C++ Language System Product Reference Manual

Chapter 1
Introduction

About This Guide

This manual describes the C++ programming language as of May 1991. C++ is a general–purpose programming language based on the C programming language

"The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, 1978 and 1988.

. In addition to the facilities provided by ANSI C, C++ provides classes, inline functions, operator overloading, function name overloading, constant types, references, free store management operators, and function argument checking and type conversion. These extensions to ANSI C are summarized in **"Extensions"** . The differences between C++ and ANSI C

American National Standard X3.159–1989.

are summarized in **"C++ and ANSI C"** . The extensions to C++ since the 1985 edition of this manual are summarized in **"Extensions"** . The section related to exception handling in **Chapter 15, "Exception Handling,"** is a placeholder for planned language extensions.

This reference manual is by Bjarne Stroustrup.

Note: This is the May 1991 version of the *C++ Reference Manual*

Audience

This manual assumes that you are familiar with the language as described in *The C++ Programming Language* (see **"Recommended Reading on C++"**) and that you have sufficient experience with C++ to recognize many of the problems that the features described in this manual are designed to solve or alleviate.

Document Overview

This manual is organized like this:

- **Chapter 1, "Introduction,"** provides an overview of the organization and of the conventions used in this manual.
- **Chapter 2, "Lexical Conventions,"** reviews the development of tokens after preprocessing including identifiers, keywords, operators, literals and other separators.
- **Chapter 3, "Basic Concepts,"** discusses name declarations and definitions, and the use of a name within a region of program text called its scope. Program and linkage, types, and storage classes are also covered in this chapter.
- **Chapter 4, "Standard Conversions,"** summarizes the conversions demanded by most ordinary operators and explains the result to be expected from such conversions.
- **Chapter 5, "Expressions,"** defines the syntax, order of evaluation, and meaning of expressions.
- **Chapter 6, "Statements,"** presents the ordered execution sequence for statements and describes the various types of statements and their usage.
- **Chapter 7, "Declarations,"** provides a detailed look at the declaration form and shows the use of the declaration specifier to indicate the fundamental type, storage class, or other properties of the objects and functions being declared.
- **Chapter 8, "Declarators,"** explains the use of declarators to state explictly the names of objects and functions, and to modify the type with operators.
- **Chapter 9, "Classes,"** defines class name, class members, member functions, static members,

unions, bit−fields, nested and local class declarations, and local type names.

- **Chapter 10, "Derived Classes,"** reviews mutiple base classes, virtual functions, and abstract classes, and also provides a summary of scope rules.
- **Chapter 11, "Member Access Control,"** explains the access rules (private, protected and public) that are available for members of a class.
- **Chapter 12, "Special Member Functions,"** describes the way objects of a class are created, copied, and destroyed, and how values may be converted to values of other types.
- **Chapter 13, "Overloading,"** defines and discusses overloaded operators. Operators can be overloaded, that is, given meaning when applied to expressions of class type. Addressing of overloaded functions, argument matching and declaration matching are reviewed.
- **Chapter 14, "Templates,"** contains information about class templates, type equivalence, function templates, declarations and definitions, member function templates, friends, static members and variables.
- **Chapter 15, "Exception Handling,"** status as a placeholder for planned language extensions is presented.
- **Chapter 16, "Preprocessing,"** reviews the phases of preprocessing, macro definition, conditional compilation, and file inclusion.
- **Appendix A, "Grammar Summary,"** presents a brief review of C++ syntax as an aid to comprehension.
- **Appendix B, "Compatibility,"** provides a brief summary of extensions to ANSI C provided by C++, demonstates the differences in language features between C++ and ANSI C, and discusses ANSI C to C++ language anachronisms.

Recommended Reading on C++

For a precise description of the C++ language, the following is indispensable:

- Ellis, Margaret A., and Stroustrup, Bjarne. *The Annotated C++ Reference Manual* (Reading, Mass.: Addison−Wesley, 1990)

The following books are recommended reading on programming in C++:

- Lippman, Stanley B. *C++ Primer* Second Edition (Reading, Mass.: Addison−Wesley, 1991)
- Stroustrup, Bjarne. *The C++ Programming Language* Second Edition (Addison−Wesley, 1991)

Notation and Syntax Conventions

This guide uses the following notation and syntax conventions:

italic	Indicates arguments in a command line that you must replace with a valid value. In text, it is used to indicate commands, document titles, file names, glossary items, new terms, and variables.
`courier`	Indicates computer output and program listings.
`courier bold`	Indicates user input to the computer and nonprinting keys.
[ ]	Brackets enclose optional command arguments. (Do not enter the brackets.)
. . .	An ellipsis indicates that the preceding optional items can appear more than once in succession.
|	The vertical bar separates items from which you can choose one.

Chapter 2

Lexical Conventions

This chapter contains the following subsections:

- **"Tokens"**
- **"Comments"**
- **"Identifiers"**
- **"Keywords"**
- **"Literals"**

A C++ program consists of one or more *files* (**"Program and Linkage"** "). A file is conceptually translated in several phases. The first phase is preprocessing (**Chapter 16, "Preprocessing"**), which performs file inclusion and macro substitution. Preprocessing is controlled by directives introduced by lines having # as the first character other than white space (""**Tokens"**). The result of preprocessing is a sequence of tokens. Such a sequence of tokens, that is, a file after preprocessing is called a *translation unit*.

Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, "white space"), as described below, are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

Comments

The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters // start a comment, which terminates at the end of the line on which they occur. The comment characters //, /*, and */ have no special meaning within a // comment and are treated just like other characters. Similarly, the comment characters // and /* have no special meaning within a /* comment.

Identifiers

An identifier is an arbitrarily long sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper−case and lower−case letters are different. All characters are significant.

Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise.

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

In addition, identifiers containing a double underscore (__) are reserved for use by C++ implementations and standard libraries and should be avoided by users.

The ASCII representation of C++ programs uses the following characters as operators or for punctuation:

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

and the following character combinations are used as operators:

```
−> ++ −− .* −>* << >>  <=  >= == != &&
|| *= /= %= +=  −= <<= >>= &= ^= |= ::
```

Each is a single token.

In addition, the following tokens are used by the preprocessor:

```
#   ##
```

Certain implementation−dependent properties, such as the type of a *sizeof* (**"Sizeof" in Chapter 5**) and the ranges of fundamental types (**"Types" in Chapter 3**), are defined in the standard header files

```
<float.h>   <limits.h>   <stddef.h>
```

These header files are part of the ANSI C standard. In addition the headers

```
<new.h>   <stdarg.h>   <stdlib.h>
```

define the types of the most basic library functions. The last two headers are part of the ANSI C standard; *<new.h>* is C++ specific.

Literals

There are several kinds of literals (often referred to as "constants").

literal:
integer−constant
character−constant
floating−constant
string−literal"

Integer Constants

An integer constant consisting of a sequence of digits is taken to be decimal (base ten) unless it begins with 0 (digit zero). A sequence of digits starting with 0 is taken to be an octal integer (base eight). The digits 8 and 9 are not octal digits. A sequence of digits preceded by *0x* or *0X* is taken to be a hexadecimal integer (base sixteen). The hexadecimal digits include *a* or *A* through *f* or *F* with decimal values ten through fifteen. For example, the number twelve can be written 12, 014, or *XC*.

The type of an integer constant depends on its form, value, and suffix. If it is decimal and has no suffix, it has the first of these types in which its value can be represented: *int*, *long int*, *unsigned long int*. If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented: *int*, *unsigned int*, *long int*, *unsigned long int*. If it is suffixed by *u* or *U*, its type is the first of these types in which its value can be represented: *unsigned int*, *unsigned long int*. If it is suffixed by *l* or *L*, its type is the first of these types in which its value can be represented: *long int*, *unsigned long int*. If it is suffixed by *ul*, *lu*, *uL*, *Lu*, *Ul*, *lU*, *UL*, or *LU*, its type is *unsigned long int*.

Character Constants

A character constant is one or more characters enclosed in single quotes, as in *'x'*. Single character constants have type *char*. The value of a single character constant is the numerical value of the character in the machine's character set. Multicharacter constants have type *int*. The value of a multicharacter constant is implementation dependent.

Certain non−graphic characters, the single quote *'*, the double quote *"*, the question mark *?*, and the backslash **, may be represented according to the escape sequences shown in Table 2−1.

Table 2−1 Escape Sequences

Character Name	ASCII Name	Escape Sequence
new−line	NL (LF)	\n
horizontal tab	HT	
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
alert	BEL	\a
backslash	\	\\

question mark	?	\?
single quote	'	\'
double quote	"	\"
octal number	*ooo*	*ooo*
hex number	*hhh*	\x*hhh*

If the character following a backslash is not one of those specified, the behavior is undefined. An escape sequence specifies a single character.

The escape *\ooo* consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape *\xhhh* consists of the backslash followed by *x* followed by a sequence of hexadecimal digits that are taken to specify the value of the desired character. There is no limit to the number of hexadecimal digits in the sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character constant is implementation dependent if it exceeds that of the largest *char*.

A character constant immediately preceded by the letter *L* for example, *L'ab'*, is a wide−character constant. A wide−character constant is of type *wchar_t*, an integral type (**"Fundamental Types"**) defined in the standard header *<stddef.h>*. Wide−characters are intended for character sets where a character does not fit into a single byte.

Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an *e* or *E*, an optionally signed integer exponent, and an optional type suffix. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the letter *e* (or *E*) and the exponent (not both) may be missing. The type of a floating constant is *double* unless explicitly specified by a suffix. The suffixes *f* and *F* specify *float*, the suffixes *l* and *L* specify *long double*.

String Literals

A string literal is a sequence of characters (as defined in **"Character Constants"**) surrounded by double quotes, as in *"..."*. A string has type "array of *char"* and storage class *static* (**"Storage Classes"**), and is initialized with the given characters. Whether all string literals are distinct (that is, are stored in non−overlapping objects) is implementation dependent. The effect of attempting to modify a string literal is undefined.

Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct. For example,

```
"\xA" "B"
```

contains the two characters *'\xA'* and *'B'* after concatenation (and not the single hexadecimal character *'\xAB'*).

After any necessary concatenation, *'\0'* is appended so that programs that scan a string can find its end. The size of a string is the number of its characters including this terminator. Within a string, the double quote character " must be preceded by a **.

A string literal immediately preceded by the letter *L*, for example, *L"asdf"*, is a wide−character string. A wide−character string is of type "array of *wchar_t*," where *wchar_t* is an integral type defined in the standard header *<stddef.h>*. Concatenation of ordinary and wide−character string literals is undefined.

Chapter 3

Basic Concepts

This chapter contains the following subsections:

- **"Declarations and Definitions"**
- **"Scopes"**
- **"Program and Linkage"**

- **"Start and Termination"**
- **"Storage Classes"**
- **"Types"**
- **"Lvalues"**

A name denotes an object, a function, a set of functions, an enumerator, a type, a class member, a template, a value, or a label. A name is introduced into a program by a declaration. A name can be used only within a region of program text called its *scope*. A name has a type, which determines its use. A name used in more than one translation unit may (or may not) refer to the same object, function, type, template, or value in these translation units depending on the linkage (**"Program and Linkage"**) specified in the translation units.

An object is a region of storage (**"Lvalues"**). A named object has a storage class (**"Storage Classes"**) that determines its lifetime. The meaning of the values found in an object is determined by the type of the expression used to access it.

Declarations and Definitions

A declaration (**Chapter 7, "Declarations"**) introduces one or more names into a program. A declaration is a definition unless it declares a function without specifying the body (**"Function Definitions"**), it contains the *extern* specifier (**"Storage Class Specifiers"**) and no initializer or function body, it is the declaration of a static data member in a class declaration (**"Static Members"**), it is a class name declaration (**"Class Names"**), or it is a typedef declaration (**"The typedef Specifier"**). The following, for example, are definitions:

```
int a;
extern const c = 1;
int f(int x) { return x+a; }
struct S { int a; int b; };
enum { up, down };
```

whereas these are just declarations:

```
extern int a;
extern const c;
int f(int);
struct S;
typedef int Int;
```

There must be exactly one definition of each object, function, class, and enumerator used in a program (**"Program and Linkage"**). If a function is never called and its address is never taken, it need not be defined. Similarly, if the name of a class is used only in a way that does not require its definition to be known, it need not be defined.

Scopes

There are four kinds of scope: local, function, file, and class.

- *Local*: A name declared in a block (**"Compound Statement, or Block"**) is local to that block and can be used only in it and in blocks enclosed by it after the point of declaration. Names of formal arguments for a function are treated as if they were declared in the outermost block of that function.
- *Function*: Labels (**"Labeled Statement"**) can be used anywhere in the function in which they are declared. Only labels have function scope.
- *File*: A name declared outside all blocks (**"Compound Statement, or Block"**) and classes (**Chapter 9, "Classes"**) has file scope and can be used in the translation unit in which it is declared after the point of declaration. Names declared with *file* scope are said to be *global.*
- *Class*: The name of a class member is local to its class and can be used only in a member function of that class (**"Member Functions"**), after the . operator applied to an object of its class (**"Class Member Access"**) or a class derived from (**Chapter 10, "Derived Classes"**) its class, after the –> operator applied to a pointer to an object of its class (**"Class Member Access"**) or a class derived

from its class, or after the *::* scope resolution operator (**"Primary Expressions"**) applied to the name of its class or a class derived from its class. A name first declared by a *friend* declaration (**"Friends"**) belongs to the same scope as the class containing the *friend* declaration. A class first declared in a return or argument type belongs to the global scope.

Special rules apply to names declared in function argument declarations (**"Functions"**), and friend declarations (**"Friends"**).

A name may be hidden by an explicit declaration of that same name in an enclosed block or in a class. A hidden class member name can still be used when it is qualified by its class name using the *::* operator (**"Primary Expressions"** , **"Static Members"** , **Chapter 10, "Derived Classes"**). A hidden name of an object, function, type, or enumerator with file scope can still be used when it is qualified by the unary*::* operator (**"Primary Expressions"**). In addition, a class name (**"Class Names"**) may be hidden by the name of an object, function, or enumerator declared in the same scope. If a class and an object, function, or enumerator are declared in the same scope (in any order) with the same name the class name is hidden. A class name hidden by a name of an object, function, or enumerator in local or class scope can still be used when appropriately (**"Type Specifiers"**) prefixed with *class*, *struct*, or *union*. Similarly, a hidden enumeration name can be used when appropriately (**"Type Specifiers"**) prefixed with *enum*. The scope rules are summarized in **"Summary of Scope Rules"** .

The *point of declaration* for a name is immediately after its complete declarator (**Chapter 8, "Declarators"**) and before its initializer (if any). For example,

```
int x = 12;
{ int x = x; }
```

Here the second *x* is initialized with its own (unspecified) value.

The point of declaration for an enumerator is immediately after the identifier that names it. For example,

```
enum { x = x };
```

Here, again, the enumerator *x* is initialized to its own (uninitialized) value.

Program and Linkage

A program consists of one or more files (**Chapter 2, "Lexical Conventions"**) linked together. A file consists of a sequence of declarations.

A name of file scope that is explicitly declared *static* is local to its translation unit and can be used as a name for other objects, functions, and so on, in other translation units. Such names are said to have internal linkage. A name of file scope that is explicitly declared *inline* is local to its translation unit. A name of file scope that is explicitly declared *const* and not explicitly declared *extern* is local to its translation unit. As is the name of a class that has not been used in the declaration of an object, function, or class that is not local to its translation unit and has no static members (**"Static Members"**) and no non−inline member functions (**"Inline Member Functions"**).

Every declaration of a particular name of file scope that is not declared to have internal linkage in one of the above mentioned ways in a multifile program refers to the same object (**"Lvalues"**), function (**"Functions"**), or class (**Chapter 9, "Classes"**). Such names are said to be external, or to have external linkage. In particular, since it is not possible to declare a class name *static*, every use of a particular file scope class name that has been used in the declaration of an object or function with external linkage or has a static member or a non−inline member function refers to the same class.

Typedef names (**"The typedef Specifier"**), enumerators (**"Enumeration Declarations"**), and template names (**Chapter 14, "Templates"**) do not have external linkage.

Static class members (**"Static Members"**) have external linkage.

Non−inline class member functions have external linkage. Inline class member functions must have exactly one definition in a program.

Local names (**"Scopes"**) explicitly declared *extern* have external linkage unless already declared *static* (**"Storage Class Specifiers"**).

The types specified in all declarations of a particular external name must be identical except for the use of typedef names (**"The typedef Specifier"**) and unspecified array bounds (**"Arrays"**). There must be exactly one definition for each function, object, class and enumerator used in a program. If, however, a function is never called and its address is never taken, it need not be defined. Similarly, if the name of a class is used only in a way that does not require its definition to be known, it need not be defined.

A function may be defined only in file or class scope.

Linkage to non−C++ declarations can be achieved using a *linkage−specification* (**"Linkage Specifications"**).

Start and Termination

A program must contain a function called *main()*. This function is the designated start of the program. This function is not predefined by the compiler, it cannot be overloaded, and its type is implementation dependent. It is recommended that the two examples below be allowed on any implementation and that any further arguments required be added after *argv*. The function *main()* may be defined as

```
int main() { /* ... */ }
or
int main(int argc, char* argv[]) { /* ... */ }
```

In the latter form *argc* shall be the number of parameters passed to the program from an environment in which the program is run. If *argc* is non−zero these parameters shall be supplied as zero−terminated strings in *argv0* through *argvargc−1* and *argv0* shall be the name used to invoke the program or *""*. It is guaranteed that *argv[argc]==0*.

The function *main()* may not be called from within a program. The linkage (**"Program and Linkage"**) of *main()* is implementation dependent. The address of *main()* cannot be taken and *main()* may not be declared *inline* or *static*.

Calling the function

```
void exit(int);
```

declared in *<stdlib.h>* terminates the program. The argument value is returned to the program's environment as the value of the program.

A return statement in *main()* has the effect of calling *exit()* with the return value as the argument.

The initialization of non−local static objects (**"Storage Classes"**) in a translation unit is done before the first use of any function or object defined in that translation unit. Such initializations (**"Initializers"** , **"Static Members"** , **"Constructors"** , **"Explicit Initialization"**) may be done before the first statement of *main()* or deferred to any point in time before the first use of a function or object defined in that translation unit. The default initialization of all static objects to zero (**"Initializers"**) is performed before any dynamic (that is, run−time) initialization. No further order is imposed on the initialization of objects from different translation units. The initialization of local static objects is described in **"Initializers"** .

Destructors (**"Destructors"**) for initialized static objects are called when returning from *main()* and when calling *exit()*. Destruction is done in reverse order of initialization. The function *atexit()* from *<stdlib.h>* can be used to specify that a function must be called at exit. If *atexit()* is to be called, objects initialized before an *atexit()* call may not be destroyed until after the function specified in the *atexit()* call has been called. Where a C++ implementation coexists with a C implementation, any actions specified by the C implementation to take place after the *atexit()* functions have been called take place after all destructors have been called.

Calling the function

```
void abort();
```

declared in *<stdlib.h>* terminates the program without executing destructors for static objects and without calling the functions passed to *atexit()*.

Storage Classes

There are two declarable storage classes: automatic and static.

- *Automatic* objects are local to each invocation of a block.
- *Static* objects exist and retain their values throughout the execution of the entire program.

Automatic objects are initialized (**"Constructors"**) each time the control flow reaches their definition and destroyed (**"Destructors"**) on exit from their block (**"Declaration Statement"**).

A named automatic object may not be destroyed before the end of its block nor may an automatic named object of a class with a constructor or a destructor with side effects be eliminated even if it appears to be unused.

Similarly, a global object of a class with a constructor or a destructor with side effects may not be eliminated even if it appears to be unused.

Static objects are initialized and destroyed as described in **"Start and Termination"** and **"Declaration Statement"** . Some objects are not associated with names; see **"New"** and **"Temporary Objects"** . All global objects have storage class *static*. Local objects and class members can be given static storage class by explicit use of the *static* storage class specifier (**"Storage Class Specifiers"**).

Types

There are two kinds of types: fundamental types and derived types.

Fundamental Types

There are several fundamental types. The standard header *<limits.h>* specifies the largest and smallest values of each for an implementation.

Objects declared as characters (*char*) are large enough to store any member of the implementation's basic character set. If a character from this set is stored in a character variable, its value is equivalent to the integer code of that character. Characters may be explicitly declared *unsigned* or *signed*. Plain *char*, *signed char*, and *unsigned char* are three distinct types. A *char*, a *signed char*, and an *unsigned char* consume the same amount of space.

Up to three sizes of integer, declared *short int*, *int*, and *long int* are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. Plain integers have the natural size suggested by the machine architecture; the other sizes are provided to meet special needs.

For each of the types *signed char*, *short*, *int*, and *long*, there exists a corresponding *unsigned* type, which occupies the same amount of storage and has the same alignment requirements. An *alignment requirement* is an implementation−dependent restriction on the value of a pointer to an object of a given type (**"Explicit Type Conversion"**).

Unsigned integers, declared *unsigned*, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. This implies that unsigned arithmetic does not overflow.

There are three *floating* types: *float*, *double*, and *long double*. The type *double* provides no less precision than *float*, and the type *long double* provides no less precision than *double*. An implementation will define the characteristics of the fundamental floating point types in the standard header *<float.h>*.

Types *char*, *int* of all sizes, and enumerations (**"Enumeration Declarations"**) are collectively called *integral* types. *Integral* and *floating* types are collectively called *arithmetic* types.

The *void* type specifies an empty set of values. It is used as the return type for functions that do not return a value. No object of type *void* may be declared. Any expression may be explicitly converted to type *void* (**"Explicit Type Conversion"**); the resulting expression may be used only as an expression statement (**"Expression Statement"**), as the left operand of a comma expression (**"Comma Operator"**), or as a second or third operand of *?:* (**"Conditional Operator"**).

Derived Types

There is a conceptually infinite number of derived types constructed from the fundamental types in the following ways:

- *arrays* of objects of a given type, **"Arrays"** ;
- *functions*, which take arguments of given types and return objects of a given type, **"Functions"** ;
- *pointers* to objects or functions of a given type, **"Pointers"** ;
- *references* to objects or functions of a given type, **"References"** ;
- *constants*, which are values of a given type, **"Type Specifiers"** ;
- *classes* containing a sequence of objects of various types (**Chapter 9, "Classes"**), a set of functions for manipulating these objects (**"Member Functions"**), and a set of restrictions on the access to these objects and functions, **Chapter 11, "Member Access Control"** ;
- *structures*, which are classes without default access restrictions, **Chapter 11, "Member Access Control"** ;
- *unions*, which are structures capable of containing objects of different types at different times, **"Unions"** ;
- *pointers to class members*, which identify members of a given type within objects of a given class, **"Pointers to Members"** .

In general, these methods of constructing objects can be applied recursively; restrictions are mentioned in **"Pointers"** , **"Arrays"** , **"Functions"** , and **"References"** .

A pointer to objects of a type *T* is referred to as a "pointer to *T*". For example, a pointer to an object of type *int* is referred to as "pointer to *int*" and a pointer to an object of class *X* is called a "pointer to *X*."

Objects of type *void** (pointer to void), *const void**, and *volatile void** can be used to point to objects of unknown type. A *void** must have enough bits to hold any object pointer.

Except for pointers to static members, text referring to "pointers" does not apply to pointers to members.

Type Names

Fundamental and derived types can be given names by the *typedef* mechanism (**"The typedef Specifier"**), and families of types and functions can be specified and named by the *template* mechanism (**Chapter 14, "Templates"**).

Lvalues

An *object* is a region of storage; an *lvalue* is an expression referring to an object or function. An obvious example of an *lvalue* expression is the name of an object. Some operators yield *lvalues*. For example, if *E* is an expression of pointer type, then **E* is an *lvalue* expression referring to the object to which *E* points. The name *"lvalue"* comes from the assignment expression *E1 = E2* in which the left operand *E1* must be an *lvalue* expression. The discussion of each operator in **Chapter 5, "Expressions"** , indicates whether it expects *lvalue* operands and whether it yields an *lvalue*. An *lvalue* is *modifiable* if it is not a function name, an array name, or *const*.

Chapter 4
Standard Conversions

This chapter contains the following subsections:
- **"Integral Promotions"**
- **"Integral Conversions"**
- **"Float and Double"**
- **"Floating and Integral"**
- **"Arithmetic Conversions"**
- **"Pointer Conversions"**
- **"Reference Conversions"**
- **"Pointers to Members"**

Some operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section summarizes the conversions demanded by most ordinary operators and explains the result to be expected from such conversions; it will be supplemented as required by the discussion of each operator. These conversions are also used in initialization (**"Initializers"** ,

"References" , **"Copying Class Objects"** , **"Constructors"**). **"Conversions"** and **"Argument Matching"** describe user–defined conversions and their interaction with standard conversions. The result of a conversion is an *lvalue* only if the result is a reference (**"References"**).

Integral Promotions

A *char*, a *short int*, enumerator, object of enumeration type (**"Enumeration Declarations"**), or an *int* bit–field (**"Bit–Fields"**) (in both their signed and unsigned varieties) may be used wherever an integer may be used. If an *int* can represent all the values of the original type, the value is converted to *int*; otherwise it is converted to *unsigned int*. This process is called *integral promotion*.

Integral Conversions

When an integer is converted to an *unsigned* type, the value is the least unsigned integer congruent to the signed integer (modulo 2^n where *n* is the number of bits used to represent the unsigned type). In a two's complement representation, this conversion is conceptual and there is no change in the bit pattern.

When an integer is converted to a signed type, the value is unchanged if it can be represented in the new type; otherwise the value is implementation dependent.

Float and Double

Single–precision floating point arithmetic may be used for *float* expressions. When a less precise floating value is converted to an equally or more precise floating type, the value is unchanged. When a more precise floating value is converted to a less precise floating type and the value is within representable range, the result may be either the next higher or the next lower representable value. If the result is out of range, the behavior is undefined.

Floating and Integral

Conversion of a floating value to an integral type truncates; that is, the fractional part is discarded. Such conversions are machine dependent; for example, the direction of truncation of negative numbers varies from machine to machine. The result is undefined if the value cannot be represented in the integral type.

Conversions of integral values to floating type are as mathematically correct as the hardware allows. Loss of precision occurs if an integral value cannot be represented exactly as a value of the floating type.

Arithmetic Conversions

Many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

- If either operand is of type *long double*, the other is converted to *long double*.
- Otherwise, if either operand is *double*, the other is converted to *double*.
- Otherwise, if either operand is *float*, the other is converted to *float*.
- Otherwise, the integral promotions (**"Integral Promotions"**) are performed on both operands.
- Then, if either operand is *unsigned long* the other is converted to *unsigned long*.
- Otherwise, if one operand is a *long int* and the other *unsigned int*, then if a *long int* can represent all the values of an *unsigned int*, the *unsigned int* is converted to a *long int*; otherwise both operands are converted to *unsigned long int*.
- Otherwise, if either operand is *long*, the other is converted to *long*.
- Otherwise, if either operand is *unsigned*, the other is converted to *unsigned*.
- Otherwise, both operands are *int*.

Pointer Conversions

The following conversions may be performed wherever pointers (**"Pointers"**) are assigned, initialized, compared, or otherwise used:

- A constant expression (**"Constant Expressions"**) that evaluates to zero is converted to a pointer, commonly called the null pointer. It is guaranteed that this value will produce a pointer distinguishable from a pointer to any object or function.
- A pointer to any non−*const* and non−*volatile* object type may be converted to a *void**.
- A pointer to function may be converted to a *void** provided a *void** has sufficient bits to hold it.
- A pointer to a class may be converted to a pointer to an accessible base class of that class (**Chapter 10, "Derived Classes"**) provided the conversion is unambiguous (**"Multiple Base Classes"**); a base class is accessible if its public members are accessible (**"Access Specifiers"**). The result of the conversion is a pointer to the base class sub−object of the derived class object. The null pointer (0) is converted into itself.
- An expression with type "array of *T"* may be converted to a pointer to the initial element of the array.
- An expression with type "function returning *T"* is converted to "pointer to function returning *T*" except when used as the operand of the address−of operator *&* or the function call *operator ()*.

Reference Conversions

The following conversion may be performed wherever references (**"References"**) are initialized (including argument passing (**"Function Call"**) and function value return (**"The return Statement"**)) or otherwise used:

- A reference to a class may be converted to a reference to an accessible base class (**Chapter 10, "Derived Classes"** , **"Access Specifiers"**) of that class (**"References"**) provided this conversion can be done unambiguously (**"Ambiguities"**). The result of the conversion is a reference to the base class sub−object of the derived class object.

Pointers to Members

The following conversion may be performed wherever pointers to members (**"Pointers to Members"**) are initialized, assigned, compared, or otherwise used:

- A constant expression (**"Constant Expressions"**) that evaluates to zero is converted to a pointer to member. It is guaranteed that this value will produce a pointer to member distinguishable from any other pointer to member.
- A pointer to a member of a class may be converted to a pointer to member of a class derived from that class provided the (inverse) conversion from the derived class to the base class pointer is accessible (**"Access Specifiers"**) and provided this conversion can be done unambiguously (**"Ambiguities"**).

The rule for conversion of pointers to members (from pointer to member of base to pointer to member of derived) appears inverted compared to the rule for pointers to objects (from pointer to derived to pointer to base) (**"Pointer Conversions"** , **Chapter 10, "Derived Classes"**). This inversion is necessary to ensure type safety.

Note that a pointer to member is not a pointer to object or a pointer to function and the rules for conversions of such pointers do not apply to pointers to members. In particular, a pointer to member cannot be converted to a *void**.

Chapter 5

Expressions

This chapter contains the following subsections:

- **"Primary Expressions"**
- **"Postfix Expressions"**
- **"Unary Operators"**
- **"Explicit Type Conversion"**

- **"Pointer − to − Member Operators"**
- **"Multiplicative Operators"**
- **"Additive Operators"**
- **"Shift Operators"**
- **"Relational Operators"**
- **"Equality Operators"**
- **"Bitwise AND Operator"**
- **"Bitwise Exclusive−OR Operator"**
- **"Bitwise Inclusive−OR Operator"**
- **"Logical AND Operator"**
- **"Logical OR Operator"**
- **"Conditional Operator"**
- **"Assignment Operators"**
- **"Comma Operator"**
- **"Constant Expressions"**

This chapter defines the syntax, order of evaluation, and meaning of expressions. . An expression is a sequence of operators and operands that specifies a computation. An expression may result in a value and may cause side effects.

Operators can be overloaded, that is, given meaning when applied to expressions of class type (**Chapter 9, "Classes"**). Uses of overloaded operators are transformed into function calls as described in **"Overloaded Operators"** . Overloaded operators obey the rules for syntax specified in this section, but the requirements of operand type, *lvalue*, and evaluation order are replaced by the rules for function call. Relations between operators, such as *++a* meaning *a+=1*, are not guaranteed for overloaded operators (**"Overloaded Operators"**).

This section defines the operators when applied to types for which they have not been overloaded. Operator overloading cannot modify the rules for operators applied to types for which they are defined by the language itself.

The order of evaluation of subexpressions is determined by the precedence and grouping of the operators. The usual mathematical rules for associativity and commutativity of operators may be applied only where the operators really are associative and commutative. Except where noted, the order of evaluation of operands of individual operators is undefined. In particular, if a value is modified twice in an expression, the result of the expression is undefined except where an ordering is guaranteed by the operators involved. For example,

```
i = v[i++];       // the value of ’i’ is undefined
i=7,i++,i++;      // ’i’ becomes 9
```

The handling of overflow and divide check in expression evaluation is implementation dependent. Most existing implementations of C++ ignore integer overflows. Treatment of division by zero and all floating−point exceptions vary among machines, and is usually adjustable by a library function.

Except where noted, operands of types *const T*, *volatile T*, *T&* , *const T&*, and *volatile T&* can be used as if they were of the plain type *T*. Similarly, except where noted, operands of type *T*const* and *T*volatile* can be used as if they were of the plain type *T**. Similarly, a plain *T* can be used where a *volatile T* or a *const T* is required. These rules apply in combination so that, except where noted, a *const T*volatile* can be used where a *T** is required. Such uses do not count as standard conversions when considering overloading resolution (**"Argument Matching"**).

If an expression has the type "reference to *T"* (**"References"**), the value of the expression is the object of type *"T"* denoted by the reference. The expression is an *lvalue*. A reference can be thought of as a name of an object.

User−defined conversions of class objects to and from fundamental types, pointers, and so on, can be defined (**"Conversions"**). If unambiguous (**"Argument Matching"**), such conversions may be applied by the compiler wherever a class object appears as an operand of an operator, as an initializer (**"Initializers"**), as the controlling expression in a selection (**"Selection Statements"**) or iteration (**"Iteration Statements"**) statement, as a function return value (**"The return Statement"**), or as a function argument (**"Function Call"**).

Primary Expressions

Primary expressions are literals, names, and names qualified by the scope resolution operator *::*.

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

A *literal* is a primary expression. Its type depends on its form (**"Literals"**).

In the body of a non−static member function (**"Member Functions"**), the keyword *this* names a pointer to the object for which the function was invoked.

The keyword *this* cannot be used outside a class member function body.

The operator *::* followed by an *identifier,* a *qualified−name,* or an *operator−function−name* is a primary expression. Its type is specified by the declaration of the identifier, name, or *operator−function−name.* The result is the identifier, name, or *operator−function−name.* The result is an *lvalue* if the identifier is of that class. The identifier or *operator−function−name* must be of file scope. Use of *::* allows a type, an object, a function, or an enumerator to be referred to even if its identifier has been hidden (**"Scopes"**).

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an *lvalue*.

A *name* is a restricted form of a *primary−expression* that can appear after . and −> (**"Class Member Access"**):

name:
identifier
operator−function−name
conversion−function−name
class−name
qualified−name

An *identifier* is a *name* provided it has been suitably declared (**Chapter 7, "Declarations"**). For *operator−function−names,* see **"Overloaded Operators"** . For *conversion−function−names,* see **"Conversion Functions"** . A *class−name* prefixed by ~ denotes a destructor; see **"Destructors"** .

qualified−name:
qualified−class−name :: name

A *qualified−class−name* (**"Type Specifiers"**) followed by *::* and the name of a member of that class (**"Class Members"**), or a member of a base of that class (**Chapter 10, "Derived Classes"**), is a *qualified−name;* its type is the type of the member. The result is the member. Also the result is an *lvalue* if the member is of that class. The *class−name* may be hidden by a non−type name, in which case the *class−name* is still found and used. Where *class−name :: class−name* or *class−name :: ~ class−name* is used, the two *class−names* must refer to the same class; this notation names constructors (**"Constructors"**) and destructors (**"Destructors"**), respectively. Multiply qualified names, such as *N1::N2::N3::n*, can be used to refer to nested types (**"Nested Class Declarations"**).

Postfix Expressions

Postfix expressions group left−to−right.

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

Subscripting

A postfix expression followed by an expression in square brackets is a postfix expression. The intuitive meaning is that of a subscript. One of the expressions must have the type "pointer to *T* " and the other must be of integral type. The type of the result is "*T*." The expression *E1[E2]* is identical (by definition) to **((E1)+(E2))*. See **"Unary Operators"** and **"Additive Operators"** for details of *** and *+* and **"Arrays"** for details of arrays.

Function Call

A function call is a postfix expression followed by parentheses containing a possibly empty, comma−separated list of expressions which constitute the actual arguments to the function. The postfix expression must be of type "function returning *T*," "pointer to function returning *T*," or "reference to function returning *T*," and the result of the function call is of type "*T*."

When a function is called, each formal argument is initialized (**"References"** , **"Copying Class Objects"** , **"Constructors"**) with its actual argument. Standard (**Chapter 4, "Standard Conversions"**) and user−defined (**"Conversions"**) conversions are performed. A function may change the values of its non−constant formal arguments, but these changes cannot affect the values of the actual arguments except where a formal argument is of a non−*const* reference type (**"References"**). Where a formal argument is of reference type a temporary variable is introduced if needed (**"Type Specifiers"** , **"Literals"** , **"String Literals"** , **"Arrays"** , **"Temporary Objects"**). In addition, it is possible to modify the values of non−constant objects through pointer arguments.

A function may be declared to accept fewer arguments (by declaring default arguments **"Default Arguments"**) or more arguments (by using the ellipsis, (...) **"Functions"**) than are specified in the function definition (**"Function Definitions"**).

A function can be called only if a declaration of it is accessible from the scope of the call. This implies that, except where the ellipsis (...) is used, a formal argument is available for each actual argument.

Any actual argument of type *float* for which there is no formal argument is converted to *double* before the call; any of *char*, *short*, enumeration, or a bit−field type for which there is no formal argument are converted to *int* or *unsigned* by integral promotion (**"Integral Promotions"**). An object of a class for which no formal argument is declared is passed as a data structure.

An object of a class for which a formal argument is declared is passed by initializing the formal argument with the actual argument by a constructor call before the function is entered (**"Temporary Objects"** , **"Copying Class Objects"**).

The order of evaluation of arguments is undefined; take note that compilers differ. All side effects of argument expressions take effect before the function is entered. The order of evaluation of the postfix expression and the argument expression list is undefined.

Recursive calls are permitted.

A function call is an *lvalue* only if the result type is a reference.

Explicit Type Conversion

A *simple−type−name* (**"Type Specifiers"**) followed by a parenthesized *expression−list* constructs a value of the specified type given by the expression list. If the expression list specifies more than a single value, the type must be a class with a suitably declared constructor (**"Initializers"** , **"Constructors"**).

A *simple−type−name* (**"Type Specifiers"**) followed by a (empty) pair of parentheses constructs a value of the specified type. If the type is a class with a suitably declared constructor that constructor will be called; otherwise the result is an undefined value of the specified type. See also (**"Explicit Type Conversion"**).

Class Member Access

A postfix expression followed by a dot *" . "* followed by a *name* is a postfix expression. The first expression must be a class object, and the *name* must name a member of that class. The result is the named member of the object, and it is an *lvalue* if the member is an *lvalue*.

A postfix expression followed by an arrow (*−>*) followed by a *name* is a postfix expression. The first expression must be a pointer to a class object and the *name* must name a member of that class. The result is the named member of the object to which the pointer points and it is an *lvalue* if the member is an *lvalue*. Thus the expression *E1−>MOS* is the same as *(*E1).MOS*.

Note that "class objects" can be structures (**"Class Members"**) and unions (**"Unions"**). Classes are discussed in **Chapter 9, "Classes"** .

Increment and Decrement

The value obtained by applying a postfix *++* is the value of the operand. The operand must be a modifiable *lvalue*. The type of the operand must be an arithmetic type or a pointer type. After the result is noted, the object is incremented by 1. The type of the result is the same as the type of the operand, but it is not an *lvalue*. See also **"Additive Operators"** and **"Assignment Operators"** .

The operand of postfix *−−* is decremented analogously to the postfix *++* operator.

Unary Operators

Expressions with unary operators group right−to−left.

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
  ** & + − ! ~*

The unary *** operator means *indirection:* the expression must be a pointer, and the result is an *lvalue* referring to the object to which the expression points. If the type of the expression is "pointer to *T*," the type of the result is "*T*."

The result of the unary *&* operator is a pointer to its operand. The operand must be a function, an *lvalue*, or a *qualified−name.* In the first two cases, if the type of the expression is "*T*," the type of the result is "pointer to *T*." In particular, the address of an object of type *const T* has type *const T**; *volatile* is handled similarly. For a *qualified−name,* if the member is not static and of type *"T"* in class "*C*," the type of the result is "pointer to member of *C* of type *T*." For a static member of type *T*, the type is plain "pointer to *T*." The address of an overloaded function (**Chapter 13, "Overloading"**) can be taken only in an initialization or an assignment where the left side uniquely determines which version of the overloaded function is referred to (**"Address of Overloaded Function" in Chapter 13**).

The operand of the unary *+* operator must have arithmetic or pointer type and the result is the value of the argument. Integral promotion is performed on integral operands. The type of the result is the type of the promoted operand.

The operand of the unary−operator must have arithmetic type and the result is the negation of its operand. Integral promotion is performed on integral operands. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is the number of bits in the promoted operand. The type of the result is the type of the promoted operand.

The operand of the logical negation operator *!* must have arithmetic type or be a pointer; its value is 1 if the value of its operand is 0 and 0 if the value of its operand is non−zero. The type of the result is *int*.

The operand of ~ must have integral type; the result is the one's complement of its operand. Integral promotions are performed. The type of the result is the type of the promoted operand.

Increment and Decrement

The operand of prefix *++* is incremented by 1 . The operand must be a modifiable *lvalue*. The type of the operand must be an arithmetic type or a pointer type. The value is the new value of the operand; it is an *lvalue*. The expression *++x* is equivalent to *x+=1*. See the discussions of addition (**"Additive Operators"**) and assignment operators (**"Assignment Operators"**) for information on conversions.

The operand of prefix *−−* is decremented analogously to the prefix *++* operator.

Sizeof

The *sizeof* operator yields the size, in bytes, of its operand. The operand is either an expression, which is not evaluated, or a parenthesized type name. The *sizeof* operator may not be applied to a function, a bit−field, an undefined class, the type *void*, or an array with an unspecified dimension. A *byte* is undefined by the language except in terms of the value of *sizeof*; *sizeof(char)* is 1.

When applied to a reference, the result is the size of the referenced object. When applied to a clas, the result is the number of bytes in an object of that class including any padding required for placing such objects in an array. The size of any class or class object is larger than zero. When applied to an array, the result is the total number of bytes in the array. This implies that the size of an array of *n* elements is *n* times the size of an element.

The *sizeof* operator may be applied to a pointer to a function, but not to a function.

The result is a constant of type *size_t*, an implementation−dependent unsigned integral type defined in the standard header *<stddef.h>*.

New

The *new* operator attempts to create an object of the *type−name* (**"Type Names"**) to which it is applied. This type must be an object type; functions cannot be allocated this way, though pointers to functions can.

allocation−expression:
   ::$_{opt}$ new placement$_{opt}$ new−type−name new−initializer$_{opt}$
   ::$_{opt}$ new placement$_{opt}$ (type−name) new−initializer$_{opt}$

placement:
   (expression−list)

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

new−declarator:
   ** cv−qualifier−list$_{opt}$ new−declarator$_{opt}$*
   *class−name :: * cv−qualifier−list$_{opt}$ new−declarator$_{opt}$*
   new−declarator$_{opt}$ [expression]

new−initializer:
   (initializer−list$_{opt}$)

The lifetime of an object created by *new* is not restricted to the scope in which it is created. The *new* operator returns a pointer to the object created. When that object is an array, a pointer to its initial element is returned. For example, both *new int* and *new int[10]* return an *int** and the type of *new int[i][10]* is *int (*)[10]*. Where an array type (**"Arrays"**) is specified all array dimensions but the first must be constant expressions (**"Constant Expressions"**) with positive values. The first array dimension can be a general *expression* even when the *type−name* is used (despite the general restriction of array dimensions in *type−names* to *constant−expressions* (**"Constant Expressions"**)).

This implies that an *operator new()* can be called with the argument zero. In this case, a pointer to an object is returned. Repeated such calls return pointers to distinct objects.

The *type−specifier−list* may not contain *const*, *volatile*, class declarations, or enumeration declarations.

The *new* operator will call the function *operator new()* to obtain storage (**"Free Store"**). A first argument of *sizeof(T)* is supplied when allocating an object of type *T*. The *placement* syntax can be used to supply additional arguments.

For example, *new T* results in a call of *operator new(sizeof(T))* and *new(2,f) T* results in a call *operator new(sizeof(T),2,f)*.

The *placement* syntax can be used only provided an *operator new()* with suitable argument types (**"Argument Matching"**) has been declared.

When an object of a non−class type (including arrays of class objects) is created with operator *new*, the global *::operator new()* is used. When an object of a class *T* is created with operator *new* , *T::operator new()* is used if it exists (using the usual lookup rules for finding members of a class and its base classes; **"Ambiguities"**); otherwise the global *::operator new()* is used. Using *::new* ensures that the global *::operator new()* is used even if *T::operator new()* exists.

A *new−initializer* may be supplied in an *allocation−expression.* For objects of classes with a constructor (**"Constructors"**) this argument list will be used in a constructor call; otherwise the initializer must be of the form *(expression)* or *()*. If present, the expression will be used to initialize the object; if not, the object will start out with an undefined value.

If a class has a constructor an object of that class can be created by *new* only if suitable arguments are provided or if the class has a default constructor (**"Constructors"**). Whether *operator new* allocates the memory itself or leaves that up to the constructor when creating an object of a class with a constructor is implementation dependent. Access and ambiguity control are done for both *operator new()* and the constructor; see **Chapter 12, "Special Member Functions"** .

No initializers can be specified for arrays. Arrays of objects of a class with constructors can be created by operator *new* only if the class has a default constructor (**"Constructors"**). In that case, the default constructor will be called for each element of the array.

Initialization is done only if the value returned by *operator new()* is non−zero. If the value returned by the *operator new()* is 0 (the null pointer) the value of the expression is 0.

The order of evaluation of the call to an *operator new()* to get memory and the evaluation of arguments to constructors is undefined. It is also undefined if the arguments to a constructor are evaluated if *operator new()* returns 0.

In a *new−type−name* used as the operand for *new*, parentheses may not be used. This implies that

```
new int(*[10])();        // error
```

is an error because the binding is

```
(new int) (*[10])();    // error
```

Objects of general type can be expressed using the explicitly parenthesized version of the *new* operator. For example,

```
new (int (*[10])());
```

allocates an array of pointers to functions (taking no argument and returning *int*).

The *new−type−name* in an *allocation−expression* is the longest possible sequence of *new−declarators.* This prevents ambiguities between declarator operators *&*, ***, *[]*, and their expression counterparts. For example,

```
new int*i;        // syntax error: parsed as '(new int*) i'
                    //                not as '(new int)*i'
```

The *** is the pointer declarator and not the multiplication operator.

Delete

The *delete* operator destroys an object created by the *new* operator.

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

The result has type *void*. The operand of *delete* must be a pointer returned by *new*. The effect of applying *delete* to a pointer not obtained from the *new* operator without a placement specification is undefined and usually harmful. Deleting a pointer with the value zero, however, is guaranteed to be harmless.

The effect of accessing a deleted object is undefined and the deletion of an object may change its value. Furthermore, if the expression denoting the object in a *delete* expression is a modifiable *lvalue*, its value is undefined after the deletion.

A pointer to constant cannot be deleted.

The *delete* operator will invoke the destructor (if any, **"Destructors"**) for the object pointed to.

To free the storage pointed to, the *delete* operator will call the function *operator delete()*; see **"Free Store"** . For objects of a non−class type (including arrays of class objects), the global *::operator delete()* is used. For an object of a class *T*, *T::operator delete()* is used if it exists (using the usual lookup rules for finding members of a class and its base classes; **"Ambiguities"**); otherwise the global *::operator delete()* is used. Using *::delete* ensures that the global *::operator delete()* is used even if *T::operator delete()* exists.

The form

```
delete [ ] cast−expression
```

is used to delete arrays. The expression points to an array. The destructors (if any) for the objects pointed to will be invoked.

The effect of deleting an array with the plain *delete* syntax is undefined, as is deleting an individual object with the *delete[]* syntax.

Explicit Type Conversion

An explicit type conversion can be expressed using either functional notation (**"Explicit Type Conversion"**) or the *cast* notation.

cast−expression:
 unary−expression
 (type−name) cast−expression

The *cast* notation is needed to express conversion to a type that does not have a *simple−type−name.*

Types may not be defined in casts.

Any type conversion not mentioned below and not explicitly defined by the user (**"Conversions"**) is an error.

Any type that can be converted to another by a standard conversion (**Chapter 4, "Standard Conversions"**) can also be converted by explicit conversion and the meaning is the same.

A pointer may be explicitly converted to any integral type large enough to hold it. The mapping function is implementation dependent, but is intended to be unsurprising to those who know the addressing structure of the underlying machine.

A value of integral type may be explicitly converted to a pointer. A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation dependent.

A pointer to one object type may be explicitly converted to a pointer to another object type (subject to the restrictions mentioned in this section). The resulting pointer may cause addressing exceptions on use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of the same or smaller size and back again without change. Different machines may differ in the number of bits in pointers and in alignment requirements for objects. Aggregates are aligned on the strictest boundary required by any of their constituents. A *void** is considered a pointer to object type.

A pointer to a class *B* may be explicitly converted to a pointer to a class *D* that has *B* as a direct or indirect base class if an unambiguous conversion from *D* to *B* exists (**"Pointer Conversions"** , **"Ambiguities"**) and if *B* is not a virtual base class (**"Multiple Base Classes"**). Such a cast from a base to a derived class assumes that the object of the base class is a sub−object of an object of the derived class; the resulting pointer points to the enclosing object of the derived class. If the object of the base class is not a sub−object of an object of the derived class, the cast may cause an exception.

The null pointer (0) is converted into itself.

A yet undefined class may be used in a pointer cast, in which case no assumptions will be made about class lattices (**"Multiple Base Classes"**).

An object may be explicitly converted to a reference type *X&* if a pointer to that object may be explicitly converted to an *X**. Constructors or conversion functions are not called as the result of a cast to a reference. Conversion of a reference to a base class to a reference to a derived class is handled similarly to the conversion of a pointer to a base class to a pointer to a derived class with respect to ambiguity, virtual classes, and so on.

The result of a cast to a reference type is an *lvalue*; the results of other casts are not. Operations performed on the result of a pointer or reference cast refer to the same object as the original (uncast) expression.

A pointer to function may be explicitly converted to a pointer to an object type provided the object pointer type has enough bits to hold the function pointer. A pointer to an object type may be explicitly converted to a pointer to function provided the function pointer type has enough bits to hold the object pointer. In both cases, use of the resulting pointer may cause addressing exceptions, or worse, if the subject pointer does not refer to suitable storage.

A pointer to a function may be explicitly converted to a pointer to a function of a different type. The effect of calling a function through a pointer to a function type that differs from the type used in the definition of the function is undefined. See also **"Pointer Conversions"** .

An object or a value may be converted to a class object (only) if an appropriate constructor or conversion operator has been declared (**"Conversions"**).

A pointer to member may be explicitly converted into a different pointer to member type when the two types are both pointers to members of the same class or when the two types are pointers to member functions of classes one of which is unambiguously derived from the other (**"Pointers to Members"**).

A pointer to an object of a *const* type can be cast into a pointer to a non−*const* type. The resulting pointer will refer to the original object. An object of a *const* type or a reference to an object of a *const* type can be cast into a reference to a non− *const* type. The resulting reference will refer to the original object. The result of attempting to modify that object through such a pointer or reference will either cause an addressing exception or be the same as if the original pointer or reference had referred a non− *const* object. It is implementation dependent whether the addressing exception occurs.

A pointer to an object of a *volatile* type can be cast into a pointer to a non− *volatile* type. The resulting pointer will refer to the original object. An object of a *volatile* type or a reference to an object of a *volatile* type can be cast into a reference to a non−*volatile* type.

Pointer − to − Member Operators

The pointer−to−member operators *−>** and .* group left−to−right.

pm−expression:
    cast−expression
    pm−expression . cast−expression*
    pm−expression −> cast−expression*

The binary operator .* binds its second operand, which must be of type "pointer to member of class *T"* to its first operand, which must be of class *T* or of a class of which *T* is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.

The binary operator *−>** binds its second operand, which must be of type "pointer to member of *T"* to its first operand, which must be of type "pointer to *T"* or "pointer to a class of which *T* is an unambiguous and accessible base class." The result is an object or a function of the type specified by the second operand.

If the result of .* or *−>** is a function, then that result can be used only as the operand for the function call operator *()*. For example,

```
(ptr_to_obj->*ptr_to_mfct)(10);
```

calls the member function denoted by *ptr_to_mfct* for the object pointed to by *ptr_to_obj*. The result of an" .*" expression or a *"−>*"* expression is an *lvalue* if its second operand is an *lvalue*.

Multiplicative Operators

The multiplicative operators ***, */*, and *%* group left−to−right.

multiplicative−expression:
pm−expression
multiplicative−expression pm−expression*
multiplicative−expression/ pm−expression
multiplicative−expression% pm−expression

The operands of *** and */* must have arithmetic type; the operands of *%* must have integral type. The usual arithmetic conversions (**"Arithmetic Conversions"**) are performed on the operands and determine the type of the result.

The binary *** operator indicates multiplication.

The binary */* operator yields the quotient, and the binary *%* operator yields the remainder from the division of the first expression by the second. If the second operand of */* or *%* is 0 the result is undefined; otherwise *(a/b)*b + a%b* is equal to *a*. If both operands are non−negative then the remainder is non−negative; if not, the sign of the remainder is implementation dependent.

Additive Operators

The "additive operators" *+* and *−* group left−to−right. The usual arithmetic conversions (**"Arithmetic Conversions"**) are performed for operands of arithmetic type.

additive−expression:
multiplicative−expression
additive−expression + multiplicative−expression
additive−expression − multiplicative−expression

The operands must be of arithmetic or pointer type. The result of the *+* operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The result is a pointer of the same type as the original pointer, which points to another object in the same array, appropriately offset from the original object. Thus if *P* is a pointer to an object in an array, the expression *P+1* is a pointer to the next object in the array. If the resulting pointer points outside the bounds of the array, except at the first location beyond the high end of the array, the result is undefined.

The result of the *−* operator is the difference of the operands. A value of any integral type may be subtracted from a pointer, and then the same conversions apply as for addition.

No further type combinations are allowed for pointers.

If two pointers to objects of the same type are subtracted, the result is a signed integral value representing the number of objects separating the pointed−to objects. Pointers to successive elements of an array differ by 1. The type of the result is implementation dependent, but is defined as *ptrdiff_t* in the standard header *<stddef.h>*. The value is undefined unless the pointers point to elements of the same array; however, if *P* points to the last element of an array then *(P+1)−1* is *P*.

Shift Operators

The shift operators *<<* and *>>* group left−to−right.

shift−expression:
  additive−expression
  shift−expression << additive−expression
  shift−expression >> additive−expression

The operands must be of integral type and integral promotions are performed. The type of the result is that of the promoted left operand. The result is undefined if the right operand is negative, or greater than or equal to the length in bits of the promoted left operand. The value of *E1 << E2* is *E1* (interpreted as a bit pattern) left−shifted *E2* bits; vacated bits are 0−filled. The value of *E1 >> E2* is *E1* right−shifted *E2* bit positions. The right shift is guaranteed to be logical (0−fill) if *E1* has an unsigned type or if it has a non−negative value; otherwise, the result is implementation dependent.

Relational Operators

The relational operators group left−to−right, but this fact is not very useful; *a<b<c* means *(a<b)<c* and *not (a<b)&&(b<c)*.

relational−expression:
  shift−expression
  relational−expression < shift−expression
  relational−expression > shift−expression
  relational−expression <= shift−expression
  relational−expression >= shift−expression

The operands must have arithmetic or pointer type. The operators *<* (less than), *>* (greater than), *<=* (less than or equal to), and *>=* (greater than or equal to) all yield if the specified relation is false and if it is true. The type of the result is *int*.

The usual arithmetic conversions are performed on arithmetic operands. Pointer conversions are performed on pointer operands. This implies that any pointer may be compared to a constant expression evaluating to 0 and any pointer can be compared to a pointer of type *void** (in the latter case the pointer is first converted to *void**). Pointers to objects or functions of the same type (after pointer conversions) may be compared; the result depends on the relative positions of the pointed−to objects or functions in the address space.

Two pointers to the same object compare equal. If two pointers point to non−static members of the same object, the pointer to the later declared member compares higher provided the two members are not separated by an *access−specifier* label (**"Access Specifiers"**) and provided their class is not a union. If two pointers point to non−static members of the same object separated by an *access−specifier* label (**"Access Specifiers"**) the result is undefined. If two pointers point to data members of the same union, they compare equal. If two pointers point to elements of the same array or one beyond the end of the array, the pointer to the object with the higher subscript compares higher. Other pointer comparisons are implementation dependent.

Equality Operators

equality−expression:
  relational−expression
  equality−expression == relational−expression
  equality−expression != relational−expression

The *==* (equal to) and the *!=* (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus
a<b == c<d is 1 whenever *a<b* and *c<d* have the same truth−value.)

In addition, pointers to members of the same type may be compared. Pointer to member conversions (**"Pointers to Members"**) are performed. A pointer to member may be compared to a constant expression that evaluates to .

Bitwise AND Operator

and−expression:
    equality−expression
    and−expression & equality−expression

The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral operands.

Bitwise Exclusive−OR Operator

exclusive−or−expression:
    and−expression
    exclusive−or−expression ^ and−expression

The usual arithmetic conversions are performed; the result is the bitwise exclusive−OR function of the operands. The operator applies only to integral operands.

Bitwise Inclusive−OR Operator

inclusive−or−expression:
    exclusive−or−expression
    inclusive−or−expression | exclusive−or−expression

The usual arithmetic conversions are performed; the result is the bitwise inclusive−OR function of its operands. The operator applies only to integral operands.

Logical AND Operator

logical−and−expression:
    inclusive−or−expression
    logical−and−expression && inclusive−or−expression

The *&&* operator groups left−to−right. It returns 1 if both its operands are non−zero, 0 otherwise. Unlike *&*, *&&* guarantees left−to−right evaluation; moreover the second operand is not evaluated if the first operand evaluates to 0.

The operands need not have the same type, but each must have arithmetic type or be a pointer. The result is an *int*. All side effects of the first expression happen before the second expression is evaluated.

Logical OR Operator

logical−or−expression:
    logical−and−expression
    logical−or−expression || logical−and−expression

The *||* operator groups left−to−right. It returns 1 if either of its operands is non−zero, and 0 otherwise. Unlike *|*, *||* guarantees left−to−right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to non−zero.

The operands need not have the same type, but each must have arithmetic type or be a pointer. The result is an *int*. All side effects of the first expression happen before the second expression is evaluated.

Conditional Operator

conditional−expression:
logical−or−expression
logical−or−expression ? expression :conditional−expression

Conditional expressions group right−to−left. The first expression must have arithmetic type or be a pointer type. It is evaluated and if it is non−zero, the result of the conditional expression is the value of the second expression, otherwise that of the third expression. All side effects of the first expression happen before the second or third expression is evaluated.

If both the second and the third expressions are of arithmetic type, then if they are of the same type the result is of that type; otherwise the usual arithmetic conversions are performed to bring them to a common type. Otherwise, if both the second and the third expressions are either a pointer or a constant expression that evaluates to 0, pointer conversions are performed to bring them to a common type. Otherwise, if both the second and the third expressions are references, reference conversions are performed to bring them to a common type. Otherwise, if both the second and the third expressions are *void*, the common type is *void*. Otherwise, if both the second and the third expressions are of the same class *T*, the common type is *T*. Otherwise the expression is illegal. The result has the common type; only one of the second and third expressions is evaluated. The result is an *lvalue* if the second and the third operands are of the same type and both are *lvalues*.

Assignment Operators

There are several assignment operators, all of which group right−to−left. All require a modifiable *lvalue* as their left operand, and the type of an assignment expression is that of its left operand. The result of the assignment operation is the value stored in the left operand after the assignment has taken place; the result is an *lvalue*.

assignment−expression:
conditional−expression
unary−expression assignment−operator assignment−expression

assignment−operator: one of
*= *= /= %= += −= >>= <<= &= ^= |=*

In simple assignment (*=*), the value of the expression replaces that of the object referred to by the left operand. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. There is no implicit conversion to an enumeration (**"Enumeration Declarations"**), so if the left operand is of an enumeration type, the right operand must be of the same type. If the left operand is of pointer type, the right operand must be of pointer type or a constant expression that evaluates to 0; the right operand is converted to the type of the left before the assignment.

A pointer of type *T*const* can be assigned to a pointer of type *T**, but the reverse assignment is illegal (**"Type Specifiers"**). Objects of types *const T* and *volatile T* can be assigned to plain *T lvalue*s and to *lvalues* of type *volatile T*; see also (**"Initializers"**).

If the left operand is of pointer to member type, the right operand must be of pointer to member type or a constant expression that evaluates to 0; the right operand is converted to the type of the left before the assignment.

Assignment to objects of a class (**Chapter 9, "Classes"**) *X* is defined by the function *X::operator=()* (**"Assignment"**). Unless the user defines an *X::operator=()*, the default version is used for assignment (**"Copying Class Objects"**). This implies that an object of a class derived from *X* (directly or indirectly) by unambiguous public derivation (**"Pointer Conversions"**) can be assigned to an *X*.

A pointer to a member of class *B* may be assigned to a pointer to a member of class *D* of the same type provided *D* is derived from *B* (directly or indirectly) by unambiguous public derivation (**"Ambiguities"**).

Assignment to an object of type "reference to *T"* assigns to the object of type *T* denoted by the reference.

The behavior of an expression of the form *E1 op* is equivalent to *E1 = E1* except that *E1* is evaluated only once. In *+=* and *−=*, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in **"Additive Operators"** ; all right operands and all non−pointer left operands must have arithmetic type.

For class objects, assignment is not in general the same as initialization (**"Initializers"** , **"Constructors" in Chapter 12** , **"Initialization"** , **"Copying Class Objects"**).

Comma Operator

The comma operator groups left−to−right.

expression:
  assignment−expression
  expression , assignment−expression

A pair of expressions separated by a comma is evaluated left−to−right and the value of the left expression is discarded. All side effects of the left expression are performed before the evaluation of the right expression. The type and value of the result are the type and value of the right operand; the result is an *lvalue* if its right operand is.

In contexts where comma is given a special meaning, for example, in lists of actual arguments to functions (**"Function Call"**) and lists of initializers (**"Initializers"**), the comma operator as described in this section can appear only in parentheses; for example,

```
f(a, (t=3, t+2), c);
```

has three arguments, the second of which has the value 5.

Constant Expressions

In several places, C++ requires expressions that evaluate to an integral constant: as array bounds (**"Arrays"**), as *case* expressions (**"The switch Statement"**), as bit−field lengths (**"Bit−Fields"**), and as enumerator initializers (**"Enumeration Declarations"**).

constant−expression:
  conditional−expression

A *constant−expression* can involve only literals (**"Literals"**), enumerators, *const* values of integral types initialized with constant expressions (**"Initializers"**), and *sizeof* expressions. Floating constants (**"Floating Constants"**) must be cast to integral types. Only type conversions to integral types may be used. In particular, except in *sizeof* expressions, functions, class objects, pointers, and references cannot be used. The comma operator and *assignment−operators* may not be used in a constant expression.

Chapter 6

Statements

This chapter contains the following subsections:

- **"Labeled Statement"**
- **"Expression Statement"**
- **"Compound Statement, or Block"**
- **"Selection Statements"**
- **"Iteration Statements"**
- **"Jump Statements"**
- **"Declaration Statement"**
- **"Ambiguity Resolution"**

Except as indicated, statements are executed in sequence.

statement:
  labeled−statement
  expression−statement
  compound−statement
  selection−statement
  iteration−statement

jump–statement
declaration–statement

Labeled Statement

A statement may be labeled.

labeled–statement:
identifier : statement
case constant–expression : statement
default : statement

An identifier label declares the identifier. The only use of an identifier label is as the target of a *goto*. The scope of a label is the function in which it appears. Labels cannot be redeclared within a function. A label can be used in a *goto* statement before its definition. Labels have their own name space and do not interfere with other identifiers.

Case labels and default labels may occur only in switch statements.

Expression Statement

Most statements are expression statements, which have the form

expression–statement:
expression$_{opt}$ *;*

Usually expression statements are assignments or function calls. All side effects from an expression statement are completed before the next statement is executed. An expression statement with the expression missing is called a null statement; it is useful to carry a label just before the *}* of a compound statement and to supply a null body to an iteration statement such as *while* (**"The while Statement"**).

Compound Statement, or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided.

compound–statement:
{ statement–list$_{opt}$ *}*

statement–list:
statement
statement–list statement

Note that a declaration is a *statement* (**"Declaration Statement"**).

Selection Statements

Selection statements choose one of several flows of control.

selection–statement:
if (expression) statement
if (expression) statement else statement
switch (expression) statement

The *statement* in a *selection–statement* may not be a *declaration*.

The if Statement

The expression must be of arithmetic or pointer type or of a class type for which an unambiguous conversion to arithmetic or pointer type exists (**"Conversions"**).

The expression is evaluated and if it is non–zero, the first substatement is executed. If *else* is used, the second substatement is executed if the expression is zero. The *else* ambiguity is resolved by connecting an *else* with the last encountered *else*–less *if*.

The switch Statement

The *switch* statement causes control to be transferred to one of several statements depending on the value of an expression.

The expression must be of integral type or of a class type for which an unambiguous conversion to integral type exists (**"Conversions"**). Integral promotion is performed. Any statement within the statement may be labeled with one or more case labels as follows:

```
case constant-expression :
```

where the *constant–expression* (**"Constant Expressions"**) is converted to the promoted type of the switch expression. No two of the case constants in the same switch may have the same value.

There may be at most one label of the form

```
default :
```

within a *switch* statement.

Switch statements may be nested; a *case* or *default* label is associated with the smallest switch enclosing it.

When the *switch* statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case label. If no case constant matches the expression, and if there is a *default* label, control passes to the statement labeled by the default label. If no case matches and if there is no *default* then none of the statements in the switch is executed.

case and *default* labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a switch, see *break*, **"The break Statement"** .

Usually, the statement that is the subject of a switch is compound. Declarations may appear in the *statement* of a switch–statement. It is illegal, however, to jump past a declaration with an explicit or implicit initializer unless the declaration is in an inner block that is not entered (that is, completely by passed by the transfer of control; **"Declaration Statement"**). This implies that declarations that contain explicit or implicit initializers must be contained in an inner block.

Iteration Statements

Iteration statements specify looping.

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

Note that a *for–init–statement* ends with a semicolon.

The *statement* in an *iteration–statement* may not be a *declaration*.

The while Statement

In the *while* statement the substatement is executed repeatedly until the value of the expression becomes zero. The test takes place before each execution of the substatement.

The expression must be of arithmetic or pointer type or of a class type for which an unambiguous conversion to arithmetic or pointer type exists (**"Conversions"**).

Do statement

In the *do* statement the substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the substatement.

The expression must be of arithmetic or pointer type or of a class type for which an unambiguous conversion to arithmetic or pointer type exists (**"Conversions"**).

The for Statement

The *for* statement

for (for−init−statement expression−1$_{opt}$ *;expression−2*$_{opt}$ *) statement*

is equivalent to

for−init−statement

while (expression−1) {
    statement
    expression−2 ;

}

except that a *continue* in *statement* will execute *expression−2* before re−evaluating *expression−1*. Thus the first statement specifies initialization for the loop; the first expression specifies a test, made before each iteration, such that the loop is exited when the expression becomes zero; the second expression often specifies incrementing that is done after each iteration. The first expression must have arithmetic or pointer type or a class type for which an unambiguous conversion to arithmetic or pointer type exists (**"Conversions"**).

Either or both of the expressions may be dropped. A missing *expression−1* makes the implied *while* clause equivalent to *while(1)*.

If the *for−init−statement* is a declaration, the scope of the names declared extends to the end of the block enclosing the *for−statement*.

Jump Statements

Jump statements unconditionally transfer control.

jump−statement:
    break ;
    continue ;
    return expression$_{opt}$ *;*
    goto identifier ;

On exit from a scope (however accomplished), destructors (**"Destructors"**) are called for all constructed class objects in that scope that have not yet been destroyed. This applies to both explicitly declared objects and temporaries (**"Temporary Objects"**).

The break Statement

The *break* statement may occur only in an *iteration−statement* or a *switch* statement and causes termination of the smallest enclosing *iteration−statement* or *switch* statement; control passes to the statement following the terminated statement, if any.

The continue Statement

The *continue* statement may occur only in an *iteration−statement* and causes control to pass to the loop−continuation portion of the smallest enclosing *iteration−statement*, that is, to the end of the loop. More precisely, in each of the statements. For example,

```
while (foo) {     do {          for (;;) {
    // ...          // ...          // ...
```

contin: ;   *contin: ;*   *contin: ;*

}   *} while (foo);*   *}*

a *continue* statement not contained in an enclosed iteration statement is equivalent to *goto contin*.

The return Statement

A function returns to its caller by the *return* statement.

A return statement without an expression can be used only in functions that do not return a value, that is, a function with the return value type *void*, a constructor (**"Constructors"**), or a destructor (**"Destructors"**). A return statement with an expression can be used only in functions returning a value; the value of the expression is returned to the caller of the function. If required, the expression is converted, as in an initialization, to the return type of the function in which it appears. This may involve the construction and copy of a temporary object (**"Temporary Objects"**). Flowing off the end of a function is equivalent to a *return* with no value; this is illegal in a value−returning function.

The goto Statement

The *goto* statement unconditionally transfers control to the statement labeled by the identifier. The identifier must be a label (**"Labeled Statement"**) located in the current function.

Declaration Statement

A declaration statement introduces a new identifier into a block; it has the form

declaration−statement:
    declaration

If an identifier introduced by a declaration was previously declared in an outer block, the outer declaration is hidden for the remainder of the block, after which it resumes its force.

Any initializations of *auto* or *register* variables are done each time their *declaration−statement* is executed. Destruction of local variables declared in the block is done on exit from the block (**"Jump Statements"**). Destruction of *auto* variables defined in a loop is done once per iteration. For example, here the *Index j* is created and destroyed once each time round the *i* loop:

```
for (int i = 0; i<100; i++)
     for (Index j = 0; j<100; j++) {
     // ...
     }
```

Transfer out of a loop, out of a block, or back past an initialized *auto* variable involves the destruction of *auto* variables declared at the point transferred from but not at the point transferred to.

It is possible to transfer into a block, but not in a way that causes initializations not to be done. It is illegal to jump past a declaration with an explicit or implicit initializer unless the declaration is in an inner block that is not entered (that is, completely bypassed by the transfer of control) or unless the jump is from a point where the variable has already been initialized. For example,

```
void f()
{
     // ...
     goto lx;    // error: jump past initializer
     // ...
ly:
     X a = 1;
     // ...
lx:
     goto ly;    // ok, jump implies destructor
     // call for 'a'
}
```

An *auto* variable constructed under a condition is destroyed under that condition and cannot be accessed outside that condition. For example,

```
if (i)
     for (int j = 0; j<100; j++) {
     // ...
     }
if (j!=100)     // error: access outside condition
     // ...
;
```

Initialization of a local object with storage class *static* (**"Storage Class Specifiers"**) is done the first time control passes through its declaration (only). Where a *static* variable is initialized with an expression that is not a *constant−expression*, default initialization to of the appropriate type (**"Initializers"**) happens before its block is first entered.

The destructor for a local *static* object will be executed if and only if the variable was constructed. The destructor must be called either immediately before or as part of the calls of the *atexit()* functions (**"Start and Termination"**). Exactly when is undefined.

Ambiguity Resolution

There is an ambiguity in the grammar involving *expression−statement*s and *declaration*s: An *expression−statement* with a function−style explicit type conversion (**"Explicit Type Conversion"**) as its leftmost subexpression can be indistinguishable from a *declaration* where the first *declarator* starts with a *(*. In those cases the *statement* is a *declaration*.

To disambiguate, the whole *statement* may have to be examined to determine if it is an *expression−statement* or a *declaration*. This disambiguates many examples. For example, assuming *T* is a *simple−type−name* (**"Type Specifiers"**),

```
T(a)->m = 7;        // expression-statement
T(a)++;             // expression-statement
T(a,5)<<c;          // expression-statement
T(*e)(int);         // declaration
T(f)[];             // declaration
T(g) = { 1, 2 };    // declaration
T(*d)(double(3));   // declaration
```

The remaining cases are *declarations.* For example,

```
T(a);          // declaration
T(*b)();       // declaration
T(c)=7;        // declaration
T(d),e,f=3;    // declaration
T(g)(h,2);     // declaration
```

The disambiguation is purely syntactic; that is, the meaning of the names, beyond whether they are *type−name*s or not, is not used in the disambiguation.

A slightly different ambiguity between *expression−statement*s and *declaration*s is resolved by requiring a *type−name* for function declarations within a block (**"Compound Statement, or Block"**). For example,

```
void g()
{
     int f();    // declaration
     int a;      // declaration
     f();        // expression-statement
     a;          // expression-statement
}
```

Chapter 7
Declarations

This chapter contains the following subsections:

- **"Specifiers"**
- **"Enumeration Declarations"**
- **"Asm Declarations"**
- **"Linkage Specifications"**

Declarations specify the interpretation given to each identifier; they do not necessarily reserve storage associated with the identifier (**"Declarations and Definitions"**). Declarations have the form

declaration:
  decl−specifiers$_{opt}$ declarator−list$_{opt}$;
  asm−declaration
  function−definition
  linkage−specification

The declarators in the *declarator−list* (**Chapter 8, "Declarators"**) contain the identifiers being declared. Only in function definitions (**"Function Definitions"**) and function declarations may the *decl−specifiers* be omitted. Only when declaring a class (**Chapter 9, "Classes"**) or enumeration (**"Enumeration Declarations"**), that is, when the *decl−specifier* is a *class−specifier* or *enum−specifier*, may the *declarator−list* be empty. *asm−declaration*s are described in **"Asm Declarations"** , and *linkage−specification*s in **"Linkage Specifications"** . A declaration occurs in a scope (**"Scopes"**); the scope rules are summarized in **"Summary of Scope Rules"** .

Specifiers

The specifiers that can be used in a declaration are

decl−specifier:
  storage−class−specifier
  type−specifier
  fct−specifier
  template−specifier
  friend
  typedef

decl−specifiers:
  decl−specifiers$_{opt}$ decl−specifier

The longest sequence of *decl−specifier*s that could possibly be a type name is taken as the *decl−specifiers* of a *declaration*. The sequence must be self−consistent as described below. For example,

```
typedef char* Pc;
static Pc;              // error: name missing
```

Here, the declaration *static Pc* is illegal because no name was specified for the static variable of type *Pc*. To get a variable of type *int* called *Pc*, the *type−specifier int* must be present to indicate that the *typedef−name Pc* is the name being (re)declared, rather than being part of the *decl−specifier* sequence. For example,

```
void f(const Pc);       // void f(char*const)
void g(const int Pc);   // void g(const int)
```

Note that since *signed*, *unsigned*, *long*, and *short* by default imply *int*, a *typedef−name* appearing after one of those specifiers must be the name being (re)declared. For example,

```
void h(unsigned Pc);        // void h(unsigned int)
void k(unsigned int Pc);    // void k(unsigned int)
```

Storage Class Specifiers

The storage class specifiers are

storage−class−specifier:
  auto
  register
  static
  extern

The *auto* or *register* specifiers can be applied only to names of objects declared in a block (**"Compound Statement, or Block"**) and for formal arguments (**"Function Definitions"**). The *auto* declarator is almost always redundant and not often used; one use of *auto* is to distinguish a *declaration−statement* from an *expression−statement* (**"Expression Statement"**) explicitly.

A *register* declaration is an *auto* declaration, together with a hint to the compiler that the variables declared will be heavily used. The hint may be ignored and in most implementations it will be ignored if the address of the variable is taken.

An object declaration is a definition unless it contains the *extern* specifier and has no initializer (**"Declarations and Definitions"**).

A definition causes the appropriate amount of storage to be reserved and any appropriate initialization (**"Initializers"**) to be done.

The *static* and *extern* specifiers can be applied only to names of objects and functions and to anonymous unions. There can be no *static* function declarations within a block, nor any *static* or *extern* formal arguments. Static class members are described in (**"Static Members"**); *extern* cannot be used for class members.

A name specified *static* has internal linkage. Objects declared *const* have internal linkage unless they have previously been given external linkage. A name specified *extern* has external linkage unless it has previously been given internal linkage. A file scope name without a *storage−class−specifier* has external linkage unless it has previously been given internal linkage and provided it is not declared *const*. For a non−member function an *inline* specifier is equivalent to a *static* specifier for linkage purposes (**"Program and Linkage"**). All linkage specifications for a name must agree. For example,

```
static char* f(); // f() has internal linkage
* f()         // f() still has internal linkage
     { /* ... */ }
char* g();         // g() has external linkage
static char* g()  // error: inconsistent linkage
     { /* ... */ }
static int a;      // 'a' has internal linkage
int a;             // error: two definitions
static int b;      // 'b' has internal linkage
extern int b;      // 'b' still has internal linkage
int c;             // 'c' has external linkage
static int c;      // error: inconsistent linkage
extern int d;      // 'd' has external linkage
static int d;      // error: inconsistent linkage
```

The name of an undefined class can be used in an *extern* declaration. Such a declaration, however, cannot be used before the class has been defined. For example,

```
struct S;
extern S a;
extern S f();
extern void g(S);
void h()
{
     g(a);      // error: S undefined
     f();      // error: S undefined
```

```
}
```

Function Specifiers

Some specifiers can be used only in function declarations.

fct−specifier:
  inline
  virtual

The *inline* specifier is a hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation. The hint may be ignored. For a non−member function *inline* specifier also gives the function default internal linkage (**"Program and Linkage"**). A function (**"Function Call"** , **"Functions"**) defined within the declaration of a class is *inline* by default.

An inline member function must have exactly the same definition in every compilation in which it appears.

A class member function need not be explicitly declared *inline* in the class declaration to be inline. When no *inline* specifier is used, linkage will be external unless an *inline* definition appears before the first call. For example,

```
class X {
public:
     int f();
     inline int g();  // X::g() has internal linkage
     int h();
};
void k(X* p)
{
     int i = p->f();  // now X::f() has external linkage
     int j = p->g();
     // ...
}
inline int X::f()    // error: called before defined as inline
{
     // ...
}
inline int X::g()
{
     // ...
}
inline int X::h()   // now X::h() has internal linkage
{
     // ...
}
```

The *virtual* specifier may be used only in declarations of non−static class member functions within a class declaration; see **"Virtual Functions"** .

The *typedef* Specifier

Declarations containing the *decl−specifier* *typedef* declare identifiers that can be used later for naming fundamental or derived types. The *typedef* specifier may not be used in a *function−definition* (**"Function Definitions"**).

typedef−name:
  identifier

Within the scope (**"Scopes"**) of a *typedef* declaration, each identifier appearing as part of any declarator therein becomes syntactically equivalent to a keyword and names the type associated with the identifier in the way described in **Chapter 8, "Declarators"** . A *typedef−name* is thus a synonym for another type. A *typedef−name* does not introduce a new type the way a class declaration (**"Class Names"**) does. For example, after

```
typedef int MILES, *KLICKSP;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
```

are all legal declarations; the type of *distance* is *int*, that of *metricp* is "pointer to *int*."

A *typedef* may be used to redefine a name to refer to the type to which it already refers even in the scope where the type was originally declared. For example,

```
typedef struct s { /* ... */ } s;
typedef int I;
typedef int I;
typedef I I;
```

An unnamed class defined in a typedef gets its typedef name as its name. For example,

```
typedef struct { /* ... */ } S; // the struct is named S
```

A *typedef* may not redefine a name of a type declared in the same scope to refer to a different type. For example,

```
class complex { /* ... */ };
typedef int complex; // error: redefinition
```

Similarly, a class may not be declared with the name of a type declared in the same scope to refer to a different type. For example,

```
typedef int complex;
class complex { /* ... */ }; // error: redefinition
```

A *typedef−name* that names a class is a *class−name* (**"Class Names"**). The synonym may not be used after a *class*, *struct*, or *union* prefix and not in the names for constructors and destructors within the class declaration itself. For example,

```
struct S {
     S();
     ~S();
};
typedef struct S T;
S a = T();    // ok
struct T * p;    // error
```

The *template* Specifier

The *template* specifier is used to specify families of types or functions; see **Chapter 14, "Templates"** .

The *friend* Specifier

The *friend* specifier is used to specify access to class members; see **"Friends"** .

Type Specifiers

The type−specifiers are

type−specifier:
    simple−type−name
    class−specifier
    enum−specifier
    elaborated−type−specifier
    :: class−name

const
volatile

The words *const* and *volatile* may be added to any legal *type−specifier* in the declaration of an object. Otherwise, at most one *type−specifier* may be given in a declaration. A *const* object may be initialized, but its value may not be changed thereafter. Unless explicitly declared *extern*, a *const* object does not have external linkage and must be initialized (**"Initializers"** ; **"Constructors"**). An integer *const* initialized by a constant expression may be used in constant expressions (**"Constant Expressions"**). Each element of a *const* array is *const* and each non−function, non−static member of a *const* class object is *const* (**"The this Pointer"**). A *const* object of a type that does not have a constructor or a destructor may be placed in read−only memory. The effect of a write operation on any part of such an object is either an addressing exception or the same as if the object had been non− *const*.

There are no implementation−independent semantics for *volatile* objects; *volatile* is a hint to the compiler to avoid aggressive optimization involving the object because the value of the object may be changed by means undetectable by a compiler. Each element of a *volatile* array is *volatile* and each non−function, non−static member of a *volatile* class object is *volatile* (**"The this Pointer"**).

If the *type−specifier* is missing from a declaration, it is taken to be *int*.

simple−type−name:
complete−class−name
qualified−type−name
char
short
int
long
signed
unsigned
float
double
void

At most one of the words *long* or *short* may be specified together with *int*. Either may appear alone, in which case *int* is understood. The word *long* may appear together with *double*. At most one of the words *signed* and *unsigned* may be specified together with *char*, *short*, *int*, or *long*. Either may appear alone, in which case *int* is understood. The *signed* specifier forces *char* objects and bit−fields to be signed; it is redundant with other integral types.

*class−specifier*s and *enum−specifier*s are discussed in **Chapter 9, "Classes"** and **"Enumeration Declarations"** , respectively.

elaborated−type−specifier:
class−key class−name
class−key identifier
enum enum−name

class−key:
class
struct
union

If an *identifier* is specified, the *elaborated−type−specifier* declares it to be a *class−name*; see **"Class Names"** .

If defined, a name declared using the *union* specifier must be defined as a union. If defined, a name declared using the *class* specifier must be defined using the *class* or *struct* specifier. If defined, a name declared using the *struct* specifier must be defined using the *class* or *struct* specifier. Names of nested types (**"Nested Class Declarations"**) can be qualified by the name of their enclosing class:

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

A name qualified by a *class−name* must be a type defined in that class or in a base class of that class. As usual, a name declared in a derived class hides members of that name declared in base classes; see **"Scopes"** .

Enumeration Declarations

An enumeration is a distinct integral type (**"Fundamental Types"**) with named constants. Its name becomes an *enum−name*, that is, a reserved word within its scope.

enum−name:
identifier

enum−specifier:
enum identifier$_{opt}$ *{ enum−list*$_{opt}$ *}*

enum−list:
enumerator
enum−list , enumerator

enumerator:
identifier
identifier = constant−expression

The identifiers in an *enum−list* are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at zero and increase by one as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers without initializers continue the progression from the assigned value. The value of an enumerator must be an *int* or a value that can be promoted to *int* by integral promotion (**"Integral Promotions"**).

The names of enumerators must be distinct from those of ordinary variables and other enumerators in the same scope. The values of the enumerators need not be distinct. An enumerator is considered defined immediately after it and its initializer, if any, has been seen. For example,

```
enum { a, b, c=0 };
enum { d, e, f=e+2 };
```

defines *a*, *c*, and *d* to be 0, *b* and *e* to be 1, and *f* to be 3.

Each enumeration defines an integral type that is different from all other integral types. The type of an enumerator is its enumeration. The value of an enumerator or an object of an enumeration type is converted to an integer by integral promotion (**"Integral Promotions"**). For example,

```
enum color { red, yellow, green=20, blue };
color col = red;
color* cp = &col;
if (*cp == blue) // ...
```

makes *color* an integral type describing various colors, and then declares *col* as an object of that type, and *cp* as a pointer to an object of that type. The possible values of an object of type *color* are *red*, *yellow* , *green*, *blue*; these values can be converted to the *int* values 0, 1, 20, and 21. Since enumerations are distinct types, objects of type *color* may be assigned only values of type *color*. For example,

```
color c = 1;      // error: type mismatch,
// no conversion from int to color
int i = yellow;  // ok: yellow converted to int value 1
// integral promotion
```

See also **"Enumeration Declarations"**

Enumerators defined in a class (**"Class Names"**) are in the scope of that class and can be referred to outside member functions of that class only by explicit qualification with the class name (**"Primary Expressions"**). The name of the enumeration itself is also local to the class (**"Nested Class Declarations"**). For example,

```
class X {
public:
     enum direction { left='l', right='r' };
     int f(int i)
        { return i==left ? 0 : i==right ? 1 : 2; }
};
void g(X* p)
{
     direction d;         // error: 'direction' not in scope
     int i;
     i = p->f(left);      // error: 'left' not in scope
     i = p->f(X::right); // ok
     // ...
}
```

Asm Declarations

An asm declaration has the form

asm−declaration:
      asm (string−literal) ;

The meaning of an *asm* declaration is implementation dependent. Typically it is used to pass information through the compiler to an assembler.

Linkage Specifications

Linkage (**"Program and Linkage"**) between C++ and non−C++ code fragments can be achieved using a *linkage−specification*:

linkage−specification:
    extern string−literal { declaration−listopt }
    extern string−literal   declaration

declaration−list:
    declaration
    declaration−list declaration

The *string−literal* indicates the required linkage. The meaning of the *string−literal* is implementation dependent. Linkage to a function written in the C programming language, *"C"*, and linkage to a C++ function, *"C++"*, must be provided by every implementation. Default linkage is *"C++"*. For example,

```
complex sqrt(complex);     // C++ linkage by default
extern "C" {
     double sqrt(double);  // C linkage
}
```

Linkage specifications nest. A linkage specification does not establish a scope. A *linkage−specification* may occur only in *file* scope (**"Scopes"**). A *linkage−specification* for a class applies to non−member functions and objects declared within it. A *linkage−specification* for a function also applies to functions and objects declared within it. A linkage declaration with a string that is unknown to the implementation is an error.

If a function has more than one *linkage−specification*, they must agree; that is, they must specify the same *string−literal*. A function declaration without a linkage specification may not precede the first linkage specification for that function. A function may be declared without a linkage specification after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.

At most one of a set of overloaded functions (**Chapter 13, "Overloading"**) with a particular name can have C linkage. See **"Linkage Specifications"** .

Linkage can be specified for objects. For example,

```
extern "C" {
      // ...
      _iobuf _iob[_NFILE];
// ...
      int _flsbuf(unsigned,_iobuf*);
      // ...
}
```

Functions and objects may be declared *static* within the *{}* of a linkage specification. The linkage directive is ignored for such a function or object. Otherwise, a function declared in a linkage specification behaves as if it was explicitly declared *extern*. For example,

```
extern "C" double f();
static double f();      // error
```

is an error (**"Storage Class Specifiers"**). An object defined within an

```
extern "C" { /* ... */ }
```

construct is still defined (and not just declared).

Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation and language dependent. Only where the object layout strategies of two language implementations are similar enough can such linkage be achieved.

When the name of a programming language is used to name a style of linkage in the *string−literal* in a *linkage−specification*, it is recommended that the spelling be taken from the document defining that language, for example, *Ada* (not *ADA*) and *FORTRAN* (not *Fortran*).

Chapter 8
Declarators

This chapter conains the following subsections:

- **"Type Names"**
- **"Meaning of Declarators"**
- **"Function Definitions"**
- **"Initializers"**

The *declarator−list* appearing in a declaration is a comma−separated sequence of declarators, each of which may have an initializer.

declarator−list:
  init−declarator
  declarator−list , init−declarator

  init−declarator:
  declarator   initializer$_{opt}$

The two components of a *declaration* are the specifiers (*decl−specifiers*; **"Specifiers"**) and the declarators (*declarator−list*). The specifiers indicate the fundamental type, storage class, or other properties of the objects and functions being declared. The declarators specify the names of these objects and functions and (optionally) modify the type with operators such as *(pointer to) and *()* (function returning). Initial values can also be specified in a declarator; initializers are discussed in **"Initializers"** and **"Initialization"** .

Declarators have the syntax

declarator:
dname
ptr−operator declarator
declarator (argument−declaration−list) cv−qualifier−list$_{opt}$
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

dname:
name
class−name
class−name
typedef−name
qualified−type−name

A *class−name* has special meaning in a declaration of the class of that name and when qualified by that name using the scope resolution operator *::* (**"Constructors"** , **"Destructors"**).

Type Names

To specify type conversions explicitly, and as an argument of *sizeof* or *new*, the name of a type must be specified. This is done with a *type−name*, which is syntactically a declaration for an object or function of that type that omits the name of the object or function.

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)

It is possible to identify uniquely the location in the *abstract−declarator* where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int                 // int i
int *               // int *pi
int *[3]            // int *p[3]
int (*)[3]          // int (*p3i)[3]
int *()             // int *f()
int (*)(double)     // int (*pf)(double)
```

name respectively the types "integer," "pointer to integer," "array of 3 pointers to integers," "pointer to array of 3 integers," "function taking no arguments and returning pointer to integer," and "pointer to function taking a *double* argument and returning an integer."

Ambiguity Resolution

The ambiguity arising from the similarity between a function−style cast and a declaration mentioned in

"Ambiguity Resolution" can also occur in the context of a declaration. In that context, it surfaces as a choice between a function declaration with a redundant set of parentheses around an argument name and an object declaration with a function−style cast as the initializer. Just as for statements, the resolution is to consider any construct that could possibly be a declaration a declaration. A declaration can be explicitly disambiguated by a non−function−style cast or *a =* to indicate initialization. For example,

```
struct S {
     S(int);
};
void foo(double a)
{
     S x(int(a));      // function declaration
     S y((int)a);      // object declaration
     S z = int(a);      // object declaration
}
```

Meaning of Declarators

A list of declarators appears after a (possibly empty) list of *decl−specifiers* (**"Specifiers"**). Each declarator contains exactly one *dname*; it specifies the identifier that is declared. Except for the declarations of some special functions (**"Conversions"** , **"Overloaded Operators"**) a *dname* will be a simple *identifier*. An *auto*, *static*, *extern*, *register*, *friend*, *inline*, *virtual*, or *typedef* specifier applies directly to each *dname* in a *declarator−list*; the type of each *dname* depends on both the *decl−specifiers* (**"Specifiers"**) and its *declarator*.

Thus, a declaration of a particular identifier has the form

```
T D
```

where *T* is a type and *D* is a declarator. In a declaration where *D* is an unadorned identifier the type of this identifier is *T*.

In a declaration where *D* has the form

```
( D1 )
```

the type of *D1* is the same as that of *D*. Parentheses do not alter the type of the embedded *dname*, but they may alter the binding of complex declarators.

Pointers

In a declaration *T D* where *D* has the form

```
* cv−qualifier−listopt D1
```

the type of the contained identifier is "... *cv−qualifier−list* pointer to *T*." The *cv−qualifier*s apply to the pointer and not to the object pointed to.

For example, the declarations

```
const ci = 10, *pc = &ci, *const cpc = pc;
int i, *p, *const cp = &i;
```

declare *ci* , a constant integer; *pc*, a pointer to a constant integer; *cpc*, a constant pointer to a constant integer; *i*, an integer; *p*, a pointer to integer; and *cp*, a constant pointer to integer. The value of *ci*, *cpc*, and *cp* cannot be changed after initialization. The value of *pc* can be changed, and so can the object pointed to by *cp*.

Examples of legal operations are

```
i = ci;
*cp = ci;
pc++;
pc = cpc;
```

```
pc = p;
```

Examples of illegal operations are

```
ci = 1;       // error
ci++;         // error
*pc = 2;      // error
cp = &ci;     // error
cpc++;        // error
p = pc;       // error
```

Each is illegal because it would either change the value of an object declared *const* or allow it to be changed through an unqualified pointer later.

volatile specifiers are handled similarly.

See also **"Assignment Operators"** and **"Initializers"** .

There can be no pointers to references (**"References"**) or pointers to bit−fields (**"Bit−Fields"**).

References

In a declaration *T D* where *D* has the form

```
& cv−qualifier−listopt D1
```

the type of the contained identifier is "... *cv−qualifier−list* reference to *T*." The type *void&* is not permitted. For example,

```
void f(double& a) { a += 3.14; }
// ...
     double d = 0;
     f(d);
```

declares *a* to be a reference argument of *f* so the call *f(d)* will add *3.14* to *d*.

```
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;
```

declares the function *g()* to return a reference to an integer so *g(3)=7* will assign to the fourth element of the array *v*.

```
struct link {
     link* next;
};
link* first;
void h(link*& p)  // 'p' is a reference to pointer
{
     p->next = first;
     first = p;
     p = 0;
}
void k()
{
link* q = new link;
h(q);
}
```

declares *p* to be a reference to a pointer to *link* so *h(q)* will leave *q* with the value 0. See also **"References"** .

There can be no references to references, no references to bit−fields (**"Bit−Fields"**), no arrays of references, and no pointers to references. The declaration of a reference must contain an *initializer* (**"Pointers to Members"**) except when the declaration contains an explicit *extern* specifier (**"Storage Class Specifiers"**), is a class member (**"Class Members"**) declaration within a class declaration, or is the declaration of an argument or a return type (**"Functions"**); see **"Declarations and Definitions"** .

Pointers to Members

In a declaration *T D* where *D* has the form

```
complete-class-name :: *  cv-qualifier-listopt D1
```

the type of the contained identifier is "... *cv–qualifier–list* pointer to member of class *complete–class–name* of type *T*."

For example,

```
class X {
public:
     void f(int);
     int a;
};
int X::* pmi = &X::a;
void (X::* pmf)(int) = &X::f;
```

declares *pmi* and *pmf* to be a pointer to a member of *X* of type *int* and a pointer to a member of *X* of type *void(int)*, respectively. They can be used like this:

```
X obj;
//...
obj.*pmi = 7;   // assign 7 to an integer
     // member of obj
(obj.*pmf)(7);  // call a function member of obj
     // with the argument 7
```

Note that a pointer to member cannot point to a static member of a class (**"Static Members"**). See also **"Pointer – to – Member Operators"** and **"Unary Operators"** .

Arrays

In a declaration *T D* where *D* has the form

```
f7D1 [constant-expressionopt]
```

then the contained identifier has type "... array of *T*." If the *constant–expression* (**Chapter 5, "Constant Expressions"**) is present, it must be of integral type and have a value greater than . The constant expression specifies the number of elements in the array. If the constant expression is *N*, the array has *N* elements numbered to *N–1*.

An array may be constructed from one of the fundamental types (except *void*), from a pointer, from a pointer to member, from a class, from an enumeration, or from another array.

When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions that specify the bounds of the arrays may be omitted only for the first member of the sequence. This elision is useful for function arguments of array types, and when the array is external and the definition, which allocates storage, is given elsewhere. The first *constant–expression* may also be omitted when the declarator is followed by an *initializer–list* (**"Initializers"**). In this case the size is calculated from the number of initial elements supplied (**"Aggregates"**).

The declaration

```
float fa[17], *afp[17];
```

declares an array of *float* numbers and an array of pointers to *float* numbers. The declaration

```
static int x3d[3][5][7];
```

declares a static three−dimensional array of integers, with rank 3x5x7. In complete detail, *x3d* is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions *x3d*, *x3d[i]*, *x3d[i][j]*, *x3d[i][j][k]* may reasonably appear in an expression.

When an identifier of array type appears in an expression, except as the operand of *sizeof* or *&* or used to initialize a reference (**"References"**), it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not modifiable lvalues. Except where it has been declared for a class (**"Subscripting"**), the subscript operator *[]* is interpreted in such a way that *E1[E2]* is identical to **((E1)+(E2))*. Because of the conversion rules that apply to *+*, if *E1* is an array and *E2* an integer, then *E1[E2]* refers to the *E2*−th member of *E1*. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed for multidimensional arrays. If *E* is an *n−dimensional* array of rank *ixjx...xk*, then *E* appearing in an expression is converted to a pointer to an *(n−1)*−dimensional array with rank *jx...xk*. If the *** operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed−to *(n−1)*−dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here *x* is a 3x5 array of integers. When *x* appears in an expression, it is converted to a pointer to (the first of three) five−membered arrays of integers. In the expression *x[i]*, which is equivalent to **(x+i)*, *x* is first converted to a pointer as described; then *x+i* is converted to the type of *x*, which involves multiplying *i* by the length of the object to which the pointer points, namely five integer objects. The results are added and indirection applied to yield an array (of five integers), which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C++ are stored row−wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

Functions

In a declaration *T D* where *D* has the form

```
D1 ( argument-declaration-list ) cv-qualifier-listopt
```

the contained identifier has the type "... *cv−qualifier−listopt* function taking arguments of type *argument−declaration−list* and returning *T*."

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

If the *argument−declaration−list* terminates with an ellipsis, the number of arguments is known only to be equal to or greater than the number of argument types specified; if it is empty, the function takes no arguments. The argument list *(void)* is equivalent to the empty argument list. Except for this special case *void* may not be an argument type (though types derived from *void*, such as *void**, may). Where legal, ", ..." is synonymous with "...". The standard header *<stdarg.h>* contains a mechanism for accessing arguments passed using the ellipsis. See **"Constructors"** for the treatment of array arguments.

A single name may be used for several different functions in a single scope; this is function overloading (**Chapter 13, "Overloading"**). All declarations for a function taking a given set of arguments must agree exactly both in the type of the value returned and in the number and type of arguments; the presence or absence of the ellipsis is considered part of the function type. Argument types that differ only in the use of typedef names or unspecified argument array bounds agree exactly. The return type and the argument types, but not the default arguments (**"Default Arguments"**), are part of the function type. A *cv−qualifier−list* can be part of a declaration or definition of a non−static member function, and of a pointer to a member function; see **"The this Pointer"** . It is part of the function type.

Functions cannot return arrays or functions, although they can return pointers and references to such things. There are no arrays of functions, although there may be arrays of pointers to functions.

Types may not be defined in return or argument types.

The *argument−declaration−list* is used to check and convert actual arguments in calls and to check pointer−to−function and reference−to−function assignments and initializations.

An identifier can optionally be provided as an argument name; if present in a function declaration, it cannot be used since it immediately goes out of scope; if present in a function definition (**"Function Definitions"**), it names a formal argument. In particular, argument names are also optional in function definitions and names used for an argument in different declarations and the definition of a function need not be the same.

The declaration

```
int i,
      *pi,
      f(),
      *fpi(int),
      (*pif)(const char*, const char*);
```

declares an integer *i*, a pointer *pi* to an integer, a function *f* taking no arguments and returning an integer, a function *fpi* taking an integer argument and returning a pointer to an integer, and a pointer *pif* to a function which takes two pointers to constant characters and returns an integer. It is especially useful to compare the last two. The binding of **fpi(int)* is **(fpi(int))*, so the declaration suggests, and the same construction in an expression requires, the calling of a function *fpi*, and then using indirection through the (pointer) result to yield an integer. In the declarator *(*pif)(const char**, *const char*)*, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function, which is then called.

The declaration

```
fseek(FILE*, long, int);
```

declares a function taking three arguments of the specified types. Since no return value type is specified it is taken to be *int* (**"Type Specifiers"**). The declaration

```
printf(const char* ...);
```

declares a function that can be called with varying number and types of arguments. For example,

```
printf("hello world");
printf("a=%d b=%d", a, b);
```

It must always have a value, however, that can be converted to a *const char** as its first argument.

Default Arguments

If an expression is specified in an argument declaration this expression is used as a default argument. All subsequent arguments must have default arguments supplied in this or previous declarations of this function. Default arguments will be used in calls where trailing arguments are missing. A default argument cannot be redefined by a later declaration (not even to the same value). A declaration may add default arguments, however, not given in previous declarations.

The declaration

```
point(int = 3, int = 4);
```

declares a function that can be called with zero, one, or two arguments of type *int*. It may be called in any of these ways:

```
point(1,2);  point(1);  point();
```

The last two calls are equivalent to *point(1,4)* and *point(3,4)*, respectively.

Default argument expressions have their names bound and their types checked at the point of declaration, and are evaluated at each point of call. In the following example, *g* will be called with the value *f(2)*:

```
int a = 1;
int f(int);
int g(int x = f(a)); // default argument: f(::a)
void h() {
     a = 2;
     {
        int a = 3;
     g();        // g(f(::a))
     }
}
```

Local variables may not be used in default argument expressions. For example,

```
void f()
{
     int i;
     extern void g(int x = i);     // error
     // ...
}
```

Note that default arguments are evaluated before entry into a function and that the order of evaluation of function arguments is implementation dependent. Consequently, formal arguments of a function may not be used in default argument expressions. Formal arguments of a function declared before a default argument expression are in scope and may hide global and class member names. For example,

```
int a;
int f(int a, int b = a);    // error: argument 'a'
     // used as default argument
typedef int I;
int g(int I, int b = I(2)); // error: 'int' called
```

Similarly, the declaration of *X::mem1()* in the following example is illegal because no object is supplied for the non−static member *X::a* used as an initializer.

```
class X {
     int a;
     static b;
     mem1(int i = a); // error: non−static member 'a'
        // used as default argument
     mem2(int i = b); // ok
};
```

The declaration of *X::mem2()* is legal, however, since no object is needed to access the static member *X::b*. Classes, objects, and members are described in **Chapter 9, "Classes"** .

A default argument is not part of the type of a function.

```
int f(int = 0);
void h()
{
     int j = f(1);
     int k = f();          // fine, means f(0)
}
```

```
int (*p1)(int) = &f;
int (*p2)() = &f;     // error: type mismatch
```

An overloaded operator (**"Overloaded Operators"**) cannot have default arguments.

Function Definitions

Function definitions have the form

function−definition:
decl−specifiers$_{opt}$ declarator ctor−initializer$_{opt}$ fct−body

fct−body:
compound−statement

The *declarator* in a *function−definition* must contain a declarator with the form

```
D1 ( argument-declaration-list ) cv-qualifier-listopt
```

as described in **"Functions"** .

The formal arguments are in the scope of the outermost block of the *fct−body.*

A simple example of a complete function definition is

```
int max(int a, int b, int c)
{
     int m = (a > b) ? a : b;
     return (m > c) ? m : c;
}
```

Here *int* is the *decl−specifiers; max(int a*, *int b*, *int c*) is the *declarator; "{ /* ... */ }"* is the *fct−body.*

A *ctor−initializer* is used only in a constructor; see **"Constructors"** and **"Initialization"** .

A *cv−qualifier−list* can be part of a non−static member function declaration, non−static member function definition, or pointer to member function only; see **"The this Pointer"** . It is part of the function type.

Note that unused formal arguments need not be named. For example,

```
void print(int a, int)
{
     printf("a = %d\n",a);
}
```

Initializers

A declarator may specify an initial value for the identifier being declared.

initializer:
= assignment−expression
= { initializer−list ,$_{opt}$ }
(expression−list)

initializer−list:
assignment−expression
initializer−list , assignment−expression
{ initializer−list ,$_{opt}$ }

Automatic, register, static, and external variables may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

```
int f(int);
int a = 2;
int b = f(a);
int c(b);
```

A pointer of type *const T**, that is, a pointer to constant *T*, can be initialized with a pointer of type *T**, but the reverse initialization is illegal. Objects of type *T* can be initialized with objects of type *T* independently of *const* and *volatile* modifiers on both the initialized variable and on the initializer. For example,

```
int a;
const int b = a;
int c = b;
const int* p0 = &a;
const int* p1 = &b;
int* p2 = &b;        // error: makes a pointer to
     // non-const point to a const
int *const p3 = p2;
int *const p4 = p1;  // error: makes a pointer to
     // non-const point to a const
const int* p5 = p1;
```

The reason for the two errors is the same: had those initializations been allowed they would have allowed the value of something declared *const* to be changed through an unqualified pointer.

Default argument expressions are more restricted; see **"Default Arguments"** .

Initialization of objects of classes with constructors is described in **"Explicit Initialization"** . Copying of class objects is described in **"Copying Class Objects"** . The order of initialization of static objects is described in **"Start and Termination"** and **"Declaration Statement"** .

Variables with storage class static (**"Storage Classes"**) that are not initialized are guaranteed to start off as converted to the appropriate type. So are members of static class objects. The initial values of automatic and register variables that are not initialized are undefined.

When an initializer applies to a pointer or an object of arithmetic type, it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

Note that since *()* is not an initializer,

```
X a();
```

is not the declaration of an object of class *X*, but the declaration of a function taking no argument and returning an *X*.

An initializer for a static member is in the scope of the member's class. For example,

```
int a;
struct X {
     static int a;
     static int b;
};
int X::a = 1;
int X::b = a;     // X::b = X::a
```

See **"Default Arguments"** for initializers used as default arguments.

Aggregates

An *aggregate* is an array or an object of a class (**Chapter 9, "Classes"**) with no constructors (**"Constructors"**), no private or protected members (**Chapter 11, "Member Access Control"**), no base classes (**Chapter 10, "Derived Classes"**), and no virtual functions (**"Virtual Functions"**). When an aggregate is initialized the *initializer* may be an *initializer−list* consisting of a brace−enclosed, comma−separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the subaggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros of the appropriate types.

For example,

```
struct S { int a; char* b; int c; };
S ss = { 1, "asdf" };
```

initializes *ss.a* with , *ss.b* with *"asdf"*, and *ss.c* with 0.

An aggregate that is a class may also be initialized with an object of its class or of a class publicly derived from it (**"Copying Class Objects"**).

Braces may be elided as follows. If the *initializer−list* begins with a left brace, then the succeeding comma−separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the *initializer−list* or a subaggregate does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes *x* as a one−dimensional array that has three members, since no size was specified and there are three initializers.

```
float y[4][3] = {
     { 1, 3, 5 },
     { 2, 4, 6 },
     { 3, 5, 7 },
};
```

is a completely−bracketed initialization: 1, 3, and 5 initialize the first row of the array *y[0]*, namely *y[0][0]*, *y[0][1]*, and *y[0][2]*. Likewise the next two lines initialize *y[1]* and *y[2]*. The initializer ends early and therefore *y[3]* is initialized with zeros. Precisely the same effect could have been achieved by

```
float y[4][3] = {
     1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The last (rightmost) index varies fastest (**"Arrays"**).

The initializer for *y* begins with a left brace, but the one for *y[0]* does not, therefore three elements from the list are used. Likewise the next three are taken successively for *y[1]* and *y[2]*. Also,

```
float y[4][3] = {
     { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of *y* (regarded as a two−dimensional array) and leaves the rest 0.

Initialization of arrays of objects of a class with constructors is described in **"Explicit Initialization"** .

The initializer for a union with no constructor is either a single expression of the same type, or a brace−enclosed initializer for the first member of the union. For example,

```
union u { int a; char* b; };
u a = { 1 };
u b = a;
u c = 1;              // error
u d = { 0, "asdf" };  // error
u e = { "asdf" };     // error
```

There may not be more initializers than there are members or elements to initialize. For example,

```
char cv[4] = { ’a’, ’s’, ’d’, ’f’, 0 };  // error
```

is an error.

Character Arrays

A *char* array (whether signed or unsigned) may be initialized by a *string−literal;* successive characters of the string initialize the members of the array. For example,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string. Note that because *´\n´* is a single character and because a trailing *´\0´* is appended, *sizeof(msg)* is 25.

There may not be more initializers than there are array elements. For example,

```
char cv[4] = "asdf";  // error
```

is an error since there is no space for the implied trailing *´\0´*

References

A variable declared to be a *T&*, that is "reference to type *T"* (**"References"**), must be initialized by an object of type *T* or by an object that can be converted into a *T*. For example,

```
void f()
{
     int i;
     int& r = i;  // ‘r’ refers to ‘i’
     r = 1;       // the value of ‘i’ becomes 1
     int* p = &r; // ‘p’ points to ‘i’
     int& rr = r; // ‘rr’ refers to what ‘r’ refers to,
     // that is, to ‘i’
}
```

A reference cannot be changed to refer to another object after initialization. Note that initialization of a reference is treated very differently from assignment to it. Argument passing (**"Function Call"**) and function value return (**"The return Statement"**) are initializations.

The initializer may be omitted for a reference only in an argument declaration (**"Functions"**), in the declaration of a function return type, in the declaration of a class member within its class declaration (**"Class Members"**), and where the *extern* specifier is explicitly used. For example,

```
int& r1;          // error: initializer missing
extern int& r2;   // ok
```

If the initializer for a reference to type *T* is an *lvalue* of type *T* or of a type derived (**Chapter 10, "Derived Classes"**) from *T* for which *T* is an accessible base (**"Pointer Conversions"**), the reference will refer to the initializer; otherwise, if and only if the reference is to a *const* an object of type *T* will be created and initialized with the initializer. The reference then becomes a name for that object. For example,

```
double d = 1.0;
double& rd = d;          // rd refers to ’d’
const double& rcd = d;   // rcd refers to ’d’
double& rd2 = 1;         // error: type mismatch
const double& rcd2 = 1; // rcd2 refers to temporary
     // with value ’1’
```

A reference to a *volatile T* can be initialized with a *volatile T* or a plain *T* but not a *const T*. A reference to a *const T* can be initialized with a *const T* or a plain *T* or something that can be converted into a plain *T* but not a *volatile T*. A reference to a plain *T* can be initialized only with a plain *T*.

The lifetime of a temporary object created in this way is the scope in which it is created (**"Storage Classes"**). Note that a reference to a class *B* can be initialized by an object of a class *D* provided *B* is an accessible and unambiguous base class of *D* (in that case a *D* is a *B*); see **"Reference Conversions"** .

Classes

This chapter contains the following subsections:

- **"Class Names"**
- **"Class Members"**
- **"Member Functions"**
- **"Static Members"**
- **"Unions"**
- **"Bit−Fields"**
- **"Nested Class Declarations"**
- **"Local Class Declarations"**
- **"Local Type Names"**

A class is a type. Its name becomes a *class−name* (**"Class Names"**), that is, a reserved word within its scope.

class−name:
  identifier

Class−specifiers and *elaborated−type−specifiers* (**"Type Specifiers"**) are used to make *class−names.* An object of a class consists of a (possibly empty) sequence of members.

class−specifier:
  class−head { member−list$_{opt}$ *}*

class−head:
  class−key identifier$_{opt}$ *base−spec*$_{opt}$
  class−key class−name base−spec$_{opt}$

class−key:
  class
  struct
  union

The name of a class can be used as a *class−name* even within the *member−list* of the class specifier itself. A *class−specifier* is commonly referred to as a class declaration. A class is considered defined when its *class−specifier* has been seen even though its member functions are in general not yet defined.

Objects of an empty class have a non−zero size.

Class objects may be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying has been restricted; see **"Copying Class Objects"**). Other plausible operators, such as equality comparison, can be defined by the user; see **"Overloaded Operators"** .

A structure is a class declared with the *class−key struct*; its members and base classes (**Chapter 10, "Derived Classes"**) are public by default (**Chapter 11, "Member Access Control"**). A union is a class declared with the *class−key union*; its members are public by default and it holds only one member at a time (**"Unions"**).

Class Names

A class declaration introduces a new type. For example,

```
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

declares three variables of three different types. This implies that

```
a1 = a2;      // error: Y assigned to X
a1 = a3;      // error: int assigned to X
```

are type mismatches, and that

```
int f(X);
int f(Y);
```

declare an overloaded (**Chapter 13, "Overloading"**) function *f()* and not simply a single function *f()* twice. For the same reason,

```
struct S { int a; };
struct S { int a; };  // error, double definition
```

is an error because it defines *S* twice.

A class declaration introduces the class name into the scope where it is declared and hides any class, object, function, or other declaration of that name in an enclosing scope (**"Scopes"**). If a class name is declared in a scope where an object, function, or enumerator of the same name is also declared the class can be referred to only using an *elaborated−type−specifier* (**"Type Specifiers"**). For example,

```
struct stat {
     // ...
};
stat gstat;              // use plain 'stat' to
     // define variable
int stat(struct stat*); // redefine 'stat' as function
void f()
{
     struct stat* ps;     // 'struct' prefix needed
     // to name struct stat
     // ...
     stat(ps);            // call stat()
     // ...
}
```

An *elaborated−type−specifier* with a *class−key* used without declaring an object or function introduces a class name exactly like a class declaration but without defining a class. For example,

```
struct s { int a; };
void g()
{
     struct s;  // hide global struct 's'
     s* p;      // refer to local struct 's'
     struct s { char* p; };  // declare local struct 's'
}
```

Such declarations allow declaration of classes that refer to each other. For example,

```
class vector;
class matrix {
     // ...
     friend vector operator*(matrix&, vector&);
};
class vector {
     // ...
     friend vector operator*(matrix&, vector&);
};
```

Declaration of *friends* is described in **"Friends"** , operator functions in **"Overloaded Operators"** . If a class mentioned as a *friend* has not been declared its name is entered in the same scope as the name of the class containing the friend declaration (**"Friends"**).

An *elaborated−type−specifier* (**"Type Specifiers"**) can also be used in the declarations of objects and functions. It differs from a class declaration in that if a class of the elaborated name is in scope the elaborated name will refer to it. For example,

```
struct s { int a; };
```

```
void g()
{
     struct s* p = new s;     // refer to global 's'
     p->a = 1;
}
```

A name declaration takes effect immediately after the *identifier* is seen. For example,

```
class A * A;
```

first specifies *A* to be the name of a class and then redefines it as the name of a pointer to an object of that class. This means that the elaborated form *class A* must be used to refer to the class. Such artistry with names can be confusing and is best avoided.

A *typedef−name* (**"The typedef Specifier"**) that names a class is a *class−name;* see also **"Type Specifiers"**

Class Members

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−specifieropt
  identifieropt : constant−expression

pure−specifier:
  = 0

A *member−list* may declare data, functions, classes, enumerations (**"Enumeration Declarations"**), bit−fields (**"Bit−Fields"**), friends (**"Friends"**), and type names (**"The typedef Specifier"** , **"Class Names"**). A *member−list* may also contain declarations adjusting the access to member names; see **"Access Declarations"** . A member may not be declared twice in the *member−list.* The *member−list* defines the full set of members of the class. No member can be added elsewhere.

Note that a single name can denote several function members provided their types are sufficiently different (**Chapter 13, "Overloading"**). Note that a *member−declarator* cannot contain an *initializer* (**"Initializers"**). A member can be initialized using a constructor; see **"Constructors"** .

A member may not be *auto*, *extern*, or *register*.

The *decl−specifiers* can be omitted in function declarations only. The *member−declarator−list* can be omitted only after a *class−specifier,* an *enum−specifier,* or *decl−specifiers* of the form *friend elaborated−type−specifier.* A *pure−specifier* may be used only in the declaration of a virtual function (**"Virtual Functions"**).

Members that are class objects must be objects of previously declared classes. In particular, a class *cl* may not contain an object of class *cl*, but it may contain a pointer or reference to an object of class *cl*. When an array is used as the type of a non−static member all dimensions must be specified.

A simple example of a class declaration is

```
struct tnode {
     char tword[20];
     int count;
```

```
      tnode *left;
      tnode *right;
};
```

which contains an array of twenty characters, an integer, and two pointers to similar structures.

Once this class declaration has been given, the declaration

```
tnode s, *sp;
```

declares *s* to be a *tnode* and *sp* to be a pointer to a *tnode*. With these declarations, *sp−>count* refers to the *count* member of the structure to which *sp* points; *s.left* refers to the *left* subtree pointer of the structure *s*; and *s.right−>tword[0]* refers to the initial character of the *tword* member of the *right* subtree of *s*.

Non−static data members of a class declared without an intervening *access−specifier* are allocated so that later members have higher addresses within a class object. The order of allocation of non−static data members separated by an *access−specifier* is implementation dependent (**"Access Specifiers"**). Implementation alignment requirements may cause two adjacent members not to be allocated immediately after each other; so may requirements for space for managing virtual functions (**"Virtual Functions"**) and virtual base classes (**"Multiple Base Classes"**); see also **"Explicit Type Conversion"** .

A function member (**"Member Functions"**) with the same name as its class is a constructor (**"Constructors"**). A static data member, enumerator, member of an anonymous union, or nested type may not have the same name as its class.

Member Functions

A function declared as a member (without the *friend* specifier; **"Friends"**) is called a member function, and is called using the class member syntax (**"Class Member Access"**). For example,

```
struct tnode {
      char tword[20];
      int count;
      tnode *left;
      tnode *right;
      void set(char*, tnode* l, tnode* r);
};
```

Here *set* is a member function and can be called like this:

```
void f(tnode n1, tnode n2)
{
      n1.set("abc",&n2,0);
      n2.set("def",0,0);
}
```

The definition of a member function is considered to be within the scope of its class. This means that (provided it is non−static **"Static Members"**) it can use names of members of its class directly. A static member function can use only the names of static members, enumerators, and nested types directly. If the definition of a member function is lexically outside the class declaration, the member function name must be qualified by the class name using the *::* operator. For example,

```
void tnode::set(char* w, tnode* l, tnode* r)
{
      count = strlen(w)+1;
      if (sizeof(tword)<=count)
         error("tnode string too long");
      strcpy(tword,w);
      left = l;
      right = r;
}
```

The notation *tnode::set* specifies that the function *set* is a member of and in the scope of class *tnode*. The member names *tword*, *count*, *left*, and *right* refer to members of the object for which the function was called. Thus, in the call *n1.set*(abc",&n2,0)" *tword* refers to *n1.tword*, and in the call *n2.set*(def",0,0)" it refers to *n2.tword*. The functions *strlen*, *error*, and *strcpy* must be declared elsewhere.

Members may be defined (**"Declarations and Definitions"**) outside their class declaration if they have already been declared but not defined in the class declaration; they may not be redeclared. See also **"Program and Linkage"** . Function members may be mentioned in friend declarations after their class has been defined. Each member function that is called must have exactly one definition in a program.

The effect of calling a non−static member function (**"Static Members"**) of a class *X* for something that is not an object of class *X* is undefined.

The this Pointer

In a non−static (**"Member Functions"**) member function, the keyword *this* is a pointer to the object for which the function is called. The type of *this* in a member function of a class *X* is *X *const* unless the member function is declared *const* or *volatile*; in those cases, the type of *this* is *const X *const* and *volatile X *const*, respectively. A function declared *const* and *volatile* has a *this* with the type *const volatile X *const*. See also **"Assignment to this" in Appendix B** . For example,

```
struct s {
      int a;
      int f() const;
      int g() { return a++; }
      int h() const { return a++; } // error
};
int s::f() const { return a; }
```

The *a++* in the body of *s::h* is an error because it tries to modify (a part of) the object for which *s::h()* is called. This is not allowed in a *const* member function where *this* is a pointer to *const*, that is, **this* is a *const*.

A *const* member function (that is, a member function declared with the *const* qualifier) may be called for *const* and non−*const* objects, whereas a non−*const* member function may be called only for a non− *const* object. For example,

```
void k(s& x, const s& y)
{
      x.f();
      x.g();
      y.f();
      y.g();      // error
}
```

The call *y.g()* is an error because *y* is *const* and *s::g()* is a non−*const* member function that could (and does) modify the object for which it was called.

Similarly, only *volatile* member functions (that is, a member function declared with the *volatile* specifier) may be invoked for *volatile* objects. A member function can be both *const* and *volatile*.

Constructors (**"Constructors"**) and destructors (**"Destructors"**) may be invoked for a *const* or *volatile* object. Constructors (**"Constructors"**) and destructors (**"Destructors"**) cannot be declared *const* or *volatile*.

Inline Member Functions

A member function may be defined (**"Function Definitions"**) in the class declaration, in which case it is *inline* (**"Function Specifiers"**). Defining a function within a class declaration is equivalent to declaring it *inline* and defining it immediately after the class declaration; this rewriting is considered to be done after preprocessing but before syntax analysis and type checking of the function definition. Thus

```
int b;
```

```
int b;
struct x {
     char* f() { return b; }
     char* b;
};
```

is equivalent to

```
int b;
struct x {
     char* f();
     char* b;
};
inline char* x::f() { return b; } // moved
```

Thus the *b* used in *x::f()* is *X::b* and not the global *b*.

Member functions can be defined even in local or nested class declarations where this rewriting would be syntactically illegal. See **"Local Class Declarations"** for a discussion of local classes and **"Nested Class Declarations"** for a discussion of nested classes.

Static Members

A data or function member of a class may be declared *static* in the class declaration. There is only one copy of a static data member, shared by all objects of the class in a program. A static member is not part of objects of a class. Static members of a global class have external linkage (**"Program and Linkage"**). The declaration of a static data member in its class declaration is *not* a definition. A definition is required elsewhere; see also **"Anachronisms"** .

A static member function does not have a *this* pointer so it can access non−static members of its class only by using . or −>. A static member function cannot be *virtual*. There cannot be a static and a non−static member function with the same name and the same argument types.

Static members of a local class (**"Local Class Declarations"**) have no linkage and cannot be defined outside the class declaration. It follows that a local class cannot have static data members.

A static member *mem* of class *cl* can be referred to as *cl::mem* (**"Primary Expressions" in Chapter 5**), that is, independently of any object. It can also be referred to using the . and −> member access operators (**"Class Member Access"**). When a static member is accessed through a member access operator, the expression on the left side of the . or −> is not evaluated. The static member *mem* exists even if no objects of class *cl* have been created. For example, in the following, *run_chain*, *idle*, and so on exist even if no *process* objects have been created:

```
class process {
     static int no_of_processes;
     static process* run_chain;
     static process* running;
     static process* idle;
     // ...
public:
     // ...
     int state();
     static void reschedule();
     // ...
};
```

and *reschedule* can be used without reference to a *process* object, as follows:

```
void f()
{
     process::reschedule();
}
```

Static members of a global class are initialized exactly like global objects and only in file scope. For example,

```
void process::reschedule() { /* ... */ };
int process::no_of_processes = 1;
process* process::running = get_main();
process* process::run_chain = process::running;
```

Static members obey the usual class member access rules (**Chapter 11, "Member Access Control"**) except that they can be initialized (in file scope).

The type of a static member does not involve its class name; thus the type of *process :: no_of_processes* is *int* and the type of *&process :: reschedule* is *void(*)()*.

Unions

A union may be thought of as a structure whose member objects all begin at offset zero and whose size is sufficient to contain any of its member objects. At most one of the member objects can be stored in a union at any time. A union may have member functions (including constructors and destructors), but not virtual (**"Virtual Functions"**) functions. A union may not have base classes. A union may not be used as a base class. An object of a class with a constructor or a destructor or a user−defined assignment operator (**"Assignment"**) cannot be a member of a union. A union can have no *static* data members.

A union of the form

```
union { member-list } ;
```

is called an anonymous union; it defines an unnamed object (and not a type). The names of the members of an anonymous union must be distinct from other names in the scope in which the union is declared; they are used directly in that scope without the usual member access syntax (**"Class Member Access"**).

For example,

```
void f()
{
     union { int a; char* p; };
     a = 1;
     // ...
     p = "Jennifer";
     // ...
}
```

Here *a* and *p* are used like ordinary (non−member) variables, but since they are union members they have the same address.

A global anonymous union must be declared *static*. An anonymous union may not have *private* or *protected* members (**Chapter 11, "Member Access Control"**). An anonymous union may not have function members.

A union for which objects or pointers are declared is not an anonymous union. For example,

```
union { int aa; char* p; } obj, *ptr = &obj;
aa = 1;        // error
ptr->aa = 1;  // ok
```

The assignment to plain *aa* is illegal since the member name is not associated with any particular object.

Initialization of unions that do not have constructors is described in **"Aggregates"** .

Bit−Fields

A *member−declarator* of the form

identifieropt   : constant-expression

specifies a bit–field; its length is set off from the bit–field name by a colon. Allocation of bit–fields within a class object is implementation dependent. Fields are packed into some addressable allocation unit. Fields straddle allocation units on some machines and not on others. Alignment of bit–fields is implementation dependent. Fields are assigned right–to–left on some machines, left–to–right on others.

An unnamed bit–field is useful for padding to conform to externally–imposed layouts. As a special case, an unnamed bit–field with a width of zero specifies alignment of the next bit–field at an allocation unit boundary.

An unnamed field is not a member and cannot be initialized.

A bit–field must have integral type (**"Fundamental Types"**). It is implementation dependent whether a plain (neither explicitly signed nor unsigned) *int* field is signed or unsigned. The "address–of operator" *&* may not be applied to a bit–field, so there are no pointers to bit–fields. Nor are there references to bit–fields.

Nested Class Declarations

A class may be declared within another class. A class declared within another is called a *nested* class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class. Except by using explicit pointers, references, and object names, declarations in a nested class can use only type names, static members, and enumerators from the enclosing class.

```
int x;
int y;
class enclose {
public:
     int x;
     static int s;
     class inner {
        void f(int i)
        {
              x = i;   // error: assign to enclose::x
              s = i;   // ok: assign to enclose::s
              ::x = i; // ok: assign to global x
              y = i;   // ok: assign to global y
        }
        void g(enclose* p, int i)
        {
              p->x = i;   // ok: assign to enclose::x
        }
     };
};
inner* p = 0;     // error 'inner' not in scope
```

Member functions of a nested class have no special access to members of an enclosing class; they obey the usual access rules (**Chapter 11, "Member Access Control"**). Member functions of an enclosing class have no special access to members of a nested class; they obey the usual access rules. For example,

```
class E {
     int x;
     class I {
        int y;
        void f(E* p, int i)
        {
           p->x = i;   // error: E::x is private
        }
     };
```

```
     };
     int g(I* p)
     {
        return p->y;    // error: I::y is private
     }
};
```

Member functions and static data members of a nested class can be defined in the global scope. For example,

```
class enclose {
     class inner {
        static int x;
        void f(int i);
     };
};
typedef enclose::inner ei;
int ei::x = 1;
void enclose::inner::f(int i) { /* ... */ }
```

Like a member function, a friend function defined within a class is in the lexical scope of that class; it obeys the same rules for name binding as the member functions (described above and in **"Summary of Scope Rules"**) and like them has no special access rights to members of an enclosing class or local variables of an enclosing function (**Chapter 11, "Member Access Control"**).

Local Class Declarations

A class can be declared within a function definition; such a class is called a *local* class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope. Declarations in a local class can use only type names, static variables, *extern* variables and functions, and enumerators from the enclosing scope. For example,

```
int x;
void f()
{
     static int s ;
     int x;
     extern int g();
     struct local {
        int h() { return x; }    // error: 'x' is auto
        int j() { return s; }    // ok
        int k() { return ::x; }  // ok
        int l() { return g(); }  // ok
     };
     // ...
}
local* p = 0;     // error: 'local' not in scope
```

An enclosing function has no special access to members of the local class; it obeys the usual access rules (**Chapter 11, "Member Access Control"**). Member functions of a local class must be defined within their class definition. A local class may not have static data members.

Local Type Names

Type names obey exactly the same scope rules as other names. In particular, type names defined within a class declaration cannot be used outside their class without qualification. For example,

```
class X {
public:
     typedef int I;
```

```
     typedef int I;
     class Y { /* ... */ };
     I a;
};
I b;     // error
Y c;     // error
X::Y d;  // ok
```

The following rule limits the context sensitivity of the rewrite rules for inline functions and for class member declarations in general. A *class−name* or a *typedef−name* or the name of a constant used in a type name may not be redefined in a class declaration after being used in the class declaration, nor may a name that is not a *class−name* or a *typedef−name* be redefined to a *class−name* or a *typedef−name* in a class declaration after being used in the class declaration. For example,

```
typedef int c;
enum { i = 1 };
class X {
     char v[i];
     int f() { return sizeof(c); }
     char c;         // error: typedef name
        // redefined after use
     enum { i = 2 }; // error: 'i' redefined after
        // use in type name 'char[i]'
};
typedef char* T;
struct Y {
     T a;
     typedef long T; // error: T already used
     T b;
};
```

Chapter 10
Derived Classes

This Chapter contains the following subsections:

- **"Multiple Base Classes"**
- **"Virtual Functions"**
- **"Abstract Classes"**
- **"Summary of Scope Rules"**

A list of base classes may be specified in a class declaration using the notation:

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

The *class−name* in a *base−specifier* must denote a previously declared class (**"Class Names"**), which is called a base class for the class being declared. A class is said to be derived from its base classes. For the meaning of *access−specifier* see **"Access Specifiers"** . Unless redefined in the derived class, members of a base class can be referred to as if they were members of the derived class. The base class members are said to be *inherited* by the derived class. The scope resolution operator *::* (**"Primary Expressions"**) may be used to refer to a base member explicitly. This allows access to a name that has been redefined in the derived class.

A derived class can itself serve as a base class subject to access control; see **"Access Specifiers for Base Classes"** . A pointer to a derived class may be implicitly converted to a pointer to an accessible unambiguous base class (**"Pointer Conversions"**). A reference to a derived class may be implicitly converted to a reference to an accessible unambiguous base class (**"Reference Conversions"**).

For example,

```
class base {
public:
     int a, b;
};
class derived : public base {
public:
     int b, c;
};
void f()
{
     derived d;
     d.a = 1;
     d.base::b = 2;
     d.b = 3;
     d.c = 4;
     base* bp = &d;  // standard conversion:
        // derived* to base*
}
```

assigns to the four members of *d* and makes *bp* point to *d*.

A class is called a *direct base* if it is mentioned in the *base−list* and an *indirect base* if it is not a direct base but is a base class of one of the classes mentioned in the *base−list.*

Note that in the *class−name :: name* notation, *name* may be a name of a member of an indirect base class; the notation simply specifies a class in which to start looking for *name.* For example,

```
class A { public: void f(); };
class B : public A { };
class C : public B { public: void f(); };
void C::f()
{
     f();    // Call C’s f()
     A::f(); // call A’s f()
     B::f(); // call A’s f()
}
```

Here, *A::f()* is called twice since it is the only *f()* in *B*.

Initialization of objects representing base classes can be specified in constructors; see **"Initializing Bases and Members"** .

Multiple Base Classes

A class may be derived from any number of base classes. For example,

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class D : public A, public B, public C { /* ... */ };
```

The use of more than one direct base class is often called multiple inheritance.

The order of derivation is not significant except possibly for default initialization by constructor (**"Constructors"**), for cleanup (**"Destructors"**), and for storage layout (**"Explicit Type Conversion"** , **"Class Members"** , **"Access Specifiers"**). The order in which storage is allocated for base classes is implementation dependent.

A class may not be specified as a direct base class of a derived class more than once but it may be an indirect base class more than once.

```
class B { /* ... */ };
class D : public B, public B { /* ... */ };  // illegal

class L { /* ... */ };
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { /* ... */ };   // legal
```

Here, an object of class *C* will have two sub−objects of class *L*.

The keyword *virtual* may be added to a base class specifier. A single sub−object of the virtual base class is shared by every base class that specified the base class to be virtual. For example,

```
class V { /* ... */ };
class A : virtual public V { /* ... */ };
class B : virtual public V { /* ... */ };
class C : public A, public B { /* ... */ };
```

Here class *C* has only one sub−object of class *V*.

A class may have both virtual and non−virtual base classes of a given type.

```
class B { /* ... */ };
class X : virtual public B { /* ... */ };
class Y : virtual public B { /* ... */ };
class Z : public B { /* ... */ };
class AA : public X, public Y, public Z { /* ... */ };
```

Here class *AA* has two sub−objects of class *B*: *Z*'s *B* and the virtual *B* shared by *X* and *Y*.

Ambiguities

Access to base class members must be unambiguous. Access to a base class member is ambiguous if the expression used refers to more than one function, object, type, or enumerator. The check for ambiguity takes place before access control (**Chapter 11, "Member Access Control"**). For example,

```
class A {
public:
     int a;
     int (*b)();
     int f();
     int f(int);
     int g();
};
class B {
     int a;
     int b();
public:
     int f();
     int g;
     int h();
     int h(int);
};
```

```
class C : public A, public B {};
void g(C* pc)
{
     pc->a = 1;  // error: ambiguous: A::a or B::a
     pc->b();    // error: ambiguous: A::b or B::b
     pc->f();    // error: ambiguous: A::f or B::f
     pc->f(1);   // error: ambiguous: A::f or B::f
     pc->g();    // error: ambiguous: A::g or B::g
     pc->g = 1;  // error: ambiguous: A::g or B::g
     pc->h();    // ok
     pc->h(1);   // ok
}
```

If the name of an overloaded function is unambiguously found overloading resolution also takes place before access control. Ambiguities can be resolved by qualifying a name with its class name. For example,

```
class A {
public:
     int f();
};
class B {
public:
     int f();
};
class C : public A, public B {
     int f() { return A::f() + B::f(); }
};
```

When virtual base classes are used, a single function, object, type, or enumerator may be reached through more than one path through the directed acyclic graph of base classes. This is not an ambiguity. The identical use with non−virtual base classes is an ambiguity; in that case more than one sub−object is involved. For example,

```
class V { public: int v; };
class A { public: int a; };
class B : public A, public virtual V {};
class C : public A, public virtual V {};
class D : public B, public C { public: void f(); };
void D::f()
{
     v++;      // ok: only one 'v' in 'D'
     a++;      // error, ambiguous: two 'a's in 'D'
}
```

When virtual base classes are used, more than one function, object, or enumerator may be reached through paths through the directed acyclic graph of base classes. This is an ambiguity unless one of the names found *dominates* the others. The identical use with non−virtual base classes is an ambiguity; in that case more than one sub−object is involved.

A name *B::f* *dominates* a name *A::f* if its class *B* has *A* as a base. If a name dominates another no ambiguity exists between the two; the dominant name is used when there is a choice. For example,

```
class V { public: int f(); int x; };
class B : public virtual V { public: int f(); int x; };
class C : public virtual V { };

class D : public B, public C { void g(); };

void D::g()
{
     x++;      // ok: B::x dominates V::x
```

```
     f();     // ok: B::f() dominates V::f()
}
```

An explicit or implicit conversion from a pointer or reference to a derived class to a pointer or reference to one of its base classes must unambiguously refer to the same object representing the base class. For example,

```
class V { };
class A { };
class B : public A, public virtual V { };
class C : public A, public virtual V { };
class D : public B, public C { };
void g()
{
     D d;
     B* pb = &d;
     A* pa = &d;  // error, ambiguous: C's A or B's A ?
     V* pv = &d;  // fine: only one V sub-object
}
```

Virtual Functions

If a class *base* contains a *virtual* (**"Function Specifiers"**) function *vf*, and a class *derived* derived from it also contains a function *vf* of the same type, then a call of *vf* for an object of class *derived* invokes *derived::vf* (even if the access is through a pointer or reference to *base*). The derived class function is said to *override* the base class function. If the function types (**"Functions"**) are different, however, the functions are considered different and the *virtual* mechanism is not invoked (see also **"Declaration Matching"**). It is an error for a derived class function to differ from a base class' virtual function in the return type only. For example,

```
struct base {
     virtual void vf1();
     virtual void vf2();
     virtual void vf3();
     void f();
};
class derived : public base {
public:
     void vf1();
     void vf2(int);     // hides base::vf2()
     char vf3();  // error: differs in return type only
     void f();
};
void g()
{
     derived d;
     base* bp = &d;  // standard conversion:
        // derived* to base*
     bp->vf1();      // calls derived::vf1
     bp->vf2();      // calls base::vf2
     bp->f();        // calls base::f
}
```

The calls invoke *derived::vf1*, *base::vf2*, and *base::f*, respectively, for the class *derived* object named *d*. That is, the interpretation of the call of a virtual function depends on the type of the object for which it is called, whereas the interpretation of a call of a non-virtual member function depends only on the type of the pointer or reference denoting that object. For example, *bp->vf1()* calls *derived::vf1()* because *bp* points to an object of class *derived* in which *derived::vf1()* has overridden the virtual function *base::vf1()*.

The *virtual* specifier implies membership, so a *virtual* function cannot be a global (non−member) (**"Function Specifiers"**) function. Nor can a virtual function be a *static* member, since a virtual function call relies on a specific object for determining which function to invoke. A virtual function can be declared a *friend* in another class. An overriding function is itself considered virtual. The *virtual* specifier may be used for an overriding function in the derived class, but such use is redundant. A virtual function in a base class must be defined or declared pure (**"Abstract Classes"**). A virtual function that has been defined in a base class need not be defined in a derived class. If it is not, the function defined for the base class is used in all calls.

Explicit qualification with the scope operator (**"Primary Expressions"**) suppresses the virtual call mechanism. For example,

```
class B { public: virtual void f(); };
class D : public B { public: void f(); };
void D::f() { /* ... */ B::f(); }
```

Here, the call of *f* in *D* really does call *B::f* and not *D::f*.

Abstract Classes

The abstract class mechanism supports the notion of a general concept, such as a *shape*, of which only more concrete variants, such as *circle* and *square*, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations.

An *abstract class* is a class that can be used only as a base class of some other class; no objects of an abstract class may be created except as objects representing a base class of a class derived from it. A class is abstract if it has at least one *pure virtual function.* A virtual function is specified *pure* by using a *pure−specifier* (**"Class Members"**) in the function declaration in the class declaration.

A pure virtual function need be defined only if explicitly called with the *qualified−name* syntax (**"Primary Expressions"**). For example,

```
class point { /* ... */ };
class shape {           // abstract class
     point center;
// ...
public:
     point where() { return center; }
     void move(point p) { center=p; draw(); }
     virtual void rotate(int) = 0;  // pure virtual
     virtual void draw() = 0;       // pure virtual
     // ...
};
```

An abstract class may not be used as an argument type, as a function return type, or as the type of an explicit conversion. Pointers and references to an abstract class may be declared. For example,

```
shape x;           // error: object of abstract class
shape* p;          // ok
shape f();         // error
void g(shape);     // error
shape& h(shape&);  // ok
```

Pure virtual functions are inherited as pure virtual functions. For example,

```
class ab_circle : public shape {
     int radius;
public:
     void rotate(int) {}
     // ab_circle::draw() is a pure virtual
};
```

Since *shape::draw()* is a pure virtual function *ab_circle::draw()* is a pure virtual by default. The alternative declaration,

```
class circle : public shape {
     int radius;
public:
     void rotate(int) {}
     void draw(); // must be defined somewhere
};
```

would make class *circle* non−abstract and a definition of *circle::draw()* must be provided somewhere.

Member functions can be called from a constructor of an abstract class; the effect of calling a pure virtual function directly or indirectly for the object being created from such a constructor is undefined.

Summary of Scope Rules

The scope rules for C++ programs can now be summarized. These rules apply uniformly for all names (including *typedef−names* (**"The typedef Specifier"**) and *class−names* (**"Class Names"**)) wherever the grammar allows such names in the context discussed by a particular rule. This section discusses lexical scope only; see **"Program and Linkage"** for an explanation of linkage issues. The notion of point of declaration is discussed in (**"Scopes"**).

Any use of a name must be unambiguous (up to overloading) in its scope (**"Ambiguities"**). Only if the name is found to be unambiguous in its scope are access rules considered (**Chapter 11, "Member Access Control"**). Only if no access control errors are found is the type of the object, function, or enumerator named considered.

A name used outside any function and class or prefixed by the unary scope operator *::* (and *not* qualified by the binary *::* operator or the *−>* or . operators) must be the name of a global object, function, enumerator, or type.

A name specified after *X::*, after *obj.*, where *obj* is an *X* or a reference to *X*, or after *ptr−>*, where *ptr* is a pointer to *X* must be the name of a member of class *X* or be a member of a base class of *X*. In addition, *ptr* in *ptr−>* may be an object of a class *Y* that has *operator−>()* declared so *ptr−>operator−>()* eventually resolves to a pointer to *X* (**"Class Member Access"**).

A name that is not qualified in any of the ways described above and that is used in a function that is not a class member must be declared in the block in which it occurs or in an enclosing block or be a global name. The declaration of a local name hides declarations of the same name in enclosing blocks and global names. In particular, no overloading occurs of names in different scopes (**"Overloaded Operators"**).

A name that is not qualified in any of the ways described above and that is used in a function that is a non−static member of class *X* must be declared in the block in which it occurs or in an enclosing block, be a member of class *X* or a base class of class *X*, or be a global name. The declaration of a local name hides declarations of the same name in enclosing blocks, members of the function's class, and global names. The declaration of a member name hides declarations of the same name in base classes and global names.

A name that is not qualified in one of the ways described above and is used in a static member function of a class *X* must be declared in the block in which it occurs, in an enclosing block, be a static member of class *X*, or a base class of class *X*, or be a global name.

A function argument name in a function definition (**"Function Definitions"**) is in the scope of the outermost block of the function (in particular, it is a local name). A function argument name in a function declaration (**"Functions"**) that is not a function definition is in a local scope that disappears immediately after the function declaration. A default argument is in the scope determined by the point of declaration (**"Scopes"**) of its formal argument, but may not access local variables or non−static class members; it is evaluated at each point of call (**"Default Arguments"**).

A *ctor−initializer* (**"Initializing Bases and Members"**) is evaluated in the scope of the outermost block of the constructor it is specified for. In particular, it can refer to the constructor's argument names.

Chapter 11

Member Access Control

This Chapter contains the following subsections:

- **"Access Specifiers"**
- **"Access Specifiers for Base Classes"**
- **"Access Declarations"**
- **"Friends"**
- **"Protected Member Access"**
- **"Access to Virtual Functions"**
- **"Multiple Access"**

A member of a class can be

- *private*; that is, its name can be used only by member functions and friends of the class in which it is declared.
- *protected*; that is, its name can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class (see **"Protected Member Access"**).
- *public*; that is, its name can be used by any function.

Members of a class declared with the keyword *class* are *private* by default. Members of a class declared with the keywords *struct* or *union* are *public* by default. For example,

```
class X {
     int a;  // X::a is private by default
};
struct S {
     int a;  // S::a is public by default
};
```

Access Specifiers

Member declarations may be labeled by an *access–specifier* (**Chapter 10, "Derived Classes"**):

access-specifier : member-list$_{opt}$

An *access–specifier* specifies the access rules for members following it until the end of the class or until another *access–specifier* is encountered. For example,

```
class X {
     int a;  // X::a is private by default: ’class’ used
public:
     int b;  // X::b is public
     int c;  // X::c is public
};
```

Any number of access specifiers is allowed and no particular order is required. For example,

```
struct S {
     int a;  // S::a is public by default: ’struct’ used
protected:
     int b;  // S::b is protected
private:
     int c;  // S::c is private
public:
     int d;  // S::d is public
};
```

The order of allocation of data members with separate *access–specifier* labels is implementation dependent (**"Class Members"**).

Access Specifiers for Base Classes

If a class is declared to be a base class (**Chapter 10, "Derived Classes"**) for another class using the *public* access specifier, the *public* members of the base class are *public* members of the derived class and *protected* members of the base class are *protected* members of the derived class. If a class is declared to be a base class for another class using the *private* access specifier, the *public* and *protected* members of the base class are *private* members of the derived class. Private members of a base class remain inaccessible even to derived classes unless *friend* declarations within the base class declaration are used to grant access explicitly.

In the absence of an *access−specifier* for a base class, *public* is assumed when the derived class is declared *struct* and *private* is assumed when the class is declared *class*. For example,

```
class B { /* ... */ };
class D1 : private B { /* ... */ };
class D2 : public B { /* ... */ };
class D3 : B { /* ... */ };    // 'B' private by default
struct D4 : public B { /* ... */ };
struct D5 : private B { /* ... */ };
struct D6 : B { /* ... */ };   // 'B' public by default
```

Here *B* is a public base of *D2*, *D4*, and *D6*, and a private base of *D1*, *D3*, and *D5*.

Specifying a base class *private* does not affect access to static members of the base class. If, however, an object or a pointer requiring conversion is used to select the static member the usual rules for pointer conversions apply.

Members and friends of a class *X* can implicitly convert an *X** to a pointer to a private immediate base class of *X*.

Access Declarations

The access to a member of a base class in a derived class can be adjusted by mentioning its *qualified−name* in the *public* or *protected* part of a derived class declaration. Such mention is called an *access declaration.* For example,

```
class B {
     int a;
public:
     int b, c;
     int bf();
};
class D : private B {
     int d;
public:
     B::c;  // adjust access to 'B::c'
     int e;
     int df();
};
int ef(D&);
```

The external function *ef* can use only the names *c*, *e*, and *df*. Being a member of *D*, the function *df* can use the names *b*, *c*, *bf*, *d*, *e*, and *df*, but not *a*. Being a member of *B*, the function *bf* can use the members *a*, *b*, *c*, and *bf*.

An access declaration may not be used to restrict access to a member that is accessible in the base class, nor may it be used to enable access to a member that is not accessible in the base class. For example,

```
class B {
```

```
public:
     int a;
private:
     int b;
protected:
     int c;
};
class D : private B {
public:
     B::a;  // make 'a' a public member of D
     B::b;  // error: attempt to grant access
            // can't make 'b' a public member of D
protected:
     B::c;  // make 'c' a protected member of D
     B::a;  // error: attempt to reduce access
            // can't make 'a' a protected member of D
};
```

An access declaration for the name of an overloaded function adjusts the access to all functions of that name in the base class. For example,

```
class X {
public:
     f();
     f(int);
};
class Y : private X {
public:
     X::f;  // makes X::f() and X::f(int) public in Y
};
```

The access to a base class member cannot be adjusted in a derived class that also defines a member of that name. For example,

```
class X {
public:
     void f();
};
class Y : private X {
public:
     void f(int);
     X::f;  // error: two declarations of f
};
```

Friends

A friend of a class is a function that is not a member of the class but is permitted to use the private and protected member names from the class. The name of a friend is not in the scope of the class, and the friend is not called with the member access operators (**"Class Member Access"**) unless it is a member of another class. The following example illustrates the differences between members and friends:

```
class X {
     int a;
     friend void friend_set(X*, int);
public:
     void member_set(int);
};

void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }
```

```
void f()
{
     X obj;
     friend_set(&obj,10);
     obj.member_set(10);
}
```

When a *friend* declaration refers to an overloaded name or operator, only the function specified by the argument types becomes a friend. A member function of a class *X* can be a friend of a class *Y*. For example,

```
class Y {
     friend char* X::foo(int);
     // ...
};
```

All the functions of a class *X* can be made friends of a class *Y* by a single declaration using an *elaborated−type−specifier* (**"Class Names"**):

```
class Y {
     friend class X;
     // ...
};
```

Declaring a class to be a friend also implies that private and protected names from the class granting friendship can be used in the class receiving it. For example,

```
class X {
     enum { a=100 };
     friend class Y;
};
class Y {
     int v[X::a];  // ok, Y is a friend of X
};
class Z {
     int v[X::a];  // error: X::a is private
};
```

If a class or a function mentioned as a friend has not been declared its name is entered in the same scope as the name of the class containing the friend declaration (**"Class Names"**).

A function first declared in a friend declaration is equivalent to an *extern* declaration (**"Program and Linkage"** , **"Storage Class Specifiers"**).

A *friend* function defined in a class declaration is *inline* and the rewriting rule specified for member functions (**"Inline Member Functions"**) is applied. A *friend* function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not.

Friend declarations are not affected by *access−specifiers* (**"Class Members"**).

Friendship is neither inherited nor transitive. For example,

```
class A {
     friend class B;
     int a;
};
class B {
     friend class C;
};
class C  {
     void f(A* p)
     {
        p->a++;  // error: C is not a friend of A
```

```
        p->a++;  // error: C is not a friend of A
                 // despite being a friend of a friend
     }
};
class D : public B  {
     void f(A* p)
     {
        p->a++;  // error: D is not a friend of A
                 // despite being derived from a friend
     }
};
```

Protected Member Access

A friend or a member function of a derived class can access a protected static member of a base class. A friend or a member function of a derived class can access a protected non−static member of one of its base classes only through a pointer to, reference to, or object of the derived class (or any class derived from that class). For example,

```
class B {
protected:
     int i;
};
class D1 : public B {
};
class D2 : public B {
     friend void fr(B*,D1*,D2*);
     void mem(B*,D1*);
};
void fr(B* pb, D1* p1, D2* p2)
{
     pb->i = 1;  // illegal
     p1->i = 2;  // illegal
     p2->i = 3;  // ok (access through a D2)
}
void D2::mem(B* pb, D1* p1)
{
     pb->i = 1;  // illegal
     p1->i = 2;  // illegal
     i = 3;      // ok (access through 'this')
}
void g(B* pb, D1* p1, D2* p2)
{
     pb->i = 1;  // illegal
     p1->i = 2;  // illegal
     p2->i = 3;  // illegal
}
```

Access to Virtual Functions

The access rules (**Chapter 11, "Member Access Control"**) for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it. For example,

```
class B {
public:
     virtual f();
};
class D : public B {
```

```
private:
     f();
};
void f()
{
     D d;
     B* pb = &d;
     D* pd = &d;
     pb->f();  // ok: B::f() is public,
               // D::f() is invoked
     pd->f();  // error: D::f() is private
}
```

Access is checked at the call point using the type of the expression used to denote the object for which the member function is called (*B** in the example above). The access of the member function in the class in which it was defined (*D* in the example above) is in general not known.

Multiple Access

If a name can be reached by several paths through a multiple inheritance graph, the access is that of the path that gives most access. For example,

```
class W { public: void f(); }; class A : private virtual W { };
class B : public virtual W { }; class C : public A, public B {
           void f() { W::f(); }  // ok };
```

Since *W::f()* is available to *C::f()* along the public path through *B*, access is legal.

Chapter 12

Special Member Functions

This chapter contains the following subsections:

- **"Constructors"**
- **"Temporary Objects"**
- **"Conversions"**
- **"Destructors"**
- **"Free Store"**
- **"Initialization"**
- **"Constructors and Destructors"**
- **"Copying Class Objects"**

Some member functions are special in that they affect the way objects of a class are created, copied, and destroyed, and how values may be converted to values of other types. Often such special functions are called implicitly.

These member functions obey the usual access rules (**Chapter 11, "Member Access Control"**). For example, declaring a constructor *protected* ensures that only derived classes and friends can create objects using it.

Constructors

A member function with the same name as its class is called a constructor; it is used to construct values of its class type. If a class has a constructor, each object of that class will be initialized before any use is made of the object; see **"Initialization"** .

A constructor can be invoked for a *const* or *volatile* object. A constructor may not be declared *const* or *volatile* (**"The this Pointer"**). A constructor may not be *virtual*. A constructor may not be *static*.

Constructors are not inherited. Default constructors and copy constructors, however, are generated (by the compiler) where needed (**"Copying Class Objects"**). Generated constructors are *public*.

A *default constructor* for a class *X* is a constructor of class *X* that can be called without an argument. A default constructor will be generated for a class *X* only if no constructor has been declared for class *X*.

A *copy constructor* for a class *X* is a constructor that can be called to copy an object of class *X*; that is, one that can be called with a single argument of type *X*. For example, *X::X(const X&)* and *X::X(X&,int=0)* are copy constructors. A copy constructor is generated only if no copy constructor is declared.

A copy constructor for a class *X* may not take an argument of type *X*. For example, *X::X(X)* is illegal.

Constructors for array elements are called in order of increasing addresses (**"Arrays"**).

If a class has base classes or member objects with constructors, their constructors are called before the constructor for the derived class. The constructors for base classes are called first. See **"Initializing Bases and Members"** for an explanation of how arguments can be specified for such constructors and how the order of constructor calls is determined.

An object of a class with a constructor cannot be a member of a union.

No return type (not even *void*) can be specified for a constructor. A *return* statement in the body of a constructor may not specify a return value. It is not possible to take the address of a constructor.

A constructor can be used explicitly to create new objects of its type, using the syntax

class-name (expression-list$_{opt}$)

For example,

```
complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );
```

An object created in this way is unnamed (unless the constructor was used as an initializer for a named variable as for *zz* above), with its lifetime limited to the expression in which it is created; see **"Temporary Objects"** .

Member functions may be called from within a constructor; see **"Constructors and Destructors"** .

Temporary Objects

In some circumstances it may be necessary or convenient for the compiler to generate a temporary object. Such introduction of temporaries is implementation dependent. When a compiler introduces a temporary object of a class that has a constructor it must ensure that a constructor is called for the temporary object.

Similarly, the destructor must be called for a temporary object of a class where a destructor is declared. For example,

```
class X {
     // ...
public:
     // ...
     X(int);
     X(X&);
     ~X();
};
X f(X);
void g()
{
     X a(1);
     X b = f(X(2));
     a = f(a);
}
```

Here, one might use a temporary in which to construct *X(2)* before passing it to *f()* by *X(X&)*; alternatively, *X(2)* might be constructed in the space used to hold the argument for the first call of *f()*. Also, a temporary might be used to hold the result of *f(X(2))* before copying it to *b* by *X(X&)*; alternatively, *f()*'s result might be constructed in *b*. On the other hand, for many functions , the expression *a=f(a)* requires a temporary for either the argument *a* or the result of *f(a)* to avoid undesired aliasing of *a*.

The compiler must ensure that a temporary object is destroyed. The exact point of destruction is implementation dependent. There are only two things that can be done with a temporary: fetch its value (implicitly copying it) to use in some other expression, or bind a reference to it. If the value of a temporary is fetched, that temporary is then dead and can be destroyed immediately. If a reference is bound to a temporary, the temporary must not be destroyed until the reference is. This destruction must take place before exit from the scope in which the temporary is created.

Another form of temporaries is discussed in **"References"** .

Conversions

Type conversions of class objects can be specified by constructors and by conversion functions.

Such conversions, often called *user−defined conversions,* are used implicitly in addition to standard conversions (**Chapter 4, "Standard Conversions"**). For example, a function expecting an argument of type *X* can be called not only with an argument of type *X* but also with an argument of type *T* where a conversion from *T* to *X* exists.

User−defined conversions are used similarly for conversion of initializers (**"Initializers"**), function arguments (**"Function Call"** , **"Functions"**), function return values (**"The return Statement"** , **"Functions"**), expression operands (**Chapter 5, "Expressions"**), expressions controlling iteration and selection statements (**"Selection Statements"** ,**"Iteration Statements"**), and explicit type conversions (**"Explicit Type Conversion"** , **"Explicit Type Conversion"**).

User−defined conversions are applied only where they are unambiguous (**"Ambiguities"** , **"Conversion Functions"**). Conversions obey the access control rules (**Chapter 11, "Member Access Control"**). As ever access control is applied after ambiguity resolution (**"Summary of Scope Rules"**).

See **"Argument Matching"** for a discussion of the use of conversions in function calls as well as examples below.

Conversion by Constructor

A constructor accepting a single argument specifies a conversion from its argument type to the type of its class. For example,

```
class X {
     // ...
public:
     X(int);
     X(const char*, int = 0);
};
void f(X arg) {
     X a = 1;          // a = X(1)
     X b = "Jessie";   // b = X("Jessie",0)
     a = 2;            // a = X(2)
     f(3);             // f(X(3))
}
```

When no constructor for class *X* accepts the given type, no attempt is made to find other constructors or conversion functions to convert the assigned value into a type acceptable to a constructor for class *X*. For example,

```
class X { /* ... */ X(int); };
class Y { /* ... */ Y(X); };
Y a = 1;             // illegal: Y(X(1)) not tried
```

Conversion Functions

A member function of a class *X* with a name of the form

conversion−function−name:
    operator conversion−type−name

conversion−type−name:
    type−specifier−list ptr−operator$_{opt}$

specifies a conversion from *X* to the type specified by the *conversion−type−name.*

Such member functions are called conversion functions. Classes, enumerations, and *typedef−names* may not be declared in the *type−specifier−list.* Neither argument types nor return type may be specified. For example,

```
class X {
     // ...
public:
     operator int();
};
void f(X a)
{
     int i = int(a);
     i = (int)a;
     i = a;
}
```

In all three cases the value assigned will be converted by *X::operator int()*. User−defined conversions are not restricted to use in assignments and initializations. For example,

```
void g(X a, X b)
{
     int i = (a) ? 1+a : 0;
     int j = (a&&b) ? a+b : i;
     if (a) { // ...
     }
}
```

Conversion operators are inherited.

Conversion functions can be virtual.

At most one user−defined conversion (constructor or conversion function) is implicitly applied to a single value. For example,

```
class X {
     // ...
public:
     operator int();
};
class Y {
     // ...
public:
     operator X();
};
Y a;
int b = a;     // illegal:
                 // a.operator X().operator int() not tried
int c = X(a); // ok: a.operator X().operator int()
```

User−defined conversions are used implicitly only if they are unambiguous. A conversion function in a derived class does not hide a conversion function in a base class unless the two functions convert to the same type. For example,

```
class X {
public:
     // ...
     operator int();
};
class Y : public X {
public:
     // ...
     operator void*();
};
void f(Y& a)
{
     if (a) {     // error: ambiguous
     // ...
     }
}
```

Destructors

A member function of class *cl* named *~cl* is called a destructor; it is used to destroy values of type *cl* immediately before the object containing them is destroyed. A destructor takes no arguments, and no return type can be specified for it (not even *void*). It is not possible to take the address of a destructor. A destructor can be invoked for a *const* or *volatile* object. A destructor may not be declared *const* or *volatile* (**"The this Pointer"**). A destructor may not be *static*.

Destructors are not inherited. If a base or a member has a destructor and no destructor is declared for its derived class a default destructor is generated. This generated destructor calls the destructors for bases and members of the derived class. Generated destructors are *public*.

The body of a destructor is executed before the destructors for member objects. Destructors for non−static member objects are executed before the destructors for base classes. Destructors for non−virtual base classes are executed before destructors for virtual base classes. Destructors for non−virtual base classes are executed in reverse order of their declaration in the derived class. Destructors for virtual base classes are executed in the reverse order of their appearance in a depth−first left−to−right traversal of the directed acyclic graph of base classes; "left−to−right" is the order of appearance of the base class names in the declaration of the derived class.

Destructors for elements of an array are called in reverse order of their construction.

A destructor may be *virtual*.

Member functions may be called from within a destructor; see **"Constructors and Destructors"** .

An object of a class with a destructor cannot be a member of a union.

Destructors are invoked implicitly;

- When an *auto* (**"Storage Classes"**) or temporary (**"Temporary Objects"** , **"References"**) object goes out of scope.
- For constructed static (**"Storage Classes"**) objects at program termination (**"Start and Termination"**).
- Through use of the *delete* operator (**"Delete"**) for objects allocated by the *new* operator (**"New"**).
- Explicitly called, when invoked by the *delete* operator, memory is freed by the destructor for the most derived class (**"Initializing Bases and Members"**) of the object using an *operator delete()* (**"Delete"**). For example,

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

```
void g(X*);

void f()     // common use:
{
     X* p = new X(111);  // allocate and initialize
     g(p);
     delete p;           // cleanup and deallocate
}
```

Explicit calls of destructors are rarely needed. One use of such calls is for objects placed at specific addresses using a *new* operator. Such use of explicit placement and destruction of objects can be necessary to cope with dedicated hardware resources and for writing memory management facilities. For example,

```
void* operator new(size_t, void* p) { return p; }
void f(X* p);
static char buf[sizeof(X)];
void g()     // rare, specialized use:
{
     X* p = new(buf) X(222);  // use buf[]
               // and initialize
     f(p);
     p->X::~X();              // cleanup
}
```

The notation for explicit call of a destructor may be used for any simple type name. For example,

```
int* p;
// ...
p->int::~int();
```

Using the notation for a type that does not have a destructor has no effect. Allowing this enables people to write code without having to know if a destructor exists for a given type.

Free Store

When an object is created with the *new* operator, an *operator new()* function is (implicitly) used to obtain the store needed (**"New"**).

If *operator new()* cannot allocate storage it will return 0.

An *X::operator new()* for a class *X* is a static member (even if not explicitly declared *static*). Its first argument must be of type *size_t*, an implementation−dependent integral type defined in the standard header *<stddef.h>*; it must return *void**. For example,

```
class X {
     // ...
     void* operator new(size_t);
     void* operator new(size_t, Arena*);
};
```

See **"New"** for the rules for selecting an *operator new()*.

An *X::operator delete()* for a class *X* is a static member (even if not explicitly declared *static*) and must have its first argument of type *void**; a second argument of type *size_t* may be added. It cannot return a value; its return type must be *void*. For example,

```
class X {
     // ...
     void operator delete(void*);
};
class Y {
```

```
    // ...
    void operator delete(void*, size_t);
};
```

Only one *operator delete()* may be declared for a single class; thus *operator delete()* cannot be overloaded. The global *operator delete()* takes a single argument of type *void**.

If the two argument style is used, *operator delete()* will be called with a second argument indicating the size of the object being deleted. The size passed is determined by the destructor (if any) or by the (static) type of the pointer being deleted; that is, it will be correct either if the type of the pointer argument to the *delete* operator is the exact type of the object (and not, for example, just the type of base class) or if the type is that of a base class with a virtual destructor.

The global *operator new()* and *operator delete()* are used for arrays of class objects (**"New"** , **"Delete"**).

Since *X::operator new()* and *X::operator delete()* are *static* they cannot be *virtual*. A destructor finds the *operator delete()* to use for freeing store using the usual scope rules. For example,

```
struct B {
    virtual ~B();
    void* operator new(size_t);
    void operator delete(void*);
};
struct D : B {
    ~D();
    void* operator new(size_t);
    void operator delete(void*);
};
void f()
{
    B* p = new D;
    delete p;
}
```

Here, storage for the object of class *D* is allocated by *D::operator new()* and, thanks to the virtual destructor, deallocated by *D::operator delete()*.

Initialization

An object of a class with no constructors, no private or protected members, no virtual functions, and no base classes can be initialized using an initializer list; see **"Aggregates"** . An object of a class with a constructor must either be initialized or have a default constructor (**"Constructors"**). The default constructor is used for objects that are not explicitly initialized.

Explicit Initialization

Objects of classes with constructors (**"Constructors"**) can be initialized with a parenthesized expression list. This list is taken as the argument list for a call of a constructor doing the initialization. Alternatively a single value is specified as the initializer using the = operator. This value is used as the argument to a copy constructor. Typically, that call of a copy constructor can be eliminated. For example,

```
class complex {
    // ...
public:
    complex();
    complex(double);
    complex(double,double);
    // ...
};
complex sqrt(complex,complex);
```

```
complex a(1);             // initialize by a call of
     // complex(double)
complex b = a;            // initialize by a copy of 'a'
complex c = complex(1,2); // construct complex(1,2)
     // using complex(double,double)
     // copy it into 'c'
complex d = sqrt(b,c);    // call sqrt(complex,complex)
     // and copy the result into 'd'
complex e;                // initialize by a call of
     // complex()
complex f = 3;            // construct complex(3) using
     // complex(double)
     // copy it into 'f'
```

Overloading of the assignment operator = has no effect on initialization.

The initialization that occurs in argument passing and function return is equivalent to the form

```
T x = a;
```

The initialization that occurs in *new* expressions (**"New"**) and in base and member initializers (**"Initializing Bases and Members"**) is equivalent to the form

```
T x(a);
```

Arrays of objects of a class with constructors use constructors in initialization (**"Constructors"**) just like individual objects. If there are fewer initializers in the list than elements in the array, the default constructor (**"Constructors"**) is used. If there is no default constructor the *initializer−list* must be complete. For example,

```
complex cc = { 1, 2 }; // error; use constructor
complex v[6] = { 1,complex(1,2),complex(),2 };
```

Here, *v[0]* and *v[3]* are initialized with *complex::complex(double)*, *v[1]* is initialized with *complex::complex(double,double)*, and *v[2]*, *v[4]*, and *v[5]* are initialized with *complex::complex()*.

An object of class *M* can be a member of a class *X* only if (1) *M* does not have a constructor, or (2) *M* has a default constructor, or (3) *X* has a constructor and if every constructor of class *X* specifies a *ctor−initializer* (**"Initializing Bases and Members"**) for that member. In case 2 the default constructor is called when the aggregate is created. If a member of an aggregate has a destructor, then that destructor is called when the aggregate is destroyed.

Constructors for non−local static objects are called in the order they occur in a file; destructors are called in reverse order. See also **"Start and Termination"** , **"Declaration Statement"** , **"Static Members"** .

Initializing Bases and Members

Initializers for immediate base classes and for members not inherited from a base class may be specified in the definition of a constructor. This is most useful for class objects, constants, and references where the semantics of initialization and assignment differ. A *ctor−initializer* has the form

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$_{opt}$)

The argument list is used to initialize the named non−static member or base class object. This is the only way to initialize non−static *const* and reference members. For example,

```
struct B1 { B1(int); /* ... */ };
struct B2 { B2(int); /* ... */ };
struct D : B1, B2 {
     D(int);
     B1 b;
     const c;
};
D::D(int a) : B2(a+1), B1(a+2), c(a+3), b(a+4)
{ /* ... */ }
D d(10);
```

First, the base classes are initialized in declaration order (independent of the order of *mem−initializers),* then the members are initialized in declaration order (independent of the order of *mem−initializers),* then the body of *D::D()* is executed (**"Constructors"**). The declaration order is used to ensure that sub−objects and members are destroyed in the reverse order of initialization.

Virtual base classes constitute a special case. Virtual bases are constructed before any non−virtual bases and in the order they appear on a depth−first left−to−right traversal of the directed acyclic graph of base classes; "left−to−right" is the order of appearance of the base class names in the declaration of the derived class.

A *complete object* is an object that is not a sub−object representing a base class. Its class is said to be the *most derived* class for the object. All sub−objects for virtual base classes are initialized by the constructor of the most derived class. If a constructor of the most derived class does not specify a *mem−initializer* for a virtual base class then that virtual base class must have a default constructor or no constructors. Any *mem−initializers* for virtual classes specified in a constructor for a class that is not the class of the complete object are ignored. For example,

```
class V {
public:
     V();
     V(int);
     // ...
};
class A : public virtual V {
public:
     A();
     A(int);
     // ...
};
class B : public virtual V {
public:
     B();
     B(int);
     // ...
};
class C : public A, public B, private virtual V {
public:
     C();
     C(int);
     // ...
};
A::A(int i) : V(i) { /* ... */ }
B::B(int i) { /* ... */ }
C::C(int i) { /* ... */ }
     V v(1);     // use V(int)
     A a(2);     // use V(int)
     B b(3);     // use V()
     C c(4);     // use V()
```

A *mem−initializer* is evaluated in the scope of the constructor in which it appears. For example,

```
class X {
     int a;
public:
     const int& r;
     X(): r(a) {}
};
```

initializes *X::r* to refer to *X::a* for each object of class *X*.

Constructors and Destructors

Member functions may be called in constructors and destructors. This implies that virtual functions may be called (directly or indirectly). The function called will be the one defined in the constructor's (or destructor's) own class or its bases, but *not* any function overriding it in a derived class. This ensures that unconstructed objects will not be accessed during construction or destruction. For example,

```
class X {
public:
     virtual void f();
     X() { f(); }   // calls X::f()
     ~X() { f(); }  // calls X::f()
};
class Y : public X {
     int& r;
public:
     void f()
     {
        r++;  // disaster if 'r' is uninitialized
     }
     Y(int& rr) :r(rr) {}
};
```

The effect of calling a pure virtual function directly or indirectly for the object being constructed from a constructor, except using explicit qualification, is undefined (**"Abstract Classes"**).

Copying Class Objects

A class object can be copied in two ways, by assignment (**"Assignment Operators"**) and by initialization (**"Constructors"** , **"Initializers"**) including function argument passing (**"Function Call"**) and function value return (**"The return Statement"**). Conceptually, for a class *X* these two operations are implemented by an assignment operator and a copy constructor (**"Constructors"**).

The programmer may define either or both the assignment operator and/or copy constructor. If not defined by the programmer, they will be defined as memberwise assignment and memberwise initialization of the members of *X*, respectively.

If all bases and members of a class *X* have copy constructors accepting *const* arguments, the generated copy constructor for *X* will take a single argument of type *const X&*, as follows:

```
X::X(const X&)
```

Otherwise it will take a single argument of type *X& :*

```
X::X(X&)
```

and initialization by copying of *const X* objects will not be possible.

Similarly, if all bases and members of a class *X* have assignment operators accepting *const* arguments, the generated assignment operator for *X* will take a single argument of type *const X&*, as follows:

```
X& X::operator=(const X&)
```

Otherwise it will take a single argument of type *X&*:

```
X& X::operator=(X&)
```

and assignment by copying of *const X* objects will not be possible. The default assignment operator will return a reference to the object for which is invoked.

Objects representing virtual base classes will be initialized only once by a generated copy constructor. Objects representing virtual base classes will be assigned only once by a generated assignment operator.

Memberwise assignment and memberwise initialization implies that if a class *X* has a member of a class *M*, *M*'s assignment operator and *M*'s copy constructor are used to implement assignment and initialization of the member, respectively. If a class has a *const* member, a reference member, or a member or a base of a class with a private *operator=()*, the default assignment operation cannot be generated. Similarly, if a member or a base of a class *M* has a private copy constructor then the default copy constructor cannot be generated.

The default assignment and copy constructor will be declared, but they will not be defined (that is, a function body generated) unless needed. That is, *X::operator=()* will be generated only if no assignment operation is explicitly declared and an object of class *X* is assigned an object of class *X* or an object of a class derived from *X* or if the address of *X::operator=* is taken. Initialization is handled similarly.

If implicitly declared, the assignment and the copy constructor will be public members and the assignment operator for a class *X* will be defined to return a reference of type *X&* referring to the object assigned.

If a class *X* has any *X::operator=()* that takes an argument of class *X*, the default assignment will not be generated. If a class has any copy constructor defined, the default copy constructor will not be generated. For example,

```
class X {
     // ...
public:
     X(int);
     X(const X&, int = 1);
};
X a(1);         // calls X(int);
X b(a,0);       // calls X(const X&,int);
X c = b;        // calls X(const X&,int);
```

Assignment of class *X* objects is defined in terms of *X::operator=(const X&)*. This implies (**"Conversions"**) that objects of a derived class can be assigned to objects of a public base class. For example,

```
class X {
public:
     int b;
};
class Y : public X {
public:
     int c;
};
void f()
{
     X x1;
     Y y1;
     x1 = y1;     // ok
     y1 = x1;     // error
}
```

Here *y1.b* is assigned to *x1.b* and *y1.c* is not copied.

Copying one object into another using the default copy constructor or the default assignment operator does not change the structure of either object. For example,

```
struct s {
     virtual f();
     // ...
};
struct ss : public s {
     f();
     // ...
};
void f()
{
     s a;
     ss b;
     a = b;       // really a.s::operator=(b)
     b = a;       // error
     a.f();       // calls s::f
     b.f();       // calls ss::f
     (s&)b = a;   // assign to b's s part
                  // really ((s&)b).s::operator=(a)
     b.f();       // still calls ss::f
}
```

The call *a.f()* will invoke *s::f()* (as is suitable for an object of class *s* (**"Virtual Functions"**)) and the call *b.f()* will call *ss::f()* (as is suitable for an object of class *ss*).

Chapter 13
Overloading

This chapter contains the following subsections:

- **"Declaration Matching" in Chapter 13**
- **"Argument Matching"**
- **"Address of Overloaded Function"**
- **"Overloaded Operators"**

When several different function declarations are specified for a single name in the same scope, that name is said to be overloaded. When that name is used, the correct function is selected by comparing the types of the actual arguments with the types of the formal arguments. For example,

```
double abs(double);
int abs(int);
abs(1);        // call abs(int);
abs(1.0);      // call abs(double);
```

Since for any type *T*, a *T* and a *T&* accept the same set of initializer values, functions with argument types differing only in this respect may not have the same name. For example,

```
int f(int i)
{
     // ...
}
int f(int& r)   // error: function types
                  // not sufficiently different
{
     // ...
}
```

Similarly, since for any type *T*, a *T*, a *const T*, and a *volatile T* accept the same set of initializer values, functions with argument types differing only in this respect may not have the same name. It is, however, possible to distinguish between *const T&*, *volatile T&*, and plain *T&* so functions that differ only in this respect may be defined. Similarly, it is possible to distinguish between *const T**, *volatile T**, and plain *T** so functions that differ only in this respect may be defined.

Functions that differ only in the return type may not have the same name.

Member functions that differ only in that one is a *static* member and the other isn't may not have the same name (**"Static Members"**).

A *typedef* is not a separate type, but only a synonym for another type (**"The typedef Specifier"**). Therefore, functions that differ by typedef "types" only may not have the same name. For example,

```
typedef int Int;
void f(int i) { /* ... */ }
void f(Int i) { /* ... */ } // error: redefinition of f
```

Enumerations, on the other hand, are distinct types and can be used to distinguish overloaded functions. For example,

```
enum E { a };
void f(int i) { /* ... */ }
void f(E i)   { /* ... */ }
```

Argument types that differ only in a pointer *** versus an array *[]* are identical. Note that only the second and subsequent array dimensions are significant in argument types (**"Arrays"**).

```
f(char*);
f(char[]);  // same as f(char*);
f(char[7]); // same as f(char*);
f(char[9]); // same as f(char*);
g(char(*)[10]);
g(char[5][10]);  // same as g(char(*)[10]);
g(char[7][10]);  // same as g(char(*)[10]);
g(char(*)[20]);  // different from g(char(*)[10]);
```

Declaration Matching

Two function declarations of the same name refer to the same function if they are in the same scope and have identical argument types (**Chapter 13, "Overloading"**). A function member of a derived class is *not* in the same scope as a function member of the same name in a base class. For example,

```
class B {
public:
     int f(int);
};
class D : public B {
public:
     int f(char*);
};
```

Here *D::f(char*)* hides *B::f(int)* rather than overloading it.

```
void h(D* pd)
{
     pd->f(1);       // error:
                     // D::f(char*) hides B::f(int)
     pd->B::f(1);    // ok
     pd->f("Ben");   // ok, calls D::f
}
```

A locally declared function is not in the same scope as a function in file scope.

```
int f(char*);
```

```
void g()
{
     extern f(int);
     f("asdf");  // error: f(int) hides f(char*)
                 // so there is no f(char*) in this scope
}
```

Different versions of an overloaded member function may be given different access rules. For example,

```
class buffer {
private:
     char* p;
     int size;
protected:
     buffer(int s, char* store) { size = s; p = store; }
     // ...
public:
     buffer(int s) { p = new char[size = s]; }
     // ...
};
```

Argument Matching

A call of a given function name chooses, from among all functions by that name that are in scope and for which a set of conversions exists so that the function could possibly be called, the function that best matches the actual arguments. The best−matching function is the intersection of sets of functions that best match on each argument. Unless this intersection has exactly one member, the call is illegal. The function thus selected must be a strictly better match for at least one argument than every other possible function (but not necessarily the same argument for each function). Otherwise, the call is illegal.

For purposes of argument matching, a function with *n* default arguments (**"Default Arguments"**) is considered to be *n+1* functions with different numbers of arguments.

For purposes of argument matching, a non−static member function is considered to have an extra argument specifying the object for which it is called. This extra argument requires a match either by the object or pointer specified in the explicit member function call notation (**"Class Member Access"**) or by the first operand of an overloaded operator (**"Overloaded Operators"**). No temporaries will be introduced for this extra argument and no user−defined conversions will be applied to achieve a type match.

Where a member of a class *X* is explicitly called for a pointer using the *−>* operator, this extra argument is assumed to have type *const X** for *const* members, *volatile X** for *volatile* members, and *X** for others. Where the member function is explicitly called for an object using the . operator or the function is invoked for the first operand of an overloaded operator (**"Overloaded Operators"**), this extra argument is assumed to have type *const X&* for *const* members, *volatile X&* for *volatile* members, and *X&* for others. The first operand of *−>** and *.** is treated in the same way as the first operand of *−>* and *.* , respectively.

An ellipsis in a formal argument list (**"Functions"**) is a match for an actual argument of any type.

For a given actual argument, no sequence of conversions will be considered that contains more than one user−defined conversion or that can be shortened by deleting one or more conversions into another sequence that leads to the type of the corresponding formal argument of any function in consideration. Such a sequence is called a *best−matching* sequence.

For example, *int−>float−>double* is a sequence of conversions from *int* to *double*, but it is not a best−matching sequence because it contains the shorter sequence *int−>double*.

Except as mentioned below, the following *trivial conversions* involving a type *T* do not affect which of two conversion sequences is better:

From:	To:
T	T&

T&	T
T[]	T*
T(args)	T(*)(args)
T	const T
T	volatile T
T*	const T*
T*	volatile T*

Sequences of trivial conversions that differ only in order are indistinguishable. Note that functions with arguments of type *T*, *const T*, *volatile T*, *T&*, *const T&*, and *volatile T&* accept exactly the same set of values. Where necessary, *const* and *volatile* are used as tie−breakers as described in rule [1] below.

A temporary variable is needed for a formal argument of type *T&* if the actual argument is not an lvalue, has a type different from *T*, or is a *volatile* and *T* is not. This does not affect argument matching. It may, however, affect the legality of the resulting match since a temporary may not be used to initialize a non−*const* reference (**"References"**).

Sequences of conversions are considered according to these rules:

- [1] Exact match: Sequences of zero or more trivial conversions are better than all other sequences. Of these, those that do not convert *T** to *const T**, *T** to *volatile T**, *T&* to *const T&*, or *T&* to *volatile T&* are better than those that do.
- [2] Match with promotions: Of sequences not mentioned in [1], those that contain only integral promotions (**"Integral Promotions"**), conversions from *float* to *double*, and trivial conversions are better than all others.
- [3] Match with standard conversions: Of sequences not mentioned in [2], those with only standard (**"Integral Promotions"** , **"Integral Conversions"** , **"Float and Double"** , **"Floating and Integral"** , **"Arithmetic Conversions"** , **"Pointer Conversions"** , **"Reference Conversions"** , **"Pointers to Members"**) and trivial conversions are better than all others. Of these, if *B* is publicly derived directly or indirectly from *A*, converting a *B** to *A** is better than converting to *void** or *const void**; further, if *C* is publicly derived directly or indirectly from *B*, converting a *C** to *B** is better than converting to *A** and converting a *C&* to *B&* is better than converting to *A&*. The class hierarchy acts similarly as a selection mechanism for pointer to member conversions (**"Pointers to Members"**).
- [4] Match with user−defined conversions: Of sequences not mentioned in [3], those that involve only user−defined conversions (**"Conversions"**), standard (**Chapter 4, "Standard Conversions,"**) and trivial conversions are better than all other sequences.
- [5] Match with ellipsis: Sequences that involve matches with the ellipsis are worse than all others.

User−defined conversions are selected based on the type of variable being initialized or assigned.

```
class Y {
     // ...
public:
     operator int();
     operator double();
};
void f(Y y)
{
     int i = y;     // call Y::operator int()
     double d;
     d = y;         // call Y::operator double()
     float f = y;  // error: ambiguous
}
```

Standard conversions (**Chapter 4, "Standard Conversions"**) may be applied to the argument for a user−defined conversion, and to the result of a user−defined conversion.

```
struct S {  S(long); operator int(); };
void f(long), f(char*);
void g(S), g(char*);
void h(const S&), h(char*);
void k(S& a)
{
     f(a);     // f(long(a.operator int()))
```

```
     f(a);      // f(long(a.operator int()))
     g(1);      // g(S(long(1)))
     h(1);      // h(S(long(1)))
}
```

If user−defined coercions are needed for an argument, no account is taken of any standard coercions that might also be involved. For example,

```
class x {
public:
     x(int);
};
class y {
public:
     y(long);
};
void f(x);
void f(y);
void g()
{
     f(1);      // ambiguous
}
```

The call *f(1)* is ambiguous despite *f(y(long(1)))* needing one more standard conversion than *f(x(1))*.

No preference is given to conversion by constructor (**"Constructors"**) over conversion by conversion function (**"Conversion Functions"**) or vice versa.

```
struct X {
     operator int();
};
struct Y {
     Y(X);
};
Y operator+(Y,Y);
void f(X a, X b)
{
     a+b;  // error, ambiguous:
           // operator+(Y(a), Y(b)) or
           // a.operator int() + b.operator int()
}
```

Address of Overloaded Function

A use of a function name without arguments selects, among all functions of that name that are in scope, the (only) function that exactly matches the target. The target may be

- an object being initialized (**"Initializers"**)
- the left side of an assignment (**"Assignment Operators"**)
- a formal argument of a function (**"Function Call"**)
- a formal argument of a user−defined operator (**"Overloaded Operators"**)
- a function return type (**"Functions"**)

Note that if *f()* and *g()* are both overloaded functions, the cross product of possibilities must be considered to resolve *f(&g)*, or the equivalent expression *f(g)*. For example,

```
int f(double);
int f(int);
int (*pfd)(double) = &f;
int (*pfi)(int) = &f;
int (*pfe)(...) = &f; // error: type mismatch
```

The last initialization is an error because no *f()* with type *int(...)* has been defined, and not because of any ambiguity.

Note also that there are no standard conversions (**Chapter 4, "Standard Conversions"**) of one pointer to function type into another (**"Pointer Conversions"**). In particular, even if *B* is a public base of *D* we have

```
D* f();
B* (*p1)() = &f;        // error
void g(D*);
void (*p2)(B*) = &g;    // error
```

Overloaded Operators

Most operators can be overloaded. The complier recognizes various functions through the context in which the function call is used. The operands or arguments are different in types and number. This ability to recognize function calls in context allows most operators to be redefined for special−purpose overload functions.

The *operator−function−name* is the keyword *operator* followed by the *operator symbol.* When defining the operator's new overload action, the *operator−function−name* is used, just as a regular function name would be used. For example,

operators: one of

```
    new  delete
    +   −   *   /   %   ^   &   |   ~
    !   =   <   >   +=  −=  *=  /=  %=
    ^=  &=  |=  <<  >>  >>= <<= ==  !=
    <=  >=  &&  ||  ++  −−  ,   −>* −>
    ()  []
```

The last two operators are function call (**"Function Call"**) and subscripting (**"Subscripting"**).

Both the unary and binary forms of

```
+     −     *     &
```

can be overloaded.

The following operators cannot be overloaded:

```
.   .*   ::   ?:   sizeof
```

nor can the preprocessing symbols *#* and *##* (**Chapter 16, "Preprocessing"**).

Operator functions are usually not called directly; instead they are invoked to implement operators (**"Unary Operators"** , **"Binary Operators"**). They can be explicitly called, though. For example,

```
complex z = a.operator+(b);  // complex z = a+b;
void* p = operator new(sizeof(int)*n);
```

The operators *new* and *delete* are described in **"New"** and **"Delete"** and the rules described below in this section do not apply to them.

An operator function must either be a member function or take at least one argument of a class or a reference to a class. It is not possible to change the precedence, grouping, or number of operands of operators. The predefined meaning of the operators *=*, (unary) *&*, and *,* (comma) applied to class objects may be changed. Except for *operator=()*, operator functions are inherited; see **"Copying Class Objects"** for the rules for *operator=()*.

Identities among operators applied to basic types (for example, *++a ≡º º a+=1*) need not hold for operators applied to class types. Some operators, for example, *+=*, require an operand to be an lvalue when applied to basic types; this is not required when the operators are declared for class types.

An overloaded operator cannot have default arguments (**"Default Arguments"**).

Operators not mentioned explicitly below in **"Assignment"** to **"Increment and Decrement"** act as

Operators not mentioned explicitly below in **"Assignment"** to **"Increment and Decrement"** act as ordinary unary and binary operators obeying the rules of section **"Unary Operators"** or **"Binary Operators"** .

Unary Operators

A prefix unary operator may be declared by a non−static member function (**"Member Functions"**) taking no arguments or a non−member function taking one argument. Thus, for any prefix unary operator *@*, *@x* can be interpreted as either *x.operator@()* or *operator@(x)*. If both forms of the operator function have been declared, argument matching (**"Argument Matching"**) determines which, if any, interpretation is used. See **"Increment and Decrement"** for an explanation of postfix unary operators, that is, *++* and *−−*.

Binary Operators

A binary operator may be declared either by a non−static member function (**"Member Functions"**) taking one argument or by a non−member function taking two arguments. Thus, for any binary operator *@*, *x@y* can be interpreted as either *x.operator@(y)* or *operator@(x,y)*. If both forms of the operator function have been declared, argument matching (**"Argument Matching"**) determines which, if any, interpretation is used.

Assignment

The assignment function *operator=()* must be a non−static member function; it is not inherited (**"Copying Class Objects"**). Instead, unless the user defines *operator=* for a class *X*, *operator=* is defined, by default, as memberwise assignment of the members of class *X*.

```
X& X::operator=(const X& from)
{
     // copy members of X
}
```

Function Call

Function call

primary-expression (*expression-list*$_{opt}$)

is considered a binary operator with the *primary−expression* as the first operand and the possibly empty *expression−list* as the second. The name of the defining function is *operator()*. Thus, a call *x(arg1,arg2, arg3)* is interpreted as *x.operator()(arg1,arg2,arg3)* for a class object *x*. *operator()* must be a non−static member function.

Subscripting

Subscripting

primary-expression [*expression*]

is considered a binary operator. A subscripting expression *x[y]* is interpreted as *x.operator[](y)* for a class object *x*. *operator[]* must be a non−static member function.

Class Member Access

Class member access using *−>*

primary-expression -> *primary-expression*

is considered a unary operator. An expression *x−>m* is interpreted as *(x.operator−>())−>m* for a class object *x*. It follows that *operator−>()* must return either a pointer to a class or an object of or a reference to a class for which *operator−>()* is defined. *operator−>* must be a non−static member function.

Increment and Decrement

A function called *operator++* taking one argument defines the prefix increment operator *++* for objects of some class. A function called *operator++* taking two arguments defines the postfix increment operator *++* for objects of some class. For postfix *operator++*, the second argument must be of type *int* and the *operator++()* will be called with the second argument 0 when invoked by a postfix increment expression. For example,

```
class X {
public:
     X operator++();      // prefix ++a
     X operator++(int);   // postfix a++
};
void f(X a)
{
     ++a;     // a.operator++();
     a++;     // a.operator++(0);
     a.operator++();   // explicit call: like ++a;
     a.operator++(0);  // explicit call: like a++;
}
```

The prefix and postfix decrement operators *−−* are handled similarly.

Chapter 14

Templates

This chapter contains the following subsections:

- **"Class Templates"**
- **"Type Equivalence"**
- **"Function Templates"**
- **"Declarations and Definitions"**
- **"Member Function Templates"**
- **"Friends"**
- **"Static Members and Variables"**

A *template* defines a family of types or functions.

template−declaration:
  template < template−argument−list > declaration

template−argument−list:
   template−argument
  template−argument−list , template argument

template−argument:
  type−argument
  argument−declaration

type−argument:
  class identifier

The *declaration* in a *template−declaration* must declare or define a function or a class.

A *type−argument* defines its *identifier* to be a *type−name* in the scope of the template declaration.

Template names obey the usual scope and access control rules. A *template−declaration* is a *declaration.* A *template−declaration* may appear only as a global declaration.

Class Templates

A class template specifies how individual classes can be constructed much as a class declaration specifies how individual objects can be constructed. A vector class template might be declared like this:

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

The prefix *template <class T>* specifies that a template is being declared and that a *type−name T* will be used in the declaration. In other words, *vector* is a parameterized type with *T* as its parameter.

A class can be specified by a *template−class−name:*

template−class−name:
  template−name < template−arg−list >

template−arg−list:
  template−arg
  template−arg−list , template−arg

template−arg:
  expression
  type−name

A *template−class−name* is a *class−name* (**"Class Names"**).

A class generated from a class template is called a template class, as is a class specifically defined with a *template−class−name* as its name; see **"Member Function Templates"** .

A *template−class−name* where the *template−name* is not defined names an undefined class.

A class template name must be unique in a program and may not be declared to refer to any other template, class, function, object, value, or type in the same scope.

The types of the *template−args* specified in a *template−class−name* must match the types specified for the template in its *template−argument−list.*

Other *template−args* must be *constant−expressions,* addresses of objects or functions with external linkage, or of static class members. An exact match (**"Argument Matching"**) is required for non−type arguments.

For example, *vector*s can be used like this:

```
vector<int> v1(20);
vector<complex> v2(30);
typedef vector<complex> cvec;  // make cvec a synonym
     // for vector<complex>
cvec v3(40); // v2 and v3 are of the same type
v1[3] = 7;
v2[3] = v3.elem(4) = complex(7,8);
```

Here, *vector<int>* and *vector<complex>* are template classes, and their definitions will by default be generated from the *vector* template.

Since a *template−class−name* is a *class−name,* it can be used wherever a *class−name* can be used. For example,

```
class vector<Shape*>;
vector<Window>* current_window;
class svector : public vector<Shape*> { /* ... */ };
```

Definition of class template member functions is described in **"Friends"** .

Type Equivalence

Two *template-class-names* refer to the same class if their *template* names are identical and their arguments have identical values. For example,

```
template<class E, int size> class buffer;
buffer<char,2*512> x;
buffer<char,1024> y;
buffer<char,512> z;
```

declares *x* and *y* to be of the same type and *z* of a different type, and

```
template<class T, void(*err_fct)()>
     class list { /* ... */ };
list<int,&error_handler1> x1;
list<int,&error_handler2> x2;
list<int,&error_handler2> x3;
list<char,&error_handler2> x4;
```

declares *x2* and *x3* to be of the same type. Their type differs from the types of *x1* and *x4*.

Function Templates

A function template specifies how individual functions can be constructed. A family of sort functions, for example, might be declared like this:

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

A function template specifies an unbounded set of (overloaded) functions. A function generated from a function template is called a template function, as is a function defined with a type that matches a function template; see **"Member Function Templates"** .

Template arguments are not explicitly specified when calling a function template; instead, overloading resolution is used. For example,

```
vector<complex> cv(100);
vector<int> ci(200);
void f(vector<complex>& cv, vector<int>& ci)
{
     sort(cv);      // invoke sort(vector<complex>)
     sort(ci);      // invoke sort(vector<int>)
}
```

A template function may be overloaded either by (other) functions of its name or by (other) template functions of that same name. Overloading resolution for template functions and other functions of the same name is done in three steps:

- [1] Look for an exact match (**"Argument Matching"**) on functions; if found, call it.
- [2] Look for a function template from which a function that can be called with an exact match can be generated; if found, call it.
- [3] Try ordinary overloading resolution (**"Argument Matching"**) for the functions; if a function is found, call it.

If no match is found the call is an error. In each case, if there is more than one alternative in the first step that finds a match, the call is ambiguous and is an error.

A match on a template (step [2]) implies that a specific template function with arguments that exactly matches the types of the arguments will be generated (**"Member Function Templates"**). Not even trivial conversions (**"Argument Matching"**) will be applied in this case.

The same process is used for type matching for pointers to functions (**"Address of Overloaded Function"**).

Here is an example,

```
template<class T> T max(T a, T b) { return a>b?a:b; };
```

```
void f(int a, int b, char c, char d)
{
     int m1 = max(a,b);     // max(int a, int b)
     char m2 = max(c,d);    // max(char a, char b)
     int m3 = max(a,c);     // error: cannot generate
                            // max(int,char)
}
```

For example, adding

```
int max(int,int);
```

to the example above would resolve the third call, by providing a function that could be called for *max(a,c)* after using the standard conversion of *char* to *int* for *c*.

A function template definition is needed to generate specific versions of the template; only a function template declaration is needed to generate calls to specific versions.

Every *template−argument* specified in the *template−argument−list* must be used in the argument types of a function template.

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

All *template−arguments* for a function template must be *type−arguments.*

Declarations and Definitions

There must be exactly one definition for each template of a given name in a program. There can be many declarations. The definition is used to generate specific template classes and template functions to match the uses of the template.

Using a *template−class−name* constitutes a declaration of a template class.

Calling a function template or taking its address constitutes a declaration of a template function. There is no special syntax for calling or taking the address of a template function; the name of a function template is used exactly as is a function name. Declaring a function with the same name as a function template with a matching type constitutes a declaration of a specific template function.

If the definition of a specific template function or specific template class is needed to perform some operation and if no explicit definition of that specific template function or class is found in the program, a definition is generated.

The definition of a (non−template) function with a type that exactly matches the type of a function template declaration is a definition of that specific template function. For example,

```
template<class T> void sort(vector<T>& v) { /* ... */ }
void sort(vector<char*>& v)  { /* ... */ }
```

Here, the function definition will be used as the sort function for arguments of type *vector<char*>*. For other *vector* types the appropriate function definition is generated from the template.

A class can be defined as the definition of a template class. For example,

```
template<class T> class stream { /* ... */ };
class stream<char> { /* ... */ };
```

Here, the class declaration will be used as the definition of streams of characters (*stream<char>*). Other streams will be handled by template functions generated from the function template.

No operation that requires a defined class can be performed on a template class until the class template has been seen. After that, a specific template class is considered defined immediately before the first global declaration that names it.

Member Function Templates

A member function of a template class is implicitly a template function with the template arguments of its class as its template arguments. For example,

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

declares three function templates. The subscript function might be defined like this:

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

The template argument for *vector<T>::operator[]()* will be determined by the vector to which the subscripting operation is applied.

```
vector<int> v1(20);
vector<complex> v2(30);
v1[3] = 7;            // vector<int>::operator[]()
v2[3] = complex(7,8); // vector<complex>::operator[]()
```

Friends

A friend function of a template is not implicitly a template function. For example,

```
template<class T> class task {
     // ...
     friend void next_time();
     friend task<T>* preempt(task<T>*);
     friend task* prmt(task*);     // error
     // ...
};
```

Here, *next_time()* becomes the friend of all *task* classes, and each *task* has an appropriately typed function called *preempt()* as a friend. The *preempt* functions might be defined as a template.

```
template<class T>
     task<T>* preempt(task<T>* t) { /* ... */ }
```

The declaration of *prmt()* is an error because there is no type *task* , only specific template types, *task<int>*, *task<record>*, and so on.

Static Members and Variables

Each template class or function generated from a template has its own copies of any static variables or members. For example,

```
template<class T> class X {
     static T s;
     // ...
};
X<int> aa;
X<char*> bb;
```

Here *X<int>* has a static member *s* of type *int* and *X<char*>* has a static member *s* of type *char**.

Similarly,

```
template<class T> f(T* p)
{
     static T s;
     // ...
};
void g(int a, char* b)
{
     f(&a);
     f(&b);
}
```

Here *f(int*)* has a static member *s* of type *int* and *f(char**)* has a static member *s* of type *char***.

Chapter 15

Exception Handling

Exception handling, as described in Ellis and Stroustrup: *The Annotated C++ Reference Manual* (Addison−Wesley 1990) and in Stroustrup: *The C++ Programming Language (2nd edition)* (Addison−Wesley 1991), has been adopted into the working drafts of the ANSI and ISO C++ standards committies. It is not supported by this release, but will be supported in some future release.

Chapter 16

Preprocessing

This chapter contains the following subsections:

- **"Phases of Preprocessing"**
- **"Trigraph Sequences"**
- **"Macro Definition and Expansion"**
- **"File Inclusion"**
- **"Conditional Compilation"**
- **"Line Control"**
- **"Error Directive"**
- **"Pragmas"**
- **"Null Directive"**
- **"Predefined Names"**

A C++ implementation contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files.

Lines beginning with *#*, optionally preceded by space and horizontal tab characters, (also called "*directives*") communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effects that last (independent of the scoping rules of C++) until the end of the translation unit (**Chapter 2, "Lexical Conventions"**).

A preprocessing directive (or any other line) may be continued on the next line in a source file by placing a backslash character, **, immediately before the new−line at the end of the line to be continued. The preprocessor effects the continuation by deleting the backslash and the new−line before the input sequence is divided into tokens. A backslash character may not be the last character in a source file.

A preprocessing token is a language token (**"Tokens"**), a file name as in a *#include* directive, or any single character, other than white space, that does not match another preprocessing token.

Phases of Preprocessing

Preprocessing is defined to occur in several phases. An implementation may collapse these phases, but the effect must be as though they had been executed.

1. If needed, new−line characters are introduced to replace system−dependent end−of−line indicators. Any other necessary system−dependent character set translations are done. Trigraph sequences are replaced by their single character equivalents (**"Trigraph Sequences"**).
2. Each pair of a backslash character \ immediately followed by a new−line is deleted, with the effect that the next source line is appended to the line that contained the sequence.
3. The source text is decomposed into preprocessing tokens and sequences of white space. A single white space replaces each comment. A source file may not end with a partial token or comment.
4. Preprocessing directives are executed and macros are expanded (**"Macro Definition and Expansion"** , **"File Inclusion"** , **"Conditional Compilation"** , **"Line Control"** , **"Error Directive"** , and **"Pragmas"**).
5. Escape sequences in character constants and string literals are replaced by their equivalents (**"Character Constants"**).
6. Adjacent string literals are concatenated.

The result of preprocessing is syntactically and semantically analyzed and translated, then linked together as necessary with other programs and libraries.

Trigraph Sequences

Before any other processing takes place, each occurrence of one of the following sequences of three characters ("*trigraph sequences*") is replaced by the single character indicated in the following list.

??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??−	~

For example,

```
??=define arraycheck(a,b) a??(b??) ??!??! b??(a??)
```

becomes

```
#define arraycheck(a,b) a[b] || b[a]
```

Macro Definition and Expansion

A preprocessing directive of the form

```
#define identifier   token−string
```

causes the preprocessor to replace subsequent instances of the identifier with the given sequence of tokens. White space surrounding the replacement token sequence is discarded. Given, for example,

```
#define SIDE 8
```

the declaration

```
char chessboard[SIDE][SIDE];
```

after macro expansion becomes

```
char chessboard[8][8];
```

An identifier defined in this form may be redefined only by another *#define* directive of this form provided the replacement list of the second definition is identical to that of the first. All white space separations are considered identical.

A line of the form

```
#define identifier( identifier, ... , identifier ) token-string
```

where there is no space between the first identifier and the *(* is a macro definition with parameters, or a " *function−lik*"" macro definition. An identifier defined as a function−like macro may be redefined by another function−like macro definition provided the second definition has the same number and spelling of parameters and the two replacement lists are identical. White space separations are considered identical.

Subsequent appearances of an identifier defined as a function−like macro followed by a *(*, a sequence of tokens delimited by commas, and a *)* are replaced by the token string in the definition. White space surrounding the replacement token sequence is discarded. Each occurrence of an identifier mentioned in the parameter list of the definition is replaced by the tokens representing the corresponding actual argument in the call. The actual arguments are token strings separated by commas; commas in quoted strings, in character constants, or within nested parentheses do not separate arguments. The number of arguments in a macro invocation must be the same as the number of parameters in the macro definition.

Once the arguments to a function−like macro have been identified, argument substitution occurs. Unless it is preceded by a *#* token (**"The # Operator"**) or is adjacent to a *##* token (**"The ## Operator"**), a parameter in the replacement list is replaced by the corresponding argument after any macros in the argument have been expanded (**"Rescanning and Further Replacement"**).

For example, given the macro definitions

```
#define index_mask      0XFF00
#define extract(word,mask)      word & mask
```

the call

```
index = extract(packed_data,index_mask);
```

expands to

```
index = packed_data & 0XFF00;
```

In both forms the replacement string is rescanned for more defined identifiers (**"Rescanning and Further Replacement"**).

The *#* Operator

If an occurrence of a parameter in a replacement token sequence is immediately preceded by a *#* token, the parameter and the *#* operator will be replaced in the expansion by a string literal containing the spelling of the corresponding argument. A ** character is inserted in the string literal before each occurrence of a ** or a *" "* within or delimiting a character constant or string literal in the argument. For example,

```
#define path(logid,cmd)    "/usr/" #logid "/bin/" #cmd
#define joe                joseph
```

the call

```
char* mytool=path(joe,readmail);
```

yields

```
char* mytool="/usr/" "joe" "/bin/" "readmail";
```

which is later concatenated (**"Phases of Preprocessing"**) to become

```
char* mytool="/usr/joe/bin/readmail";
```

The *##* Operator

If a *##* operator appears in a replacement token sequence between two tokens, first if either of the adjacent tokens is a parameter it is replaced, then the *##* operator and any white space surrounding it are deleted. The effect of the *##* operator, therefore, is concatenation. For example,

```
#define inherit(basenum) public Pubbase ## basenum, \
private Privbase ## basenum
```

the call

```
class D : inherit(1) { };
```

yields

```
class D : public Pubbase1, private Privbase1 { };
```

Any macros in the replaced tokens adjacent to the *##* are not available for further expansion, but the result of the concatenation is. For example,

```
#define concat(a)      a ## ball
#define base           B
#define baseball       sport
```

the call

```
concat(base)
```

yields

```
sport
```

and *not*

```
Bball
```

Rescanning and Further Replacement

After all parameters in the replacement list have been replaced, the resulting list is rescanned for more macros to replace. If the name of the macro being replaced is found during this scan or during subsequent rescanning, it is not replaced.

A completely replaced macro expansion is not interpreted as a preprocessing directive, even if it appears to be one.

Scope of Macro Names and #undef

Once defined, a preprocessor identifier remains defined and in scope (independent of the scoping rules of C++) until the end of the translation unit or until it is undefined in a *#undef* directive.

A *#undef* directive has the form

```
#undef identifier
```

and causes the identifier's preprocessor definition to be forgotten. If the specified identifier is not currently defined as a macro name, the *#undef* is ignored.

File Inclusion

A control line of the form

```
#include <filename>
```

causes the replacement of that line by the entire contents of the file *filename.* The named file is searched for in an implementation–dependent sequence of places.

Similarly, a control line of the form

```
#include "filename"
```

causes the replacement of that line by the contents of the file *filename,* which is searched for first in an implementation–dependent manner. If this search fails, the file is searched for as if the directive had been of the form

```
#include <filename>
```

Neither the new–line character nor > may appear in *filename* delimited by <and >. If any of the characters *´,* ,\ or *"*, or either of the sequences */** or *//* appear in such a *filename* the behavior is undefined.

Neither the new–line character nor *" "* may appear in a *filename* delimited by a *" "* pair, although *>* may appear. If either of the characters *´ or* \or either of the sequences */** or *//* appear in such a *filename* the behavior is undefined.

If a directive appears of the form

```
#include token–string
```

not matching either of the forms given above, the preprocessing tokens within *token–string* will be processed as normal text. The resulting directive must match one of the forms defined above and will be treated as such.

A *#include* directive may appear within a file that is being processed as a result of another *#include* directive.

An implementation may impose a limit on the depth of nesting of *#include* directives within source files that have been read while processing a *#include* directive in another source file.

Conditional Compilation

The preprocessor allows conditional compilation of source code. The syntax for conditional compilation follows:

conditional:
  if–part elif–partsopt else–partopt 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

The constant expression in the *#if* and *#elif*s (if any) are evaluated in the order in which they appear until one of the expressions evaluates to a non−zero value. C++ statements following a line with a zero value are not compiled, nor do preprocessor directives following such a line have any effect. When a directive with a non−zero value is found, the succeeding *#elif*s, and *#else*'s, together with their associated text (C++ statements and preprocessor directives) are ignored. The text associated with the successful directive (the first whose constant expression is non−zero) is preprocessed and compiled normally. If the expressions associated with the *#if* and all *#elif*s evaluate to zero, then the text associated with the *#else* (if any) is treated normally.

Within the *constant−expression* in a *#if* or *#elif*, a unary operator *defined* can be used in either of the forms

```
defined identifier
```

or

```
defined (identifier)
```

When applied to an identifier, its value is 1 if that identifier has been defined with a *#define* directive and not later undefined using *#undef*; otherwise its value is 0. The identifier *defined* itself may not be undefined or redefined.

After any *defined* operators are evaluated, any remaining preprocessor macros appearing in the constant expression will be replaced as described in **"Macro Definition and Expansion"** . The resulting expression must be an integral constant expression as defined in **"Constant Expressions"** , except that types *int* and *unsigned int* are treated as *long* and *unsigned long* respectively, and it may not contain a cast, a *sizeof* operator, or an enumeration constant.

A control line of the form

```
#ifdef identifier
```

is equivalent to

```
#if defined identifier
```

A line of the form

```
#ifndef identifier
```

is equivalent to

```
#if !defined identifier
```

Conditional compilation constructs may be nested. An implementation may impose a limit on the depth of nesting of conditional compilation constructs.

Line Control

For the benefit of programs that generate C++ code, a line of the form

```
#line constant "filename"opt
```

sets the predefined macro *__LINE__* (**"Predefined Names"**), for purposes of error diagnostics or symbolic debugging, such that the line number of the next source line is considered to be the given constant, which must be a decimal integer. If *"filename"* appears, *__FILE__* (**"Predefined Names"**), is set to the file named. If *"filename"* is absent the remembered file name does not change.

Macros appearing on the line are replaced before the line is processed.

Error Directive

A line of the form

```
#error token-string
```

causes the implementation to generate a diagnostic message that includes the given token sequence.

Pragmas

A line of the form

```
#pragma token-string
```

causes an implementation−dependent behavior when the token sequence is of a form recognized by the implementation. An unrecognized pragma will be ignored.

Null Directive

The null preprocessor directive, which has the form

```
#
```

has no effect.

Predefined Names

Certain information is available during compilation through predefined macros.

__LINE__	A decimal constant containing the current line number in the C++ source file.
__FILE__	A string literal containing the name of the source file being compiled.
__DATE__	A string literal containing the date of the translation, in the form *"Mmm dd yyyy"*, or *"Mmm d yyyy"* if the value of the date is less than 10 .
__TIME__	A string literal containing the time of the translation, in the form *"hh:mm:ss"*.

In addition, the name *__cplusplus* is defined when compiling a C++ program.

These names may not be undefined or redefined.

__LINE__ and *__FILE__* can be set by the *#line* directive (**"Line Control"**).

Whether *__STDC__* is defined and, if so, what its value is are implementation dependent.

Appendix A

Grammar Summary

This Appendix contains the following subsections:

- **"Keywords"**
- **"Expressions"**
- **"Declarations"**
- **"Declarators"**
- **"Class Declarations"**
- **"Statements"**
- **"Preprocessor"**
- **"Templates"**
- **"Exception Handling"**

This appendix is not part of the C++ reference manual proper and does not define C++ language features.

This summary of C++ syntax is intended to be an aid to comprehension. It is not an exact statement of the language. In particular, the grammar described here accepts a superset of valid C++ constructs. Disambiguation rules (**"Ambiguity Resolution"** , **"Specifiers"** , **"Ambiguities"**) must be applied to distinguish expressions from declarations. Further, access control, ambiguity, and type rules must be used to weed out syntactically valid but meaningless constructs.

Keywords

New context–dependent keywords are introduced into a program by *typedef* (**"The typedef Specifier"**), class (**Chapter 9, "Classes"**), enumeration (**"Enumeration Declarations"**), and *template* (**Chapter 14, "Templates"**) declarations.

class–name:
  identifier

enum–name:
  identifier

typedef–name:
  identifier

Note that a *typedef–name* naming a class is also a *class–name* (**"Class Names"**).

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:
pm−expression
*multiplicative−expression * pm−expression*
multiplicative−expression / pm−expression
multiplicative−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 placement$_{opt}$ new−type−name new−initializer$_{opt}$
::$_{opt}$ new placement$_{opt}$ (type−name) new−initializer$_{opt}$

placement:
(expression−list)

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

new−declarator:
** cv−qualifier−list$_{opt}$ new−declarator$_{opt}$*

*complete−class−name :: * cv−qualifier−list$_{opt}$ new−declarator$_{opt}$*
new−declarator$_{opt}$ [expression]

new−initializer:
(initializer−list$_{opt}$)

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

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

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:
qualified−class−name :: name

literal:
integer−constant
character−constant
floating−constant
string−literal

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

elaborated−type−specifier:
class−key identifier
class−key class−name
enum\ 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

asm−declaration:
  asm (string−literal) ;

Declarators

declarator−list:
  init−declarator
  declarator−list , init−declarator

init−declarator:
  declarator initializer$_{opt}$

declarator:
  dname
  ptr−operator declarator
  declarator (argument−declaration−list) cv−qualifier−list$_{opt}$
  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

dname:
  name
  class−name
  ~ class−name
  typedef−name
  qualified−type−name

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)

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

function−definition:

decl−specifiers$_{opt}$ declarator ctor−initializer$_{opt}$ fct−body

fct−body:
compound−statement

initializer:
= assignment−expression
= { initializer−list ,$_{opt}$ }
(expression−list)

initializer−list:
assignment−expression
initializer−list , assignment−expression
{ initializer−list ,$_{opt}$ }

Class Declarations

class−specifier:
class−head { member−list$_{opt}$ }

class−head:
class−key identifier$_{opt}$ base−spec$_{opt}$
class−key class−name base−spec$_{opt}$

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

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
+ − * / % ^ & | ~
! = < > += −= *= /= %=
^= &= |= << >> >>= <<= == !=
<= >= && || ++ −− , −>* −>
() []

Statements

statement:
labeled−statement
expression−statement
compound−statement
selection−statement
iteration−statement
jump−statement
declaration−statement

labeled−statement:
identifier : statement
case constant−expression : statement
default : statement

expression−statement:
expressionopt ;

compound−statement:
{ statement−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

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

Templates

template−declaration:
template < template−argument−list > declaration

template−argument−list:
template−argument
template−argument−list , template argument

template−argument:
type−argument
argument−declaration

type−argument:
class identifier

template−class−name:
template−name < template−arg−list >

template−arg−list:
template−arg
template−arg−list , template−arg

template−arg:
  expression
  type−name

Exception Handling

try−block:
  try compound−statement handler−list

handler−list:
  handler handler−list$_{opt}$

handler:
  catch (exception−declaration) compound−statement

exception−declaration:
  type−specifier−list declarator
  type−specifier−list abstract−declarator
  type−specifier−list
  ...

throw−expression:
  throw expression$_{opt}$

exception−specification:
  throw (type−listopt)

type−list:
  type−name
  type−list , type−name

Appendix B
Compatibility

This Appendix contains the following subsections:

- **"Extensions"**
- **"C++ and ANSI C"**
- **"Anachronisms"**

This appendix is not part of the C++ reference manual proper and does not define C++ language features.

C++ is based on C (K&R78) and adopts most of the changes specified by the ANSI C standard. Converting programs among C++, K&R C, and ANSI C may be subject to vicissitudes of expression evaluation. All differences between C++ and ANSI C can be diagnosed by a compiler. With the following three exceptions, programs that are both C++ and ANSI C have the same meaning in both languages:

In C, *sizeof('a')* equals *sizeof(int)*; in C++, it equals *sizeof(char)*.

In C, given

```
enum e { A };
```

sizeof(A) equals *sizeof(int)*; in C++, it equals *sizeof(e)*, which need not equal *sizeof(int)*.

A structure name declared in an inner scope can hide the name of an object, function, enumerator, or type in an outer scope. For example,

```
int x[99];
void f()
{
      struct x { int a; };
      sizeof(x);  /* size of the array in C */
      /* size of the struct in C++ */
```

```
}
```

Extensions

This section summarizes the major extensions to C provided by C++.

C++ Features Available in 1985

This subsection summarizes the extensions to C provided by C++ in the 1985 version of this manual:

The types of function arguments can be specified (**"Functions"**) and will be checked (**"Function Call"**). Type conversions will be performed (**"Explicit Type Conversion"**). This is also in ANSI C.

Single−precision floating point arithmetic may be used for *float* expressions; **"Fundamental Types"** and **"Float and Double"** . This is also in ANSI C.

Function names can be overloaded; **Chapter 13, "Overloading"** .

Operators can be overloaded; **"Overloaded Operators"** .

Functions can be inline substituted; **"Function Specifiers"** .

Data objects can be *const*; **"Type Specifiers"** . This is also in ANSI C.

Objects of reference type can be declared; **"References"** **"References"** .

A free store is provided by the *new* and *delete* operators; **"New"** , **"Delete"** .

Classes can provide data hiding (**Chapter 11, "Member Access Control"**), guaranteed initialization (**"Constructors"**), user−defined conversions (**"Conversions"**), and dynamic typing through use of virtual functions (**"Virtual Functions"**).

The name of a class or enumeration is a type name; **"Class Names"** .

A pointer to any non− *const* and non−*volatile* object type can be assigned to a *void**; **"Pointer Conversions"** . This is also in ANSI C.

A pointer to function can be assigned to a *void**; **"Pointer Conversions"** .

A declaration within a block is a statement; **"Declaration Statement"** .

Anonymous unions can be declared; **"Unions"** .

C++ Features Added Since 1985

This subsection summarizes the major extensions of C++ since the 1985 version of this manual:

A class can have more than one direct base class (multiple inheritance); **"Multiple Base Classes"** .

Class members can be *protected*; **Chapter 11, "Member Access Control"** .

Pointers to class members can be declared and used; **"Pointers to Members"** , **"Pointer − to − Member Operators"** .

Operators *new* and *delete* can be overloaded and declared for a class; **"New"** , **"Delete"** , **"Free Store"** . This allows the "assignment to *this*" technique for class specific storage management to be removed to the anachronism section; **"Anachronisms"** .

Objects can be explicitly destroyed; **"Destructors"** .

Assignment and initialization are defined as memberwise assignment and initialization; **"Copying Class Objects"** .

The *overload* keyword was made redundant and moved to the anachronism section; **"Anachronisms"** .

General expressions are allowed as initializers for static objects; **"Initializers"** .

Data objects can be *volatile*; **"Type Specifiers"** . Also in ANSI C.

Initializers are allowed for *static* class members; **"Static Members"** .

Member functions can be *static*; **"Static Members"** .

Member functions can be *const* and *volatile*; **"The this Pointer"** .

Linkage to non−C++ program fragments can be explicitly declared; **"Linkage Specifications"** .

Operators *−>*, *−>**, and *,* can be overloaded; **"Overloaded Operators"** .

Classes can be abstract; **"Abstract Classes"** .

Prefix and postfix application of *++* and *−−* on a user−defined type can be distinguished.

Templates; **Chapter 14, "Templates"** .

Exception handling; **Chapter 15, "Exception Handling"** .

C++ and ANSI C

In general, C++ provides more language features and fewer restrictions than ANSI C so most constructs in ANSI C are legal in C++ with their meanings unchanged. The exceptions are:

ANSI C programs using any of the C++ keywords

```
asm       catch      class     delete     friend
inline    new        operator  private    protected
public    template   try       this       virtual
throw
```

as identifiers are not C++ programs; **"Keywords"** .

Though deemed obsolescent in ANSI C, a C implementation may impose Draconian limits on the length of identifiers; a C++ implementation is not permitted to; **"Identifiers"** .

In C++, a function must be declared before it can be called; **"Function Call"** .

The function declaration *f()*; means that *f* takes no arguments (**"Functions"**); in C it means that *f* can take any number of arguments of any type at all. Such use is deemed obsolescent in ANSI C.

In ANSI C a global data object may be declared several times without using the *extern* specifier; in C++ it must be defined exactly once; **"Program and Linkage"** .

In C++, a class may not have the same name as a typedef declared to refer to a different type in the same scope; **"Class Names"** .

In ANSI C a *void** may be used as the right−hand operand of an assignment to or initialization of a variable of any pointer type; in C++ it may not; **"Type Specifiers"** .

C allows jumps to bypass an initialization; C++ does not.

In ANSI C, a global *const* by default has external linkage; in C++ it does not; **"Program and Linkage"** .

"Old style" C function definitions and calls of undeclared functions are considered anachronisms in C++ and may not be supported by all implementations;**"Old Style Function Definitions"** . This is deemed obsolescent in ANSI C.

A *struct* is a scope in C++ (**"Scopes"**); in ANSI C a *struct*, an enumeration, or an enumerator declared in a *struct* is exported to scope enclosing the *struct*.

Assignment to an object of enumeration type with a value that is not of that enumeration type is considered an anachronism in C++ and may not be supported by all implementations; **"Enumeration Declarations"** . ANSI C recommends a warning for such assignments.

Surplus characters are not allowed in strings used to initialize character arrays; **"Character Arrays"** .

The type of a character constant is *char* in C++ (**"Character Constants"**) and *int* in C.

The type of an enumerator is the type of its enumeration in C++ (**"Enumeration Declarations"**) and *int* in C.

In addition, the ANSI C standard allows conforming implementations to differ considerably; this may lead to further incompatibilities between C and C++ implementations. In particular, some C implementations may consider certain incompatible declarations legal. C++ requires consistency even across compilation boundaries; **"Declarations and Definitions"** .

How to Cope

In general, a C++ program uses many features not provided by ANSI C. For such a program, the minor differences of **"C++ and ANSI C"** don't matter since they are dwarfed by the C++ extensions. Where ANSI C and C++ need to share header files, care must be taken so that such headers are written in the common subset of the two languages.

No advantage must be taken of C++ specific features such as classes, overloading, and so on.

A name should not be used both as a structure tag and as the name of a different type.

A function *f* taking no arguments should be declared *f(void)* and not simply *f()*.

Global *consts* must be declared explicitly *static* or *extern*.

Conditional compilation using the C++ predefined name *__cplusplus* may be used to distinguish information to be used by an ANSI C program from information to be used by a C++ program.

Functions that are to be callable from both languages must be explicitly declared to have C linkage.

Anachronisms

The extensions presented here may be provided by an implementation to ease the use of C programs as C++ programs or to provide continuity from earlier C++ implementations. Note that each of these features has undesirable aspects. An implementation providing them should also provide a way for the user to ensure that they do not occur in a source file. A C++ implementation is not obliged to provide these features.

The word *overload* may be used as a *decl−specifier* (**Chapter 7, "Declarations"**) in a function declaration or a function definition. When used as a *decl−specifier, overload* is a reserved word and cannot also be used as an identifier.

The definition of a static data member of a class for which initialization by default to all zeros applies (**"Initializers"** , **"Static Members"**) may be omitted.

An old style (that is, pre−ANSI C) C preprocessor may be used.

An *int* may be assigned to an object of enumeration type.

The number of elements in an array may be specified when deleting an array of a type for which there is no destructor; **"Delete"** .

A single function *operator++()* may be used to overload both prefix and postfix *++* and a single function *operator−−()* may be used to overload both prefix and postfix *−−*; **"Class Member Access"** .

Old Style Function Definitions

The C function definition syntax

old−function−definition:
    decl−specifiers$_{opt}$ old−function−declarator declaration−list$_{opt}$ fct−body

old−function−declarator:
        declarator (parameter−list$_{opt}$)

parameter−list:
        identifier
        parameter−list , identifier

For example,

```
max(a,b) int b; { return (a<b) ? b : a; }
```

may be used. If a function defined like this has not been previously declared its argument type will be taken to be (...), that is, unchecked. If it has been declared its type must agree with that of the declaration.

Class member functions may not be defined with this syntax.

Old Style Base Class Initializer

In a *mem−initializer* (**"Initializing Bases and Members"**), the *class−name* naming a base class may be left out provided there is exactly one immediate base class. For example,

```
class B {
     // ...
public:
     B (int);
};
class D : public B {
     // ...
     D(int i) : (i) { /* ... */ }
};
```

causes the *B* constructor to be called with the argument *i*.

Assignment to this

Memory management for objects of a specific class can be controlled by the user by suitable assignments to the *this* pointer. By assigning to the *this* pointer before any use of a member, a constructor can implement its own storage allocation. By assigning a zero value to *this*, a destructor can avoid the standard deallocation operation for objects of its class. Assigning a zero value to *this* in a destructor also suppressed the implicit calls of destructors for bases and members. For example,

```
class Z {
     int z[10];
     Z()  { this = my_allocator( sizeof(Z) ); }
     ~Z() { my_deallocator( this ); this = 0; }
};
```

On entry into a constructor, *this* is non−zero if allocation has already taken place (as it will have for *auto*, *static*, and member objects) and zero otherwise.

Calls to constructors for a base class and for member objects will take place (only) after an assignment to *this*. If a base class's constructor assigns to *this*, the new value will also be used by the derived class's constructor (if any).

Note that if this anachronism exists either the type of the *this* pointer cannot be a **const* or the enforcement of the rules for assignment to a constant pointer must be subverted for the *this* pointer.

Cast of Bound Pointer

A pointer to member function for a particular object may be cast into a pointer to function, for example, *(int(*)())p−>f*. The result is a pointer to the function that would have been called using that member function for that particular object. Any use of the resulting pointer is as ever undefined.

Non-nested Classes

Where a class is declared within another class and no other class of that name is declared in the program that class can be used as if it was declared outside its enclosing class (exactly as a C *struct*). For example,

```
struct S {
     struct T {
        int a;
     };
     int b;
};
struct T x;      // meaning 'S::T x;'
```