+ (min))
```

Arguments with side effects sometimes cause macros to produce unexpected results. A given formal parameter may appear more than once in *token-string*. If that formal parameter is replaced by an expression with side effects, the expression, with its side effects, may be evaluated more than once (see examples in "Token-Pasting Operator (##)" on page 373).

Note As mentioned above, unexpected results such as those caused by calling the `getrandom` macro with an expression such as `getrandom(i++, j)` can be eliminated using inline functions instead of macros.

The **#undef** directive causes an identifier's preprocessor definition to be forgotten. See "The #undef Directive" on page 373.

If the name of the macro being defined occurs in *token-string* (even as a result of another macro expansion), it is not expanded.

A second **#define** for the same identifier generates an error unless the second token sequence is identical to the first.

Microsoft Specific

Microsoft C version 6.0 allows a macro to be redefined provided it is lexically identical to the previous definition. ANSI C considers macro redefinition an error. The Microsoft C/C++ compiler allows this behavior but generate warnings. For example, these macros are equivalent for C/C++ but generate warnings:

```
#define test( f1, f2 ) ( f1 * f2 )
#define test( a1, a2 ) ( a1 * a2 )♦
```

This example illustrates the **#define** directive:

```
#define WIDTH       80
#define LENGTH      ( WIDTH + 10 )
```

The first statement defines the identifier `WIDTH` as the integer constant 80 and defines `LENGTH` in terms of `WIDTH` and the integer constant 10. Each occurrence of `LENGTH` is replaced by `( WIDTH + 10 )`. In turn, each occurrence of `WIDTH + 10` is replaced by the expression `( 80 + 10 )`. The parentheses around `WIDTH + 10` are important because they control the interpretation in statements such as the following:

```
var = LENGTH * 20;
```

After the preprocessing stage the statement becomes

```
var = ( 80 + 10 ) * 20;
```

which evaluates to 1800. Without parentheses, the result is

```
var = 80 + 10 * 20;
```

which evaluates to 280.

Microsoft Specific

Defining macros and constants with the /D command-line option has the same effect as using a **#define** preprocessing directive at the beginning of your file. Up to 30 macros can be defined with the /D option.♦

Preprocessor Operators

Four preprocessor-specific operators are used in the context of the **#define** directive. (See "The #if, #elif, #else, and #endif Directives" on page 379 for information on the **defined** operator.) This list gives a short summary of each of the preprocessor directives described in the following sections:

Operator	Action
Stringizing operator (#)	Causes the corresponding actual argument to be enclosed in double quotation marks
Charizing operator (#@)	Causes the corresponding argument to be enclosed in single quotation marks and to be treated as a character
Token-pasting operator (##)	Allows tokens used as actual arguments to be concatenated to form other tokens
defined operator	Simplifies the writing of compound expressions in certain macro directives

Stringizing Operator (#)

The number-sign or "stringizing" operator (#) converts macro parameters (after expansion) to string constants. It is used only with macros that take arguments. If it precedes a formal parameter in the macro definition, the actual argument passed by the macro invocation is enclosed in quotation marks and treated as a string literal. The string literal then replaces each occurrence of a combination of the stringizing operator and formal parameter within the macro definition.

White space preceding the first token of the actual argument and following the last token of the actual argument is ignored. Any white space between the tokens in the actual argument is reduced to a single white space in the resulting string literal. Thus, if a comment occurs between two tokens in the actual argument, it is reduced to a single white space. The resulting string literal is automatically concatenated with any adjacent string literals from which it is separated only by white space.

Further, if a character contained in the argument usually requires an escape sequence when used in a string literal (for example, the quotation mark (") or backslash (\) character), the necessary escape backslash is automatically inserted before the character. The following example shows a macro definition that includes the stringizing operator and a main function that invokes the macro:

```
#define stringer( x ) printf( #x "\n" )

main()
{
    stringer( In quotes in the printf function call\n );
    stringer( "In quotes when printed to the screen"\n );
    stringer( "This: \"  prints an escaped double quote" );
}
```

Such invocations would be expanded during preprocessing, producing the following code:

```
main()
{
    printf( "In quotes in the printf function call\n" "\n" );
    printf( "\"In quotes when printed to the screen\"\n" "\n" );
    printf( "\"This: \\\" prints an escaped double quote\"" "\n" );
}
```

When the program is run, screen output for each line is as follows:

```
In quotes in the printf function call

"In quotes when printed to the screen"

"This: \" prints an escaped double quotation mark"
```

Microsoft Specific

The Microsoft C (versions 6.0 and earlier) extension to the ANSI C standard that previously expanded macro formal arguments appearing inside string literals and character constants is no longer supported. Code that relied on this extension should be rewritten using the stringizing (#) operator.♦

Charizing Operator (#@)

The charizing operator can be used only with arguments of macros. If #@ precedes a formal parameter in the definition of the macro, the actual argument is enclosed in single quotation marks and treated as a character when the macro is expanded. For example:

```
#define makechar(x)  #@x
```

causes the statement

```
a = makechar(b);
```

to be expanded into

```
a = 'b';
```

The single-quotation character cannot be used with the charizing operator.

Token-Pasting Operator (##)

The double-number-sign or "token-pasting" operator (##), which is sometimes called the "merging" operator, is used in both object-like and function-like macros. It permits separate tokens to be joined into a single token, and therefore cannot be the first or last token in the macro definition.

If a formal parameter in a macro definition is preceded or followed by the token-pasting operator, the formal parameter is immediately replaced by the unexpanded actual argument. Macro expansion is not performed on the argument prior to replacement.

Then each occurrence of the token-pasting operator in *token-string* is removed, and the tokens preceding and following it are concatenated. The resulting token must be a valid token. If it is, the token is scanned for possible replacement if it represents a macro name. The identifier represents the name by which the concatenated tokens will be known in the program before replacement. Each token represents a token defined elsewhere, either within the program or on the compiler command line. White space preceding or following the operator is optional.

This example illustrates use of both the "stringizing" and "token-pasting" operators in specifying program output:

```
#define paster( n ) printf( "token" #n " = %d", token##n )
```

If `token9` is declared and the macro is called with a numeric argument like

```
paster( 9 );
```

the macro yields

```
printf( "token" "9" " = %d", token9 );
```

which becomes

```
printf( "token9 = %d", token9 );
```

The #undef Directive

As its name implies, the **#undef** directive removes (undefines) a name previously created with **#define**.

Syntax

#undef *identifier*

The **#undef** directive removes the current definition of *identifier*. Consequently, subsequent occurrences of *identifier* are ignored by the preprocessor. To remove a macro definition using **#undef**, give only the macro *identifier*; do not give a parameter list.

You can also apply the **#undef** directive to an identifier that has no previous definition. This ensures that the identifier is undefined. Macro replacement is not performed within **#undef** statements.

The **#undef** directive is typically paired with a **#define** directive to create a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The **#undef** directive also works with the **#if** directive (see "The #undef Directive" on page 373) to control conditional compilation of the source program.

In the following example, the **#undef** directive removes definitions of a symbolic constant and a macro. Note that only the identifier of the macro is given.

```
#define WIDTH           80
#define ADD( X, Y )     (X) + (Y)
.
.
.
#undef WIDTH
#undef ADD
```

Microsoft Specific

Macros can be undefined from the command line using the /U option, followed by the macro names to be undefined. The effect of issuing this command is equivalent to a sequence of **#undef** *macro-name* statements at the beginning of the file.♦

Predefined Macros

The C++ compiler recognizes five predefined ANSI C macros, and the Microsoft C++ implementation provides several more. The names of the ANSI predefined macros begin and end with two underscores. These macros take no arguments and cannot be redefined. Their value must be constant throughout compilation.

Table 13.1 Predefined Macros

Identifier	Compatibility	Value
_CHAR_UNSIGNED	Microsoft	Defined only when the /J compiler option is given to make char unsigned by default.
__cplusplus	ANSI C++	The value of this macro is not significant. If it is defined, the program is compiled as C++. This macro is not defined for translation units compiled as C.
__DATE__	ANSI C, C++	The translation date of the current source file. The date is a character string of the form `"Mmm dd yyyy"`. The quotes are included to form a proper C++ string.

Table 13.1 *(continued)*

Identifier	Compatibility	Value
_DLL	Microsoft	Defined for run-time library as a DLL (/MD compiler option).
_FAST	Microsoft	Defined if /f option (fast compile) is used.
__FILE__	ANSI C, C++	The name of the current source file. __FILE__ expands to a string surrounded by double quotes.
__LINE__	ANSI C, C++	The line number in the current source file. The line number is a decimal number.
_M_I286	Microsoft	Defined for 80286 processor (/G2 compiler option).
_M_I386	Microsoft	Defined for 80386 processor (flat-model compilation).
_M_I8086	Microsoft	Defined for 8086 and 8088 processors (default or /G0 compiler option).
_M_I86	Microsoft	Always defined. Identifies target machine as a member of the 8086 family.
_M_I86*m*M	Microsoft	Always defined. Identifies memory model, where *m* is either S (small or tiny model), C (compact model), M (medium model), L (large model), or H (huge model). If huge model is used, both M_I86LM and M_I86HM are defined. Small model is the default. For more information about memory models, see Appendix B, "Microsoft-Specific Modifiers."
_MSC_VER	Microsoft	Defines the compiler version as a string literal in the form: ddd. For Microsoft C/C++ version 7.0, the string is `"700"`.
MSDOS	Microsoft	Always defined. Identifies target operating system as MS-DOS.
_MT	Microsoft	Defined for multithread library (/MT, /ML, or /MD compiler option).
NO_EXT_KEYS	Microsoft	No longer emitted by the compiler. This macro was defined in previous versions of Microsoft C for compilations that used the /Za (ANSI-conformance) option. In C 7.0, the __STDC__ macro is used instead.
__PCODE	Microsoft	Defined for sections of code that are compiled as p-code.

Table 13.1 *(continued)*

Identifier	Compatibility	Value
_QC	Microsoft	Defined if /f option (fast compile) is used.
__STDC__	ANSI C	1 if /Za (ANSI-conformance) option is used; otherwise undefined.
__TIME__	ANSI C, C++	The translation time of the current source file. The time is a character string of the form `"hh:mm:ss"`. The quotes are included to form a proper C++ string.
__TIMESTAMP__	Microsoft	The date and time of translation of the current translation unit. The timestamp is a character string of the form `"Ddd Mmm dd hh:mm:ss"`. The quotes are included to form a proper C++ string.

13.3 Include Files

The **#include** directive tells the preprocessor to treat the contents of the named file as if it appeared in the source program at the point where the directive appears. You can organize constant and macro definitions into include files and then use **#include** directives to add these definitions to any source file. Include files are also useful for incorporating declarations of external variables and complex data types. You only need to define and name the types once in an include file created for that purpose.

Syntax

#include " *q-char-sequence* "

#include < *h-char-sequence* >

q-char-sequence:
any sequence of characters except " and
the end-of-line character specifying a filename

h-char-sequence:
any sequence of characters except < > and
the end-of-line character specifying a filename

Both forms cause replacement of that directive by the entire contents of the source file given. The difference between the two forms is how they search for header files when the path is incompletely specified:

Syntax Form	Action
Quoted form	This form, specified as **#include "filename.h"**, instructs the preprocessor to look in the parent directory and paths specified using the INCLUDE environment variable or the /I compile option for include files.
Angle-bracket form	This form, specified as **#include <filename.h>**, causes the preprocessor to search along the path specified with the INCLUDE environment variable and the /I compile option for header files.

The *q-char-sequence* and *h-char-sequence* are filenames optionally preceded by a directory specification. The filename must name an existing file. The syntax of the file specification depends on the operating system on which the program is compiled.

The preprocessor stops searching as soon as it finds a file with the given name. If you specify a complete, unambiguous path specification for the include file between two sets of double quotation marks (" "), the preprocessor searches only that path specification and ignores the standard directories.

If the filename enclosed in double quotation marks is an incomplete path specification, the preprocessor first searches the "parent" file's directory. A parent file is the file containing the **#include** directive. For example, if you include a file named `file2` within a file named `file1`, `file1` is the parent file.

Include files can be "nested"; that is, an **#include** directive can appear in a file named by another **#include** directive. For example, `file2`, above, could include `file3`. In this case, `file1` would still be the parent of `file2`, but would be the "grandparent" of `file3`.

When include files are nested, directory searching begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source currently being processed. If the file is not found, the search moves to directories specified on the compiler command line. Finally, the standard directories are searched. For example:

```
#include <stdio.h>
```

This statement adds the contents of the file named STDIO.H to the source program. The angle brackets cause the preprocessor to search the standard directories for STDIO.H, after searching directories specified on the command line.

For file specifications enclosed in angle brackets, the preprocessor does not search the current working directory. It begins by searching for the file in the directories specified on the compiler command line, then in the standard directories specified

in the INCLUDE environment variable. The following example shows file inclusion using the quoted form:

```
#include "defs.h"
```

This example adds the contents of the file specified by DEFS.H to the source program. The double quotation marks mean that the preprocessor searches the directory containing the "parent" source file first.

Nesting of include files can continue up to 10 levels. Once the nested **#include** is processed, the preprocessor continues to insert the enclosing include file into the original source file.

Microsoft Specific

To locate includable source files, the preprocessor first searches the directories specified by the CL option /I. If the /I option is not present or fails, the preprocessor uses the INCLUDE environment variable to find any include files within angle brackets. If more than one directory appears as part of the /I option or within the INCLUDE environment variable, the preprocessor searches them in the order in which they appear.

For example, the command

```
CL /ID:\C700\INCLUDE MYPROG.C
```

causes the preprocessor to search the directory D:\C700\INCLUDE for include files such as STDIO.H. The commands

```
SET INCLUDE = D:\C700\INCLUDE
CL MYPROG.C
```

have the same effect. If both sets of searches fail, a fatal error is generated. ♦

Microsoft Specific

If the filename is fully specified for an include file with a path that includes a colon (for example, F:\C700\SPECIAL\INCL\TEST.H), the preprocessor follows the path.

If the filename is not fully specified, the preprocessor searches the directory of the file that included it. If the file is not found there, the preprocessor searches the parent directory, the parent's parent, and so on, terminating with the root directory. If the include file is not found in any of these directories, the rules specified in the previous Microsoft Specific note apply. ♦

13.4 Conditional Compilation

This section describes the syntax and use of directives that control "conditional compilation." These directives allow you to suppress compilation of parts of a source file by testing a constant expression or identifier to determine which text blocks are passed on to the compiler and which text blocks are removed from the source file during preprocessing.

The #if, #elif, #else, and #endif Directives

The **#if** directive, with the **#elif**, **#else**, and **#endif** directives, control compilation of portions of a source file. If the expression you write (after the **#if**) has a nonzero value, the line group immediately following the **#if** directive is retained in the translation unit.

Syntax

conditional:
  if-part elif-parts$_{opt}$ *else-part*$_{opt}$ *endif-line*

if-part:
  if-line text

if-line:
  #if *constant-expression*
  #ifdef *identifier*
  #ifndef *identifier*

elif-parts:
  elif-line text
  elif-parts elif-line text

elif-line:
  #elif *constant-expression*

else-part:
  else-line text

else-line:
  #else

endif-line:
  #endif

Each **#if** directive in a source file must be matched by a closing **#endif** directive. Any number of **#elif** directives can appear between the **#if** and **#endif** directives, but at most one **#else** directive is allowed. The **#else** directive, if present, must be the last directive before **#endif**.

The **#if**, **#elif**, **#else**, and **#endif** directives can nest in the text portions of other **#if** directives. Each nested **#else**, **#elif**, or **#endif** directive belongs to the closest preceding **#if** directive.

All conditional-compilation directives such as **#if** and **#ifdef** must be matched with closing **#endif** directives prior to the end of file, or an error message is generated. When conditional compilation directives are contained in include files, they must satisfy the same conditions: there must be no unmatched conditional-compilation directives at the end of the include file.

Macro replacement is performed within the part of the command line that follows an **#elif** command, so a macro call can be used in the *constant-expression.*

The preprocessor selects one of the given occurrences of *text* for further processing. A block specified in *text* can be any sequence of text. It can occupy more than one line. Usually *text* is program text that has meaning to the compiler or the preprocessor.

The preprocessor processes the selected *text* and passes it to the compiler. If *text* contains preprocessor directives, the preprocessor carries out those directives. Any text blocks not selected by the preprocessor are not compiled.

The preprocessor selects a single *text* item by evaluating the constant expression following each **#if** or **#elif** directive until it finds a true (nonzero) constant expression. It selects all text (including other preprocessor directives beginning with #) up to its associated **#elif**, **#else**, or **#endif**.

If all occurrences of *constant-expression* are false, or if no **#elif** directives appear, the preprocessor selects the text block after the **#else** clause. If the **#else** clause is omitted and all instances of *constant-expression* in the **#if** block are false, no text block is selected.

The *constant-expression* is an integer constant expression with these additional restrictions:

- Expressions must have integral type and can only include integer constants, character constants, and the **defined** operator.
- The expression cannot use **sizeof** or a type-cast operator.
- The target environment may not be able to represent all ranges of integers.
- The translation represents type **int** the same as type **long** and **unsigned int** the same as **unsigned long**.
- The translator can translate character constants to a set of code values different from the set for the target environment. To determine the properties of the target environment, check values of macros from LIMITS.H in an application built for the target environment.

- The expression must not perform any environmental inquiries and must remain insulated from implementation details on the target computer.

The preprocessor operator **defined** can be used in special constant expressions, as shown by the following syntax:

Syntax

defined (*identifier*)

defined *identifier*

This constant expression is considered true (nonzero) if the *identifier* is currently defined; otherwise, the condition is false (0). An identifier defined as empty text is considered defined. The **defined** directive can be used in a **#if** and a **#elif** directive, but nowhere else.

In the following example, the **#if** and **#endif** directives control compilation of one of three function calls:

```
#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif
```

The function call to `credit` is compiled if the identifier `CREDIT` is defined. If the identifier `DEBIT` is defined, the function call to `debit` is compiled. If neither identifier is defined, the call to `printerror` is compiled. Note that `CREDIT` and `credit` are distinct identifiers in C++ because their case is different.

These next conditional compilation statements assume a previously defined symbolic constant named `DLEVEL`.

```
#if DLEVEL > 5
    #define SIGNAL  1
    #if STACKUSE == 1
        #define STACK   200
    #else
        #define STACK   100
    #endif
#else
    #define SIGNAL  0
    #if STACKUSE == 1
        #define STACK   100
    #else
        #define STACK   50
    #endif
#endif
```

```
#if DLEVEL == 0
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
    display( debugptr );
#else
    #define STACK 200
#endif
```

The first **#if** block shows two sets of nested **#if**, **#else**, and **#endif** directives. The first set of directives is processed only if `DLEVEL > 5` is true. Otherwise, the statements after **#else** are processed.

The **#elif** and **#else** directives in the second example are used to make one of four choices, based on the value of `DLEVEL`. The constant `STACK` is set to 0, 100, or 200, depending on the definition of `DLEVEL`. If `DLEVEL` is greater than 5, then the statement

```
#elif DLEVEL > 5
display(debugptr);
```

is compiled and `STACK` is not defined.

A common use for conditional compilation is to prevent multiple inclusions of the same header file. In C++, where classes are often defined in header files, constructs like the following can be used to prevent multiple definitions:

```
// EXAMPLE.H - Example header file

#if !defined( EXAMPLE_H )
#define EXAMPLE_H

class Example
{
...
};

#endif // !defined( EXAMPLE_H )
```

The preceding code checks to see if the symbolic constant `EXAMPLE_H` is defined. If so, the file has already been included and need not be reprocessed. If not, the constant `EXAMPLE_H` is defined to mark EXAMPLE.H as already processed.

Microsoft Specific

Conditional compilation expressions are treated as **signed long** values, and these expressions are evaluated using the same rules as expressions in C++. For example, this expression:

```
#if 0xFFFFFFFFL > 1UL
```

is true.♦

The #ifdef and #ifndef Directives

The **#ifdef** and **#ifndef** directives perform the same task as the **#if** directive when it is used with **defined**(*identifier*).

Syntax

#ifdef *identifier*

#ifndef *identifier*

is equivalent to

#if defined *identifier*

#if !defined *identifier*

You can use the **#ifdef** and **#ifndef** directives anywhere **#if** can be used. The **#ifdef** *identifier* statement is equivalent to `#if 1` when *identifier* has been defined, and is equivalent to `#if 0` when *identifier* has not been defined or has been undefined with the **#undef** directive. These directives check only for the presence or absence of identifiers defined with **#define**, not for identifiers declared in the C++ source code.

These directives are provided only for compatibility with previous versions of the language. The **defined**(*identifier*) constant expression used with the **#if** directive is preferred.

The **#ifndef** directive checks for the opposite of the condition checked by **#ifdef**. If the identifier has not been defined (or its definition has been removed with **#undef**), the condition is true (nonzero). Otherwise, the condition is false (0).

Microsoft Specific

The *identifier* can be passed from the command line using the /D option. Up to 30 macros can be specified with /D.

This is useful for checking if a definition exists since a definition can be passed from the command line. For example:

```
// PROG.CPP
#ifndef test    // These three statements go in your source code.
#define final
#endif

CL /Dtest prog.cpp // This is the command for compilation.♦
```

The Null Directive (#)

The null preprocessor directive is a single number sign (#) alone on a line. It has no effect.

Syntax #

13.5 Line Control

The **#line** directive tells the preprocessor to change the compiler's internally stored line number and filename to a given line number and filename. The compiler uses the line number and filename to refer to errors that it finds during compilation. The line number usually refers to the current input line, and the filename refers to the current input file. The line number is incremented after each line is processed.

Syntax **#line** *constant* *"filename"*$_{opt}$

The *constant* is interpreted as a decimal integer. Macro replacement can be performed on the preprocessing tokens, but the result must evaluate to the correct syntax.

You can alter the source line number and filename by writing a **#line** directive. The translator uses the line number and filename to determine the values of the predefined macros **__FILE__** and **__LINE__**. For information on these predefined macros, see "Predefined Macros" on page 374.

If you change the line number and filename, the compiler ignores the previous values and continues processing with the new values. The **#line** directive is typically used by program generators to cause error messages to refer to the original source file instead of to the generated program.

The *constant* value in the **#line** directive can be any integer constant. The *filename* can be any combination of characters and must be enclosed in double quotation marks (" "). If *filename* is omitted, the previous filename remains unchanged.

The current line number and filename are always available through the predefined macros **__LINE__** and **__FILE__**. You can use the **__LINE__** and

__FILE__ identifiers to insert self-descriptive error messages into the program text. See "Predefined Macros" on page 374.

The __FILE__ macro expands to a string whose contents are the filename, surrounded by double quotation marks (" ").

```
#line 151 "copy.cpp"
```

In this statement, the internally stored line number is set to 151 and the filename is changed to `copy.cpp`.

```
#define ASSERT(cond) \
((cond) ? (void)0 : \
          ((void)(cerr << "assertion failure \"" << #cond << \
                          "\" line " << __LINE__ << \
                          " file (" << __FILE__ << ")\n")))
```

In this example, the macro `ASSERT` uses the predefined identifiers __LINE__ and __FILE__ to print an error message about the source file if a given "assertion" is not true.

13.6 Error Directives

Error directives produce compiler-time error messages.

Syntax

#error *token-string*

The error messages include the argument *token-string* and are subject to macro expansion. These directives are most useful for detecting programmer inconsistencies and violation of constraints during preprocessing. The following example demonstrates error processing during preprocessing:

```
#if !defined(__cplusplus)
#error C++ compiler required.
#endif
```

When **#error** directives are encountered, compilation terminates.

13.7 Pragma Directives

Each implementation of C++ supports some features unique to its host machine. Some programs, for instance, need to exercise precise control over the memory areas where data is placed or to control the way certain functions receive parameters. The **#pragma** directives offer a way for each C++ compiler to offer machine-specific features while retaining overall compatibility with the C++ language.

Since pragmas are machine-specific by definition, they usually are different for every C++ compiler.

Important The pragmas discussed in this section apply to Microsoft C/C++ version 7.0.

Syntax

#pragma *token-string*

The *token-string* is a series of characters that gives a specific compiler instruction and arguments, if any. The number sign (#) must be the first nonwhite-space character on the line containing the pragma; white-space characters can separate the number sign and the word **pragma**. Following **#pragma**, write any text that the translator can parse as preprocessing tokens. The argument to **#pragma** is subject to macro expansion.

If the compiler finds a pragma that it does not recognize, it issues a warning, but compilation continues.

Pragmas can be used in conditional statements, to provide new preprocessor functionality, or to provide implementation-defined information to the compiler.

The **#pragma** directives instruct the compiler to implement the features specified by the argument. The Microsoft C++ compiler recognizes the following pragmas:

alloc_text	**function**	**loop_opt**	**pagesize**
auto_inline	**inline_depth**	**message**	**same_seg**
check_pointer	**inline_recursion**	**optimize**	**skip**
check_stack	**intrinsic**	**pack**	**subtitle**
comment	**linesize**	**page**	**title**

These pragma directives are summarized in the following list.

#pragma alloc_text(*textsegment*, *function1*, ...)
Names the segment where the specified routine definitions are to reside. This pragma takes effect at the first function defined after the pragma is seen.

#pragma auto_inline([[on | off]])
Inhibits the inline expansion of a function. The **auto_inline** pragma inhibits the preprocessor from expanding a function when the /Ob2 command-line option is in effect. To use it, place one pragma before and immediately after a function definition. This pragma takes effect at the first function definition after the pragma is seen.

#pragma check_pointer ([[{ on | off }]])
Instructs the compiler to turn off pointer checking if **off** is specified, or to turn on pointer checking if **on** is specified. If you do not specify either **on** or **off**, the effect of **check_pointer** is to reverse the current state of pointer checking. If pointer checking is on, it is turned off and vice versa.

This pragma takes effect at the first function defined after the pragma is seen.

#pragma check_stack ([[{ on | off }]])
Instructs the compiler to turn off stack probes if **off** is specified, or to turn on stack probes if **on** is specified. If no argument is given, stack probes are treated according to the default (**on**, unless /Gs was used). You can reduce the size of a program and speed up execution slightly by removing stack probes with either the /Gs option or the **check_stack** pragma.

This pragma takes effect at the first function defined after the pragma is seen.

#pragma code_seg ([["*segment-name*" [[, "*segment-class*"]])
The **code_seg** pragma specifies the default segment for functions. This is equivalent to using the /NT (name of TEXT segment) compilation option. It is also equivalent to specifying functions as based. You can, optionally, specify the class as well as the segment name. For example:

```
#pragma code_seg( "MY_CODE", "CODE" )
```

Specifying the **code_seg** pragma with no arguments causes the compiler to allocate all following functions in the default code segment.

The preceding example causes functions following to be allocated in a segment called MY_CODE. The segment is given a CODE class.

Note There is no way to specify the segment class using based function allocation or the /NT compilation option.

C++ functions allocated using the /NT option or **code_seg** pragma do not retain any information about their location. However, C++ functions allocated using the based function-allocation syntax retain that information in their external name.

#pragma comment(*comment-type* [[, *commentstring*]])
Allows you to place a comment record in an object file or executable file. The *comment-type* specifies the type of comment record. The optional *comment-string* is a string literal that provides additional information for some comment types. Because *comment-type* is a string literal, it obeys all the rules for string literals with respect to escape characters, embedded quotation marks ("), and concatenation.

#pragma data_seg ([["*segment-name*"[[, "*segment-class*"]])

The **data_seg** pragma specifies the default segment for data. This is equivalent to using the /ND (name of DATA segment) compilation option. It is also equivalent to specifying objects or pointers as based. You can, optionally, specify the class as well as the segment name. For example:

```
#pragma data_seg( "MY_DATA", "DATA" )
```

Specifying the **data_seg** pragma with no arguments causes the compiler to allocate all following data in the default data segment.

The preceding example causes functions following to be allocated in a segment called MY_DATA. The segment is given a DATA class.

Note There is no way to specify the segment class using based function allocation or the /ND compilation option.

C++ data allocated using the /ND option or **data_seg** pragma do not retain any information about their location. However, C++ data allocated using the based syntax retain that information in their external name.

#pragma function(*function1* [[**,** *function2*, ...]] **)**

Specifies that calls to the specified functions will be normal. The intrinsic pragma affects a specified function beginning where the pragma appears. The effect continues to the end of the source file or to the appearance of a function pragma specifying that function.

This pragma takes effect at the first function defined after the pragma is seen.

#pragma hdrstop [[("*filename*")]]

The **hdrstop** pragma controls the way precompiled headers work. The *filename* is the name of the precompiled header file to use or create (depending on compilation options). If *filename* does not contain a path specification, the precompiled header file is assumed to be in the same directory as the source file. See Chapter 2 in the *Programming Techniques* manual for more information about precompiled header files.

#pragma init_seg ({ **compiler** | **lib** | **user** | "*seg-name*" })

The **init_seg** pragma specifies a keyword or segment that affects the order in which startup code is executed. Because initialization of global static objects can involve executing code, you must specify a keyword that defines when the objects are to be constructed. It is particularly important to use the **init_seg** pragma in DLLs or libraries requiring initialization.

The options to the **init_seg** pragma are:

Option	Meaning
compiler	Reserved for Microsoft C run-time library initialization. Objects in this group are constructed first.
lib	Available for third-party class-library vendors' initializations. Objects in this group are constructed after those marked as **compiler** but before any others.
user	Available to any user. Objects in this group are constructed last.
seg-name	Allows explicit specification of the initialization segment. Objects in a user-specified *seg-name* are not implicitly constructed; however, their addresses are placed in the segment named by *seg-name*.

If you need to defer initialization (for example, in a DLL), you may choose to specify the segment name explicitly. You must then call the constructors for each static object. For an example of how these initializations are done, see the file CRT0DAT.ASM in the STARTUP\DOS directory.

#pragma inline_depth([[0... 255]])

Controls the number of times inline expansion can occur by controlling the number of times that a series of function calls can be expanded (from 0 to 255 times). Use this pragma to control functions marked as **inline** and **__inline**, or functions that the compiler automatically expands under the /Ob2 option. Requires an /Ob command-line option setting of either 1 or 2.

This pragma takes effect at the first function defined after the pragma is seen. Its default setting is 8. Once the depth set by this pragma is exceeded, calls are made.

#pragma inline_recursion([[on | off]])

Controls the inline expansion of direct or mutually recursive function calls. Use this pragma to control functions marked as **inline** and **__inline**, or functions that the compiler automatically expands under the /Ob2 option. Requires an /Ob command-line option setting of either 1 or 2. The default state for **inline_recursion** is off.

This pragma takes effect at the first function defined after the pragma is seen.

#pragma intrinsic(*function1* [[**,** *function2*, ...]])

Specifies that calls to the specified functions are intrinsic. Alternatively, you can use the /Oi option to make intrinsic the default for functions that have intrinsic forms. In this case, you can use the **function** pragma to override /Oi for specified functions.

This pragma takes effect at the first function defined after the pragma is seen.

The following functions have intrinsic forms:

abs	**_inpw**	**memcpy**	**_rotr**	**strlen**
_alloca	**labs**	**memset**	**setjmp**	**_strset**
_disable	**_lrotl**	**_outp**	**strcat**	
_enable	**_lrotr**	**_outpw**	**strcmp**	
_inp	**memcmp**	**_rotl**	**strcpy**	

#pragma linesize([[*characters*]] **)**
Specifies the number of characters per line in the source listing. The optional parameter *characters* is an integer constant in the range 79–132. If *characters* is absent, the compiler uses the value specified in the /Sl option or, if that option is absent, the default value of 79 characters per line.

#pragma loop_opt([[{off | on}]] **)**
The **loop_opt** pragma is obsolete. Use the **l** option of the optimize pragma in its place.

#pragma message(*messagestring* **)**
Sends a string literal to the standard output without terminating the compilation. The *messagestring* parameter can be a macro that expands to a string literal. You can concatenate such macros with string literals in any combination.

#pragma native_caller ([[{ on | off }]] **)**
The **native_caller** pragma controls the removal of native-code entry points from within source code. If you have p-code functions that are called only by other p-code functions, you can omit those entry points and save those bytes by using the /Gn compilation option or, on a function-by-function basis, using this pragma. See Chapter 3 in the *Programming Techniques* manual for more information about p-code.

#pragma optimize("[[*optimization-option-list*]]", {off | on} **)**
Specifies optimizations to be performed. Must appear outside a function. The optimization option list may be zero or more of the following: a, c, e, g, l, n, p, q, t, and w. These letters correspond to the /O compilation options.

This pragma takes effect at the first function defined after the pragma is seen.

#pragma pack([[{1 | 2 | 4 }]] **)**
Specifies packing alignment for structure types. You can use the /Zp option to specify the same packing for all structures in a module.

This pragma takes effect at the first function defined after the pragma is seen.

#pragma page([[*pages*]])
Skips the specified number of pages in the source listing. The **page** pragma generates a formfeed (page eject) in the source listing (created with /Fs) at the place where the pragma appears.

#pragma pagesize([[*lines*]])
Sets the number of lines per page in the source listing. The optional *lines* parameter is an integer constant in the range 15–255 that specifies the number of lines that you want each page of the source listing to have. If *lines* is absent, the pragma sets the page size to the number of lines specified in the /Sp option or, if that option is absent, to a default value of 63 lines.

#pragma same_seg(*variable1*, ...)
Tells the compiler to assume that the specified external variables are allocated in the same data segment. You are responsible for making sure that these variables are put in the same data segment. One way to do this is to specify the /ND option when you compile the program. Variables specified in a **same_seg** pragma must be explicitly declared with **extern** storage class, and they must either be explicitly declared with the **__far** keyword or assumed to be far because of the memory model used (compact, large, or huge).

This pragma takes effect at the first function defined after the pragma is seen.

#pragma skip([[*lines*]])
Skips the specified number of lines in the source listing. The skip pragma generates a newline character (carriage return–linefeed) in the source listing at the point where the pragma appears. The optional *lines* parameter is an integer constant in the range 1–127 that specifies the number of lines to skip. If this parameter is absent, the skip pragma defaults to one line.

#pragma subtitle("*subtitlename*")
Specifies a subtitle for the source listing. The *titlename* parameter can be a macro that expands to a string literal. You can concatenate such macros with string literals in any combination.

#pragma title("*titlename*")
Specifies a title for the source listing. The title appears in the upper-left corner of each page of the listing. The *titlename* parameter can be a macro that expands to a string literal. You can concatenate such macros with string literals in any combination.

#pragma warning(*warning-specifier* : *warning-number-list* [[,*warning-specifier* : *warning-number-list*...]])
The **warning** pragma allows selective modification of the behavior of compiler warning messages.

The *warning-specifier* can be: **once**, **default**, **1**, **2**, **3**, **4**, **disable**, or **error**. These work as follows:

warning-specifier	Meaning
once	Cause the message(s) to be displayed only once.
default	Cause the compiler's default behavior to apply to the message(s).
1, **2**, **3**, **4**	Force the warning message(s) to have the specified warning level.
disable	Cause the compiler not to issue the warning message(s).
error	Cause the compiler to report the warnings as error.

The *warning-number-list* can contain any warning numbers in the ranges 1 through 699, and 4001 through 4699. Multiple options can be specified in the same pragma directive as follows:

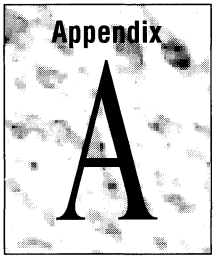
```
#pragma warning( disable : 4507 34; once : 4385; error : 164 )
```

This is functionally equivalent to:

```
#pragma warning( disable : 4138 34 )     // Disable warning messages
                                         //  4507 and 34.
#pragma warning( once : 4306 )           // Issue warning 4385
                                         //  only once.
#pragma warning( error : 164 )           // Report warning 164
                                         //  as an error.
```

Appendixes

Phases of Translation



A C++ program consists of one or more "source files," each of which contains some of the text of the program. A source file, together with its "include files" (files that are included using the **#include** preprocessor directive) but not including sections of code removed by conditional-compilation directives such as **#if**, is called a "translation unit."

Source files can be translated at different times—in fact, it is common to translate only out-of-date files. The translated translation units can be kept either in separate object files or in object-code libraries. These separate translation units are then linked to form an executable program (for example, a .EXE or .COM file).

Translation units can communicate using:

- Calls to functions that have external linkage.
- Calls to class member functions that have external linkage.
- Direct modification of objects that have external linkage.
- Direct modification of files.
- Interprocess communication (for Microsoft Windows applications only).

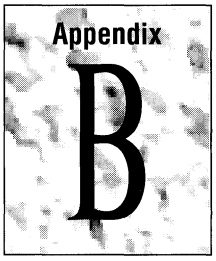
The following translation phases are not strictly required, but every implementation of C++, including Microsoft C++, must behave "as if" these rules were followed. (The actual order of translation is not important.)

1. Character mapping. Characters in the source file are mapped to the internal source representation. Trigraph sequences are converted to single-character internal representation in this phase.
2. Line splicing. All lines ending in a backslash (\) and immediately followed by a newline character are joined with the next line in the source file, forming logical lines from the physical lines. Unless it is empty, a source file must end in a newline character that is not preceded by a backslash.
3. Tokenization. The source file is broken into preprocessing tokens and white-space characters. Comments in the source file are replaced with one space character each. Newline characters are retained.

4. Preprocessing. Preprocessing directives are executed and macros are expanded into the source file. Use **#include** statements to invoke translation steps 1 through 4 on included text.
5. Character-set mapping. All source-character-set members and escape sequences are converted to their equivalents in the execution-character set. For Microsoft C++, both the source and the execution character sets are ASCII.
6. String concatenation. All adjacent string and wide-string literals are concatenated. For example, `"String " "concatenation"` becomes `"String concatenation"`.
7. Translation. All tokens are analyzed syntactically and semantically; these tokens are converted into object code.
8. Linkage. All external references are resolved to create an executable program.

The compiler issues warnings or errors during phases of translation in which it encounters syntax errors.

Microsoft-Specific Modifiers



Many of the Microsoft-specific keywords can be used to modify declarators to form derived types. These keywords are shown in Table B.1. (For more information about declarators, see Chapter 7.)

Table B.1 Microsoft-Specific Keywords

Keyword	Meaning	Used to Form Derived Types?
__asm	Insert the following assembly-language code.	No
__based	The name that follows declares a 16-bit offset to the base contained in the declaration.[1]	Yes
__cdecl	The name that follows uses the C naming and calling conventions.	Yes
__emit	Emit the following byte exactly (only legal inside __asm blocks).[2]	No
__export	The name that follows is marked with the EXPORT attribute.	Yes
__far	The name that follows declares an object or function that uses segmented addressing.[2]	Yes
__fastcall	The name that follows declares a function that uses registers, when available, instead of the stack for argument passing.	Yes
__fortran	The name that follows uses the FORTRAN/Pascal naming and calling conventions.[2]	Yes
__huge	The name that follows declares an object or function that uses segmented addressing.[2]	Yes
__interrupt	The name that follows declares a function that is an interrupt service routine.	Yes

Table B.1 *(continued)*

Keyword	Meaning	Used to Form Derived Types?
__loadds	The name that follows declares a function that must load the DS register as part of the entry sequence.[2]	Yes
__near	The name that follows declares a name that represents a 16-bit offset into DGROUP.[4]	Yes
__pascal	The name that follows uses the FORTRAN/Pascal naming and calling conventions.[2]	Yes
__saveregs	The name that follows declares a function. The function entry sequence saves the values in all registers.[2]	Yes
__segment	The name that follows specifies a segment value—for use with based pointers and objects.	Yes
__segname	Built-in conversion function that takes the name of a segment and returns a value of type __segment—for use with based pointers and objects.	Yes
__self	Specifies the name of the segment in which a based pointer is stored—for use with based pointers and objects.	Yes
__stdcall	The name that follows specifies a function that observes the standard calling convention.[3]	Yes

[1] Only based-on-pointer form allowed in 32-bit target compilations. In such compilations, they represent a 32-bit offset to a 32-bit base.
[2] Illegal in 32-bit compilations.
[3] Legal only in 32-bit target compilations.
[4] The __**near** keyword is allowed in 32-bit target compilations, but it is ignored.

The following sections discuss the syntactic usage and semantic meaning of the keywords in Table B.1.

B.1 Memory-Model Modifiers

Microsoft C++ supports explicit modification of addressing of an object or function. This permits more flexibility in writing code for Intel 80*x*86 and 80*x*87 based

computers. You can specify a memory model at compile time; this explicit specification overrides the compile-time memory model. Classes can also have an "ambient" memory model—the memory model used if none is explicitly specified. Explicit specification overrides the ambient memory model as well.

Memory-Model Modifiers and Objects

Memory-model modifiers can be used when declaring objects to explicitly specify addressing. A memory-model modifier affects the token immediately to its right. The declaration of pointers provides a good example of how this works. Consider the following declarations:

```
char __far *lpchObject;                 // Declaration 1
char * __far pchlFarPointer;            // Declaration 2
char __far * __far lpchlFarObject;      // Declaration 3
```

In the first declaration, the keyword `__far` modifies the token `*`, making `lpchObject` a pointer in the default data segment to a "far" object—an object accessed with a 32-bit *segment:offset* address.

In the second declaration, the keyword `__far` modifies the `pchlFarPointer` token. This modification makes `pchlFarPointer` a pointer that is not necessarily in the default data segment that refers to an object of type **char** in the default data segment. It declares a far-allocated pointer to a near object.

In the third declaration, the keyword `__far` is used twice: once to modify the pointer and once to modify the object. Therefore, it declares a far-allocated pointer to a far object.

Memory-Model Modifiers and Nonmember Functions

The memory-model modifiers can also be used to declare functions. In that context, they specify whether the function call is a far (intersegment) or near (intrasegment) call. Consider the following declaration:

```
char __far * __far _fstrcpy( char __far *dst, char __far *src );
```

The preceding code declares a far function (`_fstrcpy`) that takes two arguments of type "far pointer to `char`" and returns type "far pointer to `char`." The declaration can be broken down as follows:

- Function returning.
- Accepting two arguments of type far pointer to `char`.
- Making a far (intersegment) call to the function named `_fstrcpy`.
- Returning a far pointer to type `char`.

Memory-Model Modifiers and Member Functions

Memory-model modifiers can be used with nonstatic class-member functions to specify:

- Calling of the function (intersegment or intrasegment).
- Addressing for the **this** pointer.

The following example declares a class that illustrates how these options work:

```
class String
{
public:
  char * __far GetStringPointer1();
  char * GetStringPointer2() __far;
};
```

In the preceding class declaration, `GetStringPointer1` is always called with an intersegment (far) call. The **this** pointer passed to `GetStringPointer1` is dependent on the ambient memory model (if any) and the compilation options. The function `GetStringPointer2` is invoked with the call implied by the compilation options. The **this** pointer passed to `GetStringPointer2` is always far.

The declaration syntax shown for `GetStringPointer2` uses the *cv-mod-list* part of the syntax (see "Overview" in Chapter 7, on page 183) and is illegal for nonmember functions or static member functions. Using this syntax, only the memory-model specifiers, **__near**, **__far**, and **__huge** can be used.

By using different modifiers for the **this** pointer, you can create functions that behave properly with respect to the address space in which their objects are located. This is particularly useful for any class that does memory management.

Memory-Model Modifiers and Classes

Memory-model modifiers can also be used in declaration of an ambient memory model for classes. This "ambient memory model" overrides the addressing specified in compilation options for a given class.

Syntax

class-head:
class-key $rmodel_{opt}$ $identifier_{opt}$ $base\text{-}spec_{opt}$
class-key $rmodel_{opt}$ $class\text{-}name_{opt}$ $base\text{-}spec_{opt}$

Consider this example:

```
class __far CollectionOfCustomers
{
public:
       CollectionOfCustomers();
  int    CCAdd( Customer *CNewCustomer );
  Customer& CCFindName( char *Name );
};
```

The preceding example declares a class called `CollectionOfCustomers`. No matter what memory model is specified in the compilation options, objects of type `CollectionOfCustomers`, along with **this** pointers for member functions, are far unless specifically declared as near. Consider the following example:

```
CollectionOfCustomers *pBestCustomers;
```

The preceding example declares a pointer to an object whose data is addressed far (not necessarily in the default data segment). The **this** pointer, which is used for the member functions `CollectionOfCustomers` (the constructor), `CCAdd`, and `CCFindName`, is a far pointer.

The ambient memory model can be overridden in explicit declarations as follows:

```
CollectionOfCustomers __near *npBestCustomers;
```

In the preceding declaration, `npBestCustomers` stores its member data in the default data segment (near) even though the ambient memory model calls for far addressing. The **this** pointer, however, is still far, as specified by the ambient memory model.

Warning If a class has an ambient memory model that is specifically overridden in an object declaration, only methods with appropriate addressing are guaranteed to work.

Memory-Model Specifiers and Overloading

The Microsoft memory-model specifiers for function arguments and for the **this** pointer are used for disambiguation between overloaded functions. The following two functions are considered different because the arguments use different addressing:

```
char *strchr( char *szString, char chTarget );
char *strchr( char __far *szString, char chTarget );
```

This functionality enables different implementations, depending on the addressing specified. Similarly, the member functions in the following class fragment are different because the **this** pointer is overloaded:

```
class __far List
{
public:
  List() __far;   // This class is intended to be far.
  List() __huge;  // The near and huge constructors simply
  List() __near;  //  issue run-time error messages.
...
protected:
  void AddressingError( char *szModel );
};

List::List() __far // Correct instantiation.
{
  // Set up the list.
}

List::List() __near // Incorrect addressing.
{
  AddressingError( "__near" );
}

List::List() __huge // Incorrect addressing.
{
  AddressingError( "__huge" );
}

// Issue error message.
inline void AddressingError( char *szModel )
{
  cerr << "Cannot create a << szModel
     << " instance of a List.\n"
}
```

In the preceding example, the functions `List::List() __far`, `List::List() __near`, and `List::List() __huge`, are considered different because of the overloaded **this** pointer.

__near

The **__near** keyword specifies that a name is to have 16-bit addressing. For functions, this 16-bit offset is assumed to be from the default code segment, **_TEXT**; for objects, it is assumed to be from the default data segment, **_DATA**. Functions declared as **__near** can be allocated in other segments by declaring them as **__based** or using the /NT compilation option.

Class-type objects declared as near have:

- Near addressing for member data.
- Near **this** pointer.
- Function calling determined by the compilation options.

For information about converting near addresses to far and huge, see "Pointer Conversions" in Chapter 3, on page 71.

32-Bit Specific

The **__near** keyword is ignored in 32-bit compilations; however, no error or warning message is generated.♦

__far

The **__far** keyword specifies that a name is to have 32-bit addressing. For functions, this 32-bit address contains the segment and offset of the called function; for objects, it contains the segment and offset of the object.

Functions in the same compilation unit reside in the same segment unless the **alloc_text**, **code_seg**, or **same_seg** pragmas are used, or unless the function is declared as based. The default naming convention for functions specified as **__far** is *source-filename*_**TEXT**. The default naming convention for objects specified as **__far** is *source-filename*_**DATA**.

Class-type objects declared as far have:

- Far addressing for member data.
- Far **this** pointer.
- Function calling determined by the compilation options.
- Limit on object or array size; the practical limit on the size of a single far object is 64K– *data-threshold-size*. The *data-threshold-size* defaults to 32,767, but it can be set using the /Gt compilation option.

Objects declared as **__far** must reside within the segment in which they start. Therefore, they must be smaller than 64K. This restriction allows pointer arithmetic to be done on 16-bit values while still retaining the ability to address data in multiple segments.

The **alloc_text** pragma affects where segment functions declared as **__far** reside; the **same_seg** pragma affects assumptions the compiler makes regarding where segment functions and objects declared as **__far** reside. For more information about the **alloc_text** and **same_seg** pragmas, see "Pragma Directives" in Chapter 13, on page 385.

32-Bit Specific

The **__far** keyword is not supported in 32-bit compilations.♦

__huge

The **__huge** keyword specifies that a name is to have 32-bit addressing. For objects, this 32-bit address contains the segment and offset of the object; for functions, it is illegal.

The default naming convention for objects specified as **__huge** is *source-filename*_**DATA**. (This is identical to the naming convention described for objects declared as **__far**.)

Class-type objects declared as **__huge** have:

- Far addressing for member data.
- Far **this** pointer.
- Function calling determined by the compilation options.
- Relaxed limit on object and array size. Objects and arrays can be greater than 64K in size; however, if an array exceeds 128K bytes in size, the individual elements must be of a length that is a power of two.

Objects declared as **__huge** trade efficiency in pointer arithmetic for relaxed limits on array and object size.

Declaring an object of automatic storage class as **__huge** generates an error. Only static arrays and memory allocated using the **new** operator can be declared as huge, as in the following example:

```
struct Customer
{
  char szName[40];   // An object of type Customer
  char szAddr1[30];  //  requires 149 bytes of
  char szAddr2[30];  //  memory.
  char szCity[30];
  char szZip[9];
  char szPhone[10];
};

int main()
{
  // Allocate a static array that occupies 447,000 bytes.
  static Customer __huge GiantArray[3000];

  // Allocate a dynamic array that occupies 447,000 bytes.
  Customer __huge *pCustList = new Customer[3000];

  Customer __huge GiantAutoArray[3000]; // Error. Automatic.
}
```

Note Some, but not all, of the run-time library functions support huge types. Check the *Run-Time Library Reference* manual to see if a particular function will work with objects declared as **__huge**.

32-Bit Specific

The **__huge** keyword is not supported in 32-bit compilations.♦

__based

Based addressing is useful when you need to:

- Exercise precise control over the segment in which objects are allocated (static and dynamic based data).
- Reference far objects using a 16- rather than a 32-bit address. This can decrease the size of an executable file while increasing its speed.
- Store pointers to memory not allocated by your program—for example, ROM data.

32-Bit Specific

The only form of based addressing acceptable in 32-bit compilations is "based on a pointer" which defines a type that contains a 32-bit displacement to a 32-bit base.

Syntax

based-range-modifier:
__based (*base-expression*)

base-expression:
base-constant
based-variable
based-abstract-declarator
__self

base-constant:
segment-name
segment-cast

based-variable:
identifier

based-abstract-declarator:
abstract-declarator

base-type:
type-name

segment-name:
__segment (string-literal)

segment-cast:
(__segment) pointer-id
(__segment) & identifier

The keywords and operators in the following list have been added to the language to support based addressing:

Keyword or Operator	Use
__segment	A type that contains a segment value.
__segname	A built-in function that returns type __segment. The **__segname** function can be used in declarations to initialize variables of type **__segment**, or it can be used in declarations of based objects or pointers.
__self	The result of the **__self** keyword is a far pointer to the segment where a pointer is stored. Basing a pointer on **__self** avoids the segment-register reloads implicit in storing the pointer and object in different segments.
:>	Base operator. The base operator combines a segment and an address that can be dereferenced using the * operator.

Based Pointers

Based pointers are short (16-bit) offsets from a segment base. The effective address is calculated using the formula:

effective address = base + pointer value

32-Bit Specific

In 32-bit target compilations, based pointers are 32-bit offsets from a 32-bit base. ♦

When dereferencing a based pointer, the base must either be explicitly specified or implicitly known through the declaration. The following code shows some example declarations of based pointers:

```
// Binary tree
struct BTree
{
  char *szSymbolName;
  BTree *btLeft;
  BTree *btRight;
};

// Declare a pointer to data that resides in segment SYM_DATA.
BTree __based( __segname( "SYM_DATA" ) ) *btSymTable1;

// Declare a pointer to data that resides in segment 0x7000.
__segment SegVar = 0x7000;
BTree __based( SegVar ) *btSymTable2;

// Declare a pointer to data that resides in the same segment
//  as the pointer btSymTable3.
BTree __based( (__segment)__self ) *btSymTable3;
```

A based pointer can be based on the following:

- A constant. A pointer can be based on an expression that evaluates to a constant. These constants can be supplied as segment names using the **__segname** operator.
- A variable segment. A pointer based on a variable segment is specified using an expression that evaluates type to **__segment** but that does not evaluate to a constant.
- Self. A pointer based on self is specified using the **__self** function. It causes the pointer to refer only to objects in its own segment.
- Void. A pointer based on **void** has no implicit base; the base must be supplied at the point of dereference.
- A pointer. A pointer based on a pointer allows the base to be manipulated at run time.

Pointers Based on a Constant

Pointers based on a constant are specified in several ways:

In the following example, `sgConst` is explicitly declared as a constant. The pointer `bp1`, therefore, is based on a constant.

```
const __segment sgConst = 0x3000;
char __based( sgConst ) *bp1;
```

In the following example, the **__segname** function returns a constant value of type **__segment**. The pointer `bp3`, therefore, is based on a constant.

```
char __based( __segname("INFO_STRINGS") ) *bp3;
```

Pointers Based on a Segment Variable

Pointers based on a segment variable require that a variable of type **__segment** be declared. This variable determines what segment the based pointer refers to and can be changed at run time. The following uses a pointer, `pbv`, based on a segment variable, `sgVar`:

```
#include <iostream.h>

__segment sgVar;

void __based( sgVar ) *pbv;

int main()
{
  static __segment sgList[] =  // Initialize a static array
  {                            //  with the CODE, DATA, and
    __segname( "_CODE" ),      //  STACK segment values.
    __segname( "_DATA" ),
    __segname( "_STACK" )
  };
  static char *aszSegNames[] = // Initialize an array of
  {                            //  strings with the names
    "_CODE",                   //  of the above segments.
    "_DATA",
    "_STACK"
  };

  // Calculate the size of the array.
  int cSegments = sizeof( sgList ) / sizeof ( __segment );

  // Dump the first 256 bytes of each segment.
  for( int i = 0; i < cSegments; ++i )
  {
    cout << "Segment: " << aszSegNames[i] << "\n\n";

    // Set the variable on which pbv is based to point to
    //  the ith segment in the list.
    sgVar = sgList[i];
```

```
    // Initialize pbv to point to the
    //  initial byte of the current
    //  segment (sgList[i]:0000).
    for( int row = 0, pbv = 0; row < 16; ++row )
    {
      for( int col = 0; col < 16; ++col, pbv++ )
        cout << hex << *(unsigned*)pbv << " ";
      for( col = 0, pbv -= 16; col < 16; ++col, pbv++ )
        cout << isprint(*(char*)pbv) ? *(char*)pbv : '.';
      cout << "\n";
    }
  }
  return 0;
}
```

Pointers Based on Self

Pointers based on self can access data anywhere in the segment in which the pointer resides. They are declared using the **__self** function cast to the **__segment** type as the *base-expression*. You can base a pointer only on **(__segment)__self**. You cannot base a pointer on **__self** alone. Basing a pointer on self can improve program performance by requiring that the segment register be the same for addressing both the pointer and the data it addresses.

Note While both the pointer and the data are in the same segment, some segment-register reloads can be forced by such operations as conversions, references to other data, and function calls.

Pointers based on self commonly refer to objects allocated using the **new** operator. For example:

```
// Declare the pointer; the actual segment is unimportant.
char __based( (__segment)__self ) *pbch;

// Allocate storage for objects.
pbch = new char __based( (__segment)__self )[1024];
```

Pointers based on self are particularly useful for optimizing access in self-referential data structures such as linked lists and trees.

Functions cannot return pointers based on self. Also, register variables cannot be based on self. Only addressable l-values can be converted to pointers based on self.

Since only l-values can be converted to pointer types based on self, you cannot assign a constant to a pointer based on self. Consider the following code:

```
int __based((__segment)__self) *piself;
piself = 1;
```

Before performing the assignment, the compiler attempts to convert 1, which is of type **const int**, to type **int __based(void) ***, which is an illegal conversion. To put the value 1 in the pointer `piself`, you first must cast 1 to a pointer based on void, as follows:

```
piself = (int __based(void)*)1;
```

32-Bit Specific

Pointers based on self are not supported in 32-bit target compilations. ♦

Pointers Based on void

Pointers based on **void** defer address calculation until the pointer is dereferenced. Unlike most other forms of based pointers, a pointer that is based on **void** has no implied segment as its base.

The segment specified at the point of dereference can be a constant or a segment variable. It is combined with the offset using the base operator (**:>**) to form an address that can be dereferenced using the indirection (*) operator.

The following example shows how to declare and dereference a pointer based on **void**.

```
struct BiosEquipList  // Structure for the BIOS Equipment List
{                     //  that starts at 0000:0410 (hex).

...

};

// Declare ROM data as const and supply the offset, hex 410.
const BiosEquipList __based( void ) *bpelROM = 0x410;

int main()
{
  BiosEquipList elLocal;  // Local copy of equipment list.

  // Make a local "shadow" copy of the BIOS equipment list.
  elLocal = *((__segment)0x0000 :> bpelROM);

  return 0;
}
```

Pointers Based on Pointers

The "based on pointer" variant of based addressing enables specification of a pointer as a *base-expression.* The based pointer, then, is an offset into the segment starting at the beginning of the pointer on which it is based.

One use for pointers based on pointers is for persistent objects that contain pointers. A linked list that consists of pointers based on pointers can be saved to disk and reloaded to another place in memory, and the pointers are still valid. The following example declares such a linked list:

```
void *vpBuffer;

struct llist_t
{
  void __based( vpBuffer ) *vpData;
  llist_t __based( vpBuffer ) *llNext;
};
```

The pointer, `vpBuffer`, is assigned the address of memory allocated at some later point in the program; the linked list is then relocated relative to the value of `vpBuffer`.

32-Bit Specific

Pointers based on pointers are the only form of **__based** valid in 32-bit compilations. In such compilations, they are 32-bit displacements from a 32-bit base. ♦

Based Objects

Static and external objects can be declared using the **__based** keyword. In this context, the **__based** specification causes the object to be allocated in the specified segment. The following example shows how to declare a based object:

```
Class Symbol
{
...
};

Symbol __based( __segname( "SYM_DATA" ) ) Symbol;
```

The preceding declaration allocates one object of type `Symbol` in segment `SYM_DATA`. Similarly, arrays of objects can be declared as based:

```
Symbol __based( __segname( "SYM_DATA" ) ) Symbol[3000];
```

Based Functions

If a function is to be allocated in a given segment, it can be declared as based on a segment constant. No other forms of based declarations are accepted for functions. Consider the following example:

```
char __based(__segname("STRING_TEXT"))
  __near *StringCompare( char __near *String1, char __near *String2 );
char __based(__segname("STRING_TEXT"))
  __far *FStringCompare( char __far *String1, char __far *String2 );
```

In the preceding declaration, both `StringCompare` and `FStringCompare` are allocated in the segment `STRING_TEXT`. However, the segment in which the functions are allocated does not affect the calling protocol specified by the `__near` and `__far` keywords. In the preceding example, `StringCompare` can be called only by other functions in the same segment, and it is accessed using a near (intrasegment) call. The `FStringCompare` function, however, is declared as `__far`, and can be accessed from any function in the program using a far call.

Note Declaring functions as **__based** replaces the **alloc_text** pragma. However, the **alloc_text** pragma is retained for backward compatibility.

This application of **__based** is not used for disambiguation of overloaded functions. Therefore, the following two function declarations conflict:

```
char __based(__segname("FAR_TEXT")) Print( int iValue );
char __based(__segname("NEAR_TEXT")) Print( int iValue );
```

The compiler recognizes the suffix `_TEXT` as a code segment designation. Therefore, any function based on a segment that has the `_TEXT` suffix is placed in a code segment. If you choose a segment name that does not have the `_TEXT` suffix, however, the segment chosen is a data segment.

__segment Keyword

The **__segment** type is used either at the point of declaration of a based type or at the point of a based dereference to specify the segment in which the based object resides. If specified at the point of declaration, the segment value is implicit and need not be respecified at the point of dereference (based pointers). Otherwise, the segment must be explicitly specified.

The following example shows specification of a segment in a declaration and at the point of dereference:

```
// Dump the first 256 bytes of the data segment (DS)
//  using based pointers.

#include <iostream.h>
```

```
int main()
{
  // Segment specified at point of declaration.
  unsigned char __based( __segname( "_DATA" ) ) *puc = 0;

  // Segment specification deferred until point of dereference.
  // pv is a generic pointer to type unsigned char.
  unsigned char __based( void ) *pv = 0;

  int i, j;

  for( i = 0; i < 16; ++i )
  {
    // Specify the segment at point of dereference.
    for( j = 0; j < 16; ++j )
      cout << hex << setw( 2 )
         << (unsigned)*(__segname("_DATA"):>pv++) << " ";

    // Because the segment for puc was specified at the point
    // of declaration, it can be used like a normal pointer.
    for( j = 0, pv -= 16; j < 16; ++j )
      cout << *puc++;

    cout << "\n";
  }

  return 0;
}
```

Type **__segment** behaves like any other C++ type—derivative types can be formed using the pointer-to, array-of, and function-returning operators (*, [ ], and ()), respectively).

An object of type **__segment** can be initialized with the following four items:

- An expression that evaluates to an integral constant value.
- The result of the built-in function **__segname**. This allows specification of a segment by name, causing the linker to insert segment fixups in the executable file, to be resolved at load time.
- Another expression of type **__segment**. For example:

```
__segment sgCustomerData = 0x7000;
__segment sgCurrent = sgCustomerData;
```

- Another expression explicitly cast to type **__segment**. The following list shows how various expressions are converted in explicit casts to type **__segment**:

Expression	Base
Cast from pointer not explicitly declared with a memory-model modifier	Segment value for that pointer
Cast from near pointer	Segment value of the default data segment
Cast from far pointer	Segment portion of the far pointer
Cast from based pointer	Base segment of based pointer
Cast from address value obtained using the address-of operator (**&**) on an object	Segment in which the object is located

32-Bit Specific

The **__segment** type is not supported in 32-bit compilations. ♦

__segname Function

The built-in function **__segname** accepts a quoted string and returns a value of type **__segment**. A **__segname** declaration is not an l-value and its address cannot be taken.

The primary use for the **__segname** function is in initializing objects of type **__segment** or in declarations of pointers or objects of based type. It enables specification of a segment name instead of a segment value when declaring a based type. Consider the following example:

```
// Keep all message strings for the File.Open dialog box
//  in the same segment.
char __based(__segname("ERR_STRINGS")) *szErrStr[] =
{
  "Cannot find file: %s",
  "Cannot open file: %s",
  "Invalid path: %s",
  "Drive %c: does not exist or is not ready",
  ...
};
```

Collecting like data into the same segment is beneficial in Windows programming because users are commonly performing a task—such as opening a file—that requires only a certain subset of the application's data. Microsoft Windows manages segment swapping; however, the way data is organized in those segments can greatly affect how often these disk-intensive swaps occur.

The following list shows the predefined segment-name constants you can use as arguments to the **__segname** function:

Name	Refers To
_CODE	The default code segment
_DATA	The default data segment
_CONST	The default CONST segment
_STACK	The stack segment

__self Function

Pointers can be declared as based on **__self**. This form of declaration defers calculation of the pointer value until the point of dereference. It also ensures that the object the pointer references is in the same segment as the pointer itself. For example:

```
char __based( (__segment)__self ) *pszMessage;

...

pszMessage = new char __based( __self )[1024];
if( pszMessage == NULL )
  cerr << "Cannot allocate 1K buffer.\n";
else
  // 1K buffer allocated in same segment as pointer.
```

In the preceding example, the pointer is initialized using the **new** operator.

B.2 Calling and Naming Convention Modifiers

The modifiers described in this section affect the function "calling conventions" and "naming conventions." Calling conventions determine how functions are called; naming conventions determine how external names are treated. Calling and naming conventions are composed of the following elements:

Element	Description
Argument-passing order	Some conventions pass arguments to functions from right to left; others from left to right.
Argument-passing convention	In some calling conventions, the default argument-passing convention is by value; in others, it is by reference.
Stack-maintenance responsibility	Because arguments are pushed on the stack for communication with the function, either the called function or the caller must adjust the stack.

Name-decoration convention	Some calling conventions add characters to the name as it is declared in the program. This is called "name decoration."
Case-translation convention	Some calling conventions require case translation to uppercase.
Return-value convention	Some calling conventions return values in registers; others return values on the stack. Most return values both ways, depending on the size of the value to be returned.

__cdecl

The **__cdecl** convention is the convention used by the compiler. The following list shows the implementation of this calling convention:

Element	Implementation
Argument-passing order	Right to left
Argument-passing convention	By value, unless a pointer or reference type is passed
Stack-maintenance responsibility	Calling function adjusts the stack
Name-decoration convention	Underscore character (_) is prefixed to names
Case-translation convention	No case translation performed

To declare a function as **__cdecl**, use a declaration of the form:

```
char * __cdecl stricmp( char *szString1,
            char *szString2 );
```

Note that, in the preceding declaration, the modifier `__cdecl` modifies the name immediately to its right, `stricmp`.

The following list shows how values are returned from functions specified as **__cdecl**:

Type	Return Location
char, **unsigned char**	Returned in AL register.
int, **unsigned**, **short**, **unsigned short**, "pointer to **near**"	Returned in AX register.
long, **unsigned long**	Returned in DX:AX registers.
float, **double**	Copied to the global variable __fac; returns a pointer to **__fac** in AX or DX:AX, depending on addressing model.

long double	Placed on the NDP (numeric data processor) stack using the FLD instruction.
Structures	Depends on the size of the structure: 1 byte: returned in AL register 2 bytes: returned in AX register 4 bytes: returned in DX:AX registers For structures larger than 4 bytes, the caller passes a pointer to a hidden variable that will receive the return value as the last item pushed. The structure being returned is copied into the location pointed to by the hidden variable. The pointer to the hidden variable is then returned in AX or DX:AX (for near or far addressing models, respectively).

In 16-bit compilations, the variable-argument facility of ANSI C can be used only with functions of type **__cdecl**. In 32-bit compilations, variable-argument lists can also be used with the **__stdcall** calling convention, discussed on page 419.

__fastcall

The **__fastcall** convention specifies that arguments to functions are to be passed in registers, when possible. The following list shows the implementation of this calling convention:

Element	Implementation
Argument-passing order	Left to right
Argument-passing convention	By value, unless a pointer or reference type is passed
Stack-maintenance responsibility	Called function adjusts the stack
Name-decoration convention	At sign (@) is prefixed to names
Case-translation convention	No case translation performed
Return-value conventions	Identical to **__cdecl**

To declare a function as `__fastcall`, use a declaration of the form:

```
char * __fastcall parsechar( char *szString, char chTarget );
```

Note that, in the preceding declaration, the modifier `__fastcall` modifies the name immediately to its right, `parsechar`.

The compiler allocates registers for the arguments according to the type of the argument and availability of an appropriate register. If no appropriate register is available, the argument is pushed on the stack, much as in the **__pascal** calling convention. The following list shows the register candidates for various types:

Type	Register Candidate
char, unsigned char	AL, DL, BL
int, unsigned int	AX, DX, BX
long, unsigned long	DX:AX
"pointer to **near**"	BX, AX, DX
"pointer to **far**," "pointer to **huge**"	Passed on the stack

All far and huge pointers, structures, unions, and floating types are passed on the stack.

__fortran/__pascal

The **__fortran** and **__pascal** conventions specify argument passing that is compatible with Microsoft FORTRAN and Microsoft Pascal. The following list shows the implementation of this calling convention:

Element	Implementation
Argument-passing order	Left to right
Argument-passing convention	By value, unless a pointer or reference type is passed
Stack-maintenance responsibility	Called function adjusts the stack
Name-decoration convention	None
Case-translation convention	Names translated to uppercase

To declare a function as **__fortran** or **__pascal**, use a declaration of the form:

```
void __fortran regress( double series[],
            double *slope,
            double *intercept );
```

Note that, in the preceding declaration, the modifier `__fortran` modifies the name immediately to its right, `regress`. Functions declared using the `__fortran` or `__pascal` modifiers return values the same way as those declared using `__cdecl`.

Note The **__fortran** and **__pascal** modifiers are identical; the two keywords are supported for internal code documentation.

32-Bit Specific

The **__fortran** and **__pascal** keywords are not supported in 32-bit compilations.♦

__stdcall

The **__stdcall** calling convention is a faster, more type-safe version of **__cdecl**. The following list shows the implementation of this calling convention:

Element	Implementation
Argument-passing order	Right to left
Argument-passing convention	By value, unless a pointer or reference type is passed
Stack-maintenance responsibility	Called function adjusts the stack
Name-decoration convention	An underscore (_) is prefixed to the name. The name is followed by the at-sign (@) character, followed by the number of bytes in the argument list. Therefore, the function declared as `int func( int a, double b )` is decorated as follows: `_func@12`
Case-translation convention	None

Functions declared using the **__stdcall** modifier return values the same way as functions declared using **__cdecl**.

Note Functions with variable argument lists must be prototyped, or a linker error occurs.

B.3 Special Modifiers

The Microsoft-specific keywords, **__export**, **__interrupt**, and **__loadds**, do not necessarily modify the calling or naming convention of a function or object. They can, however, modify other behavior.

__export

The **__export** keyword specifies that the named function or object is to be marked for export from a DLL or to Windows. Applying the **__export** attribute to a function or object is the same as specifying the name in the **EXPORTS** section of a module-definition file.

It is not possible to use the **__export** keyword to specify any of the following additional information that can be provided in a module-definition file for exported functions or objects:

- Name aliasing (internal versus external)
- Tagging functions with ordinal values
- Keeping names resident in memory
- Tagging exported real-mode Windows functions as having no data

In most cases, the defaults provided by the **__export** keyword are sufficient, especially for call-back functions exported to Windows.

The **__export** keyword can be used with any of the calling-convention modifiers discussed in the previous section, but should be used only with functions declared as **__far**.

__interrupt

The **__interrupt** keyword indicates that the function is an interrupt handler. The compiler generates appropriate entry and exit sequences for the interrupt-handling function, including saving and restoring all registers and executing an IRET instruction to return.

When an interrupt function is entered, the **DS** register is initialized to the default (near) data segment. This allows access to global variables from within an interrupt function.

In addition, all registers (except **SS**) are saved on the stack. These registers can be accessed within the function by declaring a function-parameter list containing a formal parameter for each saved register. The following example illustrates such a declaration:

```
void __interrupt __far int_handler(
  unsigned _es, unsigned _ds,
  unsigned _di, unsigned _si,
  unsigned _bp, unsigned _sp,
  unsigned _bx, unsigned _dx,
  unsigned _cx, unsigned _ax,
  unsigned _ip, unsigned _cs,
  unsigned flags )
```

Restrictions on Interrupt Functions

An interrupt function must be far. Programs compiled with the small or compact memory model must explicitly declare interrupt functions with the **__far** keyword.

Interrupt functions must observe the **__cdecl** calling convention. Therefore, the only calling convention keyword that can be used in combination with **__interrupt** is **__cdecl**.

Functions declared with the **__interrupt** keyword cannot also be declared with the **__saveregs** keyword.

Considerations When Using __interrupt

When an interrupt function is called by an **INT** instruction, the interrupt enable flag is cleared. This means that no further interrupts (including keyboard, time-of-day, and other crucial interrupts) are processed until the interrupt function returns.

If an interrupt function needs to do significant processing, it should call the **_enable** function to reset the interrupt-enable flag.

Interrupt functions must obey special rules because they are potentially reentrant (that is, the function can be entered more than once before the first return is executed). When designing an interrupt-handling function, consider the following guidelines:

- If the function does not use the **_enable** function to set the interrupt flag, important interrupts may be ignored. This can result in such undesirable effects as lost keystrokes and inaccurate real-time clocks.
- If the function uses the **_enable** function to set the interrupt flag, another interrupt of the same sort may take place. To make sure that the interrupt handler takes this into account, serialize access to all objects that are not automatic or accessed on the free store using an automatic pointer.

__loadds

The **__loadds** keyword causes the compiler to generate a function-entry sequence to load the data-segment (**DS**) register with the segment value of the most recently specified data segment. The compiler also generates code to restore the previous **DS** value when the function terminates.

Loading the **DS** register is essential for Windows call-back functions and Windows DLL entry points. Because there is no requirement for Windows or clients of a DLL to "know" about the entry function's use of memory, these functions cannot depend on **DS** pointing to their data segment at entry unless they are declared as **__loadds** or compiled with the /Au compilation option.

Note The **__loadds** keyword does not imply any change in calling convention; it can be specified with any calling-convention modifier that is compatible with 16-bit compilations.

When loading the **DS** register, the compiler uses the segment specified by the /ND (name-data-segment) option, or, if no segment has been specified, the default group **DGROUP**. Note that this function modifier has the same effect as the /Au option, but on a function-by-function basis.

32-Bit Specific The **__loadds** keyword is not supported in 32-bit compilations. ♦

__saveregs

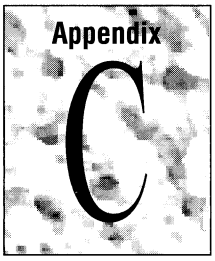
The **__saveregs** keyword causes the compiler to generate a function entry sequence that saves all CPU registers and an exit sequence that restores the registers. Note that **__saveregs** does not restore registers used for a return value (the **AX** register, or **AX** and **DX**).

The **__saveregs** keyword is useful when the register conventions of the caller are not known. For instance, **__saveregs** can be used for a general-purpose function in a dynamic-link library. Because such a function can be called from any language, it is unsafe to assume any particular calling convention. Functions that are intended to be interfaced with assembly-language code can protect the caller against undesirable side effects by using the **__saveregs** attribute.

It is illegal to declare a function with both the **__saveregs** and the **__interrupt** attributes.

32-Bit Specific The **__saveregs** keyword is not supported in 32-bit compilations. ♦

Grammar Summary



This appendix describes the formal grammar of the C++ language, as implemented in the Microsoft C/C++ version 7.0 compiler. It is loosely organized around the chapter organization of this book as follows:

- The "Keywords" section describes keywords covered in Chapter 1, "Lexical Conventions."
- The "Expressions" section describes the syntax of expressions described in Chapter 4, "Expressions."
- The "Declarations" section describes the syntax of declarations described in Chapter 6, "Declarations."
- The "Declarators" section describes the syntax of declarators covered in Chapter 7, "Declarators."
- The "Classes" section covers the syntax used in declaring classes as covered in Chapter 8, "Classes."
- The "Statements" section covers the syntax used in writing statements, as covered in Chapter 5, "Statements."
- The "Preprocessor" section covers the syntax of preprocessing directives and operators, covered in Chapter 13, "Preprocessing."
- The "Microsoft Extensions" section covers the syntax of features unique to Microsoft C++. Many of these features are covered in Appendix B, "Microsoft-Specific Modifiers."

C.1 Keywords

class-name:
  identifier

enum-name:
  identifier

typedef-name:
  identifier

C.2 Expressions

expression:
  assignment-expression
  expression , *assignment-expression*

assignment-expression:
  conditional-expression
  unary-expression assignment-operator assignment-expression

assignment-operator: one of
  = *= /= %= += -= >= <= &= ^= |=

conditional-expression:
  logical-or-expression
  logical-or-expression ? *expression* : *conditional-expression*

logical-or-expression:
  logical-and-expression
  logical-or-expression | | *logical-and-expression*

logical-and-expression:
  inclusive-or-expression
  logical-and-expression & & *inclusive-or-expression*

inclusive-or-expression:
  exclusive-or-expression
  inclusive-or-expression | *exclusive-or-expression*

exclusive-or-expression:
  and-expression
  exclusive-or-expression ^ *and-expression*

and-expression:
  equality-expression
  and-expression **&** *equality-expression*

equality-expression:
  relational-expression
  equality-expression **==** *relational-expression*
  equality-expression **!=** *relational-expression*

relational-expression:
  shift-expression
  relational-expression **<** *shift-expression*
  relational-expression **>** *shift-expression*
  relational-expression **<=** *shift-expression*
  relational-expression **=>** *shift-expression*

shift-expression:
  additive-expression
  shift-expression **<<** *additive-expression*
  shift-expression **>>** *additive-expression*

additive-expression:
  multiplicative-expression
  additive-expression **+** *multiplicative-expression*
  additive-expression **–** *multiplicative-expression*

multiplicative-expression:
  segment-expression
  multiplicative-expression ***** *segment-expression*
  multiplicative-expression **/** *segment-expression*
  multiplicative-expression **%** *segment-expression*

segment-expression:
  pm-expression
  segment-expression **:>** *pm-expression*

pm-expression:
  cast-expression
  pm-expression **.*** *cast-expression*
  pm-expression **–>*** *cast-expression*

cast-expression:
  unary-expression
  (*type-name* **)** *cast-expression*

unary-expression:
postfix-expression
++ *unary-expression*
-- *unary-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof (*type-name*)
allocation-expression
deallocation-expression

unary-operator: one of
*** & + - ! ~**

allocation-expression:
: :$_{opt}$ *new nmodel*$_{opt}$ *placement*$_{opt}$ *new-type-name new-initializer*$_{opt}$
: :$_{opt}$ *new nmodel*$_{opt}$ *placement*$_{opt}$ (*type-name*) *new-initializer*$_{opt}$

placement:
(*expression-list*)

new-type-name:
type-specifier-list new declarator$_{opt}$

new-declarator:
ms-modifier-list$_{opt}$ * *cv-qualifier-list*$_{opt}$ *new-declarator*$_{opt}$
ms-modifier-list$_{opt}$ *complete-class-name* **: :** **cv-qualifier-list*$_{opt}$
new-declarator$_{opt}$
new-declarator$_{opt}$ [*expression*]

new-initializer:
(*initializer-list*)

deallocation-expression:
: :$_{opt}$ **delete** *cast-expression*
: :$_{opt}$ **delete** [ ] *cast-expression*

postfix-expression:
primary-expression
postfix-expression [*expression*]
postfix-expression (*expression-list*)
simple-type-name (*expression-list*)
postfix-expression **.** *name*
postfix-expression **->** *name*
postfix-expression **++**
postfix-expression **--**

expression-list:
  assignment-expression
  expression-list , *assignment-expression*

primary-expression:
  literal
  this
  : : *identifier*
  : : *operator-function-name*
  : : *qualified-name* (*expression*)
  name

name:
  identifier
  operator-function-name
  conversion-function-name
    ~ *class-name*
  qualified-name

qualified-name:
  ms-modifier-list$_{opt}$ *qualified-class-name* : : *name*

literal:
  integer-constant
  character-constant
  floating-constant
  string-literal

C.3 Declarations

declaration:
  decl-specifiers$_{opt}$ *declarator-list*$_{opt}$;
  asm-declaration
  function-definition
  template-declaration
  linkage-specification

decl-specifier:
  storage-class-specifier
  type-specifier
  fct-specifier
  friend
  typedef

decl-specifiers:
  decl-specifiers$_{opt}$ *decl-specifier*

storage-class-specifier:
auto
register
static
extern

fct-specifier:
inline
virtual

type-specifier:
simple-type-name
class-specifier
enum-specifier
elaborated-type-specifier
const
volatile

simple-type-name:
complete-class-name
qualified-type-name
char
short
int
long
signed
unsigned
float
double
void
__segment

elaborated-type-specifier:
class-key $rmodel_{opt}$ *identifier*
class-key $rmodel_{opt}$ *class-name*
enum-name

class-key:
class
struct
union

qualified-type-name:
typedef-name
class-name **::** *qualified-type-name*

complete-class-name:
  qualified-class-name
  :: *qualified-class-name*

qualified-class-name:
  class-name
  class-name **::** *qualified-class-name*

enum-specifier:
  enum *identifier*$_{opt}$ { *enum-list*$_{opt}$ }

enum-list:
  enumerator
  enum-list , *enumerator*

enumerator:
  identifier
  identifier = *constant-expression*

constant-expression:
  conditional-expression

linkage-specification:
  extern *string-literal* { *declaration-list*$_{opt}$ }
  extern *string-literal declaration*

declaration-list:
  declaration
  declaration-list declaration

C.4 Declarators

declarator-list:
  init-declarator
  declarator-list , *init-declarator*

init-declarator:
  ms-modifier-list$_{opt}$ *declarator initializer*$_{opt}$

declarator:
  dname
  ptr-operator declarator
  declarator (*argument-declaration-list*) *cv-mod-list*$_{opt}$
  declarator [*constant-expression*$_{opt}$]
  (*declarator*)

cv-mod-list:
cv-qualifier *cv-mod-list*$_{opt}$
rmodel *cv-mod-list*$_{opt}$

ptr-operator:
ms-modifier-list$_{opt}$ * *cv-qualifier-list*$_{opt}$
ms-modifier-list$_{opt}$ & *cv-qualifier-list*$_{opt}$
ms-modifier-list$_{opt}$ *complete-class-name* : : * *cv-qualifier-list*$_{opt}$

cv-qualifier-list:
cv-qualifier *cv-qualifier-list*$_{opt}$

cv-qualifier:
const
volatile

dname:
name
class-name
~ *class-name*
typedef-name
qualified-type-name

type-name:
type-specifier-list *ms-modifier-list*$_{opt}$ *abstract-declarator*$_{opt}$

type-specifier-list:
type-specifier *type-specifier-list*$_{opt}$

abstract-declarator:
ptr-operator *ms-modifier-list*$_{opt}$ *abstract-declarator*$_{opt}$
abstract-declarator$_{opt}$ (*argument-declaration-list*) *cv-qualifier-list*$_{opt}$
abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
(*ms-modifier-list*$_{opt}$ *abstract-declarator*)

argument-declaration-list:
arg-declaration-list$_{opt}$ . . .$_{opt}$
arg-declaration-list , . . .

arg-declaration-list:
argument-declaration
arg-declaration-list , *argument-declaration*

argument-declaration:
  decl-specifiers ms-modifier-list$_{opt}$ *declarator*
  decl-specifiers ms-modifier-list$_{opt}$ *declarator* = *expression*
  decl-specifiers ms-modifier-list$_{opt}$ *abstract-declarator*$_{opt}$
  decl-specifiers ms-modifier-list$_{opt}$ *abstract-declarator*$_{opt}$ = *expression*

function-definition:
  decl-specifiers$_{opt}$ *ms-modifier-list*$_{opt}$ *declarator ctor-initializer*$_{opt}$ *fct-body*

fct-body:
  compound-statement

initializer:
  = *expression*
  = { *initializer-list* ,$_{opt}$ }
  (*expression-list*)

initializer-list:
  expression
  initializer-list , *expression*
  { *initializer-list* ,$_{opt}$ }

C.5 Classes

class-specifier:
  class-head { *member-list*$_{opt}$ }

class-head:
  class-key ambient-model$_{opt}$ *identifier*$_{opt}$ *base-spec*$_{opt}$
  class-key ambient-model$_{opt}$ *class-name base-spec*$_{opt}$

member-list:
  member-declaration member-list$_{opt}$
  access-specifier : *member-list*$_{opt}$

member-declaration:
  decl-specifiers$_{opt}$ *member-declaration-list*$_{opt}$;
  function-definition ;$_{opt}$
  qualified-name ;

member-declarator-list:
  member-declarator
  member-declarator-list , *member-declarator*

member-declarator:
  ms-modifier-list$_{opt}$ *declarator pure-specifier*$_{opt}$
  identifier$_{opt}$ **:** *constant-expression*

pure-specifier:
  = 0

base-spec:
  : *base-list*

base-list:
  base-specifier
  base-list **,** *base-specifier*

base-specifier:
  complete-class-name
  virtual *access-specifier*$_{opt}$ *complete-class-name*
  access-specifier **virtual**$_{opt}$ *complete-class-name*

access-specifier:
  private
  protected
  public

conversion-function-name:
  operator *conversion-type-name*

conversion-type-name:
  type-specifier-list ptr-operator$_{opt}$

ctor-initializer:
  : *mem-initializer-list*

mem-initializer-list:
  mem-initializer
  mem-initializer **,** *mem-initializer-list*

mem-initializer:
  complete-class-name **(** *expression-list*$_{opt}$ **)**
  identifier **(** *expression-list*$_{opt}$ **)**

operator-function-name:
  operator *operator*

operator: one of
new delete
+ - * / % ^ & | ~
! = < > += -= *= /= %=
^= &= |= << >> >>= <<= == !=
<= >= && || ++ -- , ->* ->
() []

C.6 Statements

statement:
labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
declaration-statement
asm-statement

labeled-statement:
identifier **:** *statement*
case *constant-expression* **:** *statement*
default : *statement*

expression-statement:
$expression_{opt}$ **;**

compound-statement:
{ $statement\text{-}list_{opt}$ **}**

statement-list:
statement
statement-list statement

selection-statement:
if (*expression* **)** *statement*
if (*expression* **)** *statement* **else** *statement*
switch (*expression* **)** *statement*

iteration-statement:
while (*expression* **)** *statement*
do *statement* **while (** *expression* **) ;**
for (*for-init-statement* $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*

for-init-statement:
  expression-statement
  declaration-statement

jump-statement:
  break ;
  continue ;
  return *expression*$_{opt}$;
  goto *identifier* ;

declaration-statement:
  declaration

C.7 Preprocessor

#define *identifier token-string*
#define *identifier* (*identifier* , ... , *identifier*) *token-string*

#include "*filename*"
#include <*filename*>

#line *constant* "*filename*"$_{opt}$
#undef *identifier*

conditional:
  if-part elif-parts$_{opt}$ *else-part*$_{opt}$ *endif-line*

if-part:
  if-line text

if-line:
  # if *constant-expression*
  # ifdef *identifier*
  # ifndef *identifier*

elif-parts:
  elif-line text
  elif-parts elif-line text

elif-line:
  # elif *constant-expression*

else-part:
  else-line text

else-line:
  # else

endif-line:
  # endif

C.8 Microsoft Extensions

asm-statement:
  __asm *assembly-instruction* ;$_{opt}$
  __asm { *assembly-instruction-list* } ;$_{opt}$

assembly-instruction-list:
  assembly-instruction ;$_{opt}$
  assembly-instruction ; *assembly-instruction-list* ;$_{opt}$

Note The definition of *assembly-instruction* and more information about inline assembly can be found in the *Programming Techniques* manual.

amodel:
  __near
  __far

rmodel:
  amodel
  __huge

nmodel:
  rmodel
  __based (*expression*)

ambient-model:
  amb-export
  amb-model

amb-model:
  amodel amb-model$_{opt}$
  __novtordisp *amb-model*$_{opt}$ (reserved for future implementations)

amb-export:
__**export** *amb-export*$_{opt}$
__**novtordisp** *amb-export*$_{opt}$ (reserved for future implementations)

ms-modifier-list:
ms-modifier ms-modifier-list$_{opt}$

Note Many combinations of these modifiers, while syntactically legal, are invalid because of the context in which they appear. For example, __**fastcall** is not permitted when declaring data.

ms-modifier:
__**cdecl**
__**fortran**
__**pascal**
__**fastcall**
__**interrupt**
__**export**
__**saveregs**
__**loadds**
__**stdcall** (reserved for future implementations)
__**syscall** (reserved for future implementations)
__**oldcall** (reserved for future implementations)
__**novtordisp** (reserved for future implementations)
rmodel
based-modifier

based-modifier:
__**based** (*based-type*)

based-type:
void
__**self**
__**segname** (*string-literal*)
(__**segment**) *expression*
__**segment** (*expression*)
name

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Microsoft®

1191 Part No. 24772