MetaWare® C/C++

Language Reference

4089-015

MetaWare® C/C++ Language Reference

© 1998-2007 ARC® International. All rights reserved.

ARC International

North America
3590 N. First Street, Suite 200
San Jose, CA 95134 USA
Tel. +1-408-437-3400
Fax +1-408-437-3401

Europe
Verulam Point
Station Way
St Albans, AL1 5HE
UK
Tel.+44 (0) 20-8236-2800
Fax +44 (0) 20-8236-2801

www.ARC.com

ARC Confidential and Proprietary Information

Notice

This document, material and/or software contains confidential and proprietary information of ARC International and is protected by copyright, trade secret, and other state, federal, and international laws, and may be embodied in patents issued or pending. Its receipt or possession does not convey any rights to use, reproduce, disclose its contents, or to manufacture, or sell anything it may describe. Reverse engineering is prohibited, and reproduction, disclosure, or use without specific written authorization of ARC International is strictly forbidden. ARC and the ARC logotype are trademarks of ARC International.

The product described in this manual is licensed, not sold, and may be used only in accordance with the terms of a License Agreement applicable to it. Use without a License Agreement, in violation of the License Agreement, or without paying the license fee is unlawful.

Every effort is made to make this manual as accurate as possible. However, ARC International shall have no liability or responsibility to any person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this manual, including but not limited to any interruption of service, loss of business or anticipated profits, and all direct, indirect, and consequential damages resulting from the use of this manual. ARC International's entire warranty and liability in respect of use of the product are set forth in the License Agreement.

ARC International reserves the right to change the specifications and characteristics of the product described in this manual, from time to time, without notice to users. For current information on changes to the product, users should read the “readme” and/or “release notes” that are contained in the distribution media. Use of the product is subject to the warranty provisions contained in the License Agreement

Licensee acknowledges that ARC International is the owner of all Intellectual Property rights in such documents and will ensure that an appropriate notice to that effect appears on all documents used by Licensee incorporating all or portions of this Documentation.

The manual may only be disclosed by Licensee as set forth below.

- Manuals marked “ARC Confidential & Proprietary” may be provided to Licensee’s subcontractors under NDA. The manual may not be provided to any other third parties, including manufacturers. Examples—source code software, programmer guide, documentation.
- Manuals marked “ARC Confidential” may be provided to subcontractors or manufacturers for use in Licensed Products. Examples—product presentations, masks, non-RTL or non-source format.
- Manuals marked “Publicly Available” may be incorporated into Licensee's documentation with appropriate ARC permission. Examples—presentations and documentation that do not embody confidential or proprietary information.

U.S. Government Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and/or in similar or successor clauses in the FAR, or the DOD or NASA FAR Supplement.

CONTRACTOR/MANUFACTURER IS ARC International I. P., Inc., 3590 N. First Street, Suite 200, San Jose, California 95134.

Trademark Acknowledgments

ARC-Based, ARC-OS Changer, ARCangel, ARCform, ARChitect, ARCompact, ARCtangent, BlueForm, CASSEIA, High C/C++, High C++, iCon186, IPShield, MetaDeveloper, the MQX Embedded logo, Precise Solution, Precise/BlazeNet, Precise/EDS, Precise/MFS, Precise/MQX, Precise/MQX Test Suites, Precise/MQXsim, Precise/RTCS, Precise/RTCSsim, RTCS, SeeCode, TotalCore, Turbo186, Turbo86, V8 µ-RISC, V8 microRISC, and VAutomation are trademarks of ARC International. ARC, the ARC logo, High C, MetaWare, MQX and MQX Embedded are registered under ARC International. All other trademarks are the property of their respective owners.

4089-015 August 2007

Contents

Chapter 1 — Before you begin

In This Chapter

Technical Support

We encourage customers to visit our Technical Support site for technical information before submitting support requests via telephone. The Technical Support site is richly populated with the following:

ARCSolve — Frequently asked questions

ARCSpeak — Glossary of terms

White papers — Technical documents

Datasheets — ARC product flyers

Known Issues — Known bugs and workarounds.

Training — Future ARC Training sessions

Support — Entry point to sending in a query

Downloads — Access to useful scripts and tools

You can access this site via our corporate website at http://www.ARC.com, or you can access the site directly: http://support.ARC.com.

Note that you must register before you can submit a support request via our Extranet site.

About this book

This *Language Reference* compares ANSI Standard C and the C++ language, discusses how to migrate from C to MetaWare C++, describes MetaWare C/C++ extensions to ANSI Standard C and the C++ language, and describes the syntax and semantics of the C++ language.

This manual does not discuss features of MetaWare C/C++ that conform to the ANSI C Standard, because those are effectively documented by the Standard.

This book contains the following topics:

- Chapter 14 — Special cast notations
- Chapter 15 — C++ type information
- Appendix A — Template instantiation issues

Where to go for more information

Related C and C++ publications

For information about the C language, consult the following:

ANSI Committee X3J11. *American National Standard for Information Systems — Programming Language C*. Doc. No. X3.159. CBEMA, 1989.

Harbison, Samuel P., and Guy L. Steele Jr. *C: A Reference Manual*. Second Edition. Prentice-Hall, Inc., 1987.

Kernigan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Prentice Hall, Inc., 1988.

Plauger, P. J., and J. Brodie. *Standard C*. Microsoft Press, 1989.

For information about C++, see the following:

ISO Committee SC22/WG21.

Stroustrup, B. *The C++ Programming Language*, second edition. Addison-Wesley Publishing Company, 1991.

Stroustrup, Bjarne, AT&T Bell Laboratories. *The Design and Evolution of C++*. Addison-Wesley Publishing Company, 1994.

Stroustrup, B. and Ellis, M. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990.

For an overview of object-oriented programming, you might find the following book helpful:

Booch, G. *Object-Oriented Design with Applications*. Benjamin/Cummings Publishing Company, Inc., 1991

Documents in the MetaWare C/C++ Toolset

For information about the manuals in your MetaWare C/C++ document set, see the “Where to Go for More Information” section in the “Before you begin” chapter in the *MetaWare C/C++ Programmer’s Guide*.

Document conventions

Notes

Notes point out important information.

> **NOTE** Names of command-line options are case-sensitive

Cautions

Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware.

CAUTION Comments in assembly code can cause the preprocessor to fail if they contain C preprocessing tokens such as **#if** or **#end**, C comment delimiters, or invalid C tokens.

Language standards

The MetaWare C/C++ compiler conforms to the American National Standards Institute (ANSI) X3.159 standard for the C programming language. MetaWare C also includes selected features from the ISO/ANSI C99 Standard.

MetaWare C++ implements all of the C++ programming language as defined in *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup. MetaWare C++ incorporated language changes to track the ANSI X3J16 standard as it emerged.

MetaWare C/C++ includes many extensions not found in other C and C++ products. For information about these extensions, see Extensions on page 29 and Iterators in MetaWare C/C++ on page 65.

Chapter 2 — Contrasting C and C++

In This Chapter

Introduction

C++ is not quite a superset of ANSI Standard C. Some of the important differences are listed in this chapter. If you have been programming in C, you should keep these differences in mind when you write C++ programs or add C++ features to existing C programs. See Migrating from C to C++ on page 57 for more information about adding C++ functionality to your C programs.

Comments

C++ recognizes two forms of comment, "/*...*/" and '//'.

The first form, familiar to C programmers, is shown in the following example:

```
/* Any sequence of characters, except: */
```

This comment starts with the characters '/*' and ends with the characters '*/'. Comments of this type cannot be nested:

```
/* This is a valid comment. */
/* This is not /* a valid comment. */ */
```

The second form of a comment in C++ starts with two slashes (//) with no space between them, as shown in the following examples:

```
// This whole line is a comment.
main ()   // This part of the line is a comment.
int i = 3 / / Oops, cannot have space between slashes.
```

The '//' form of comment has no ending delimiter, and extends to the end of the line.

NOTE If a line containing a //-style comment ends in a backslash (\), the next line is a continuation of the comment.

NOTE The string // has no lexical meaning inside /*...*/ comments. Therefore, if you use //-style comments for all normal commenting in C++ code, you can use /*...*/-style comments to comment out any section of code, even if it contains //-style comments.

Data types

ANSI-Standard C data types and C++ data types differ in the following ways:

- In ANSI Standard C, a global **const** has external linkage by default; in C++ it does not.
- In ANSI Standard C, single-character constants have data type **int**; in C++ they have type **char**. If you write **sizeof(***char_const***)**, where *char_const* is any single-character constant, in C++ you get the value 1 (one), while in ANSI Standard C you get **sizeof(int)**.
- The type of an enumerator is **int** in ANSI Standard C; in C++ it is the type of its enumeration. (See Type specifiers in enum types on page 17.)
- In ANSI Standard C a **void*** can be assigned to a pointer of any type. In C++, a **void*** can be assigned only to another **void***.

Type specifiers in enum types

In ANSI Standard C, **enum** names are not type names. In C++, if an identifier appears in an enumeration definition after the **enum** keyword and before the enumeration list, that name is declared as a type name, and can be used as a subsequent type specifier. In the following example, `color` is an enumeration type, and you can declare `mycolor` to be of type `color`:

```
enum color { red, green, blue, violet };
color mycolor;
mycolor = blue;
```

C++ declaration syntax allows type specifiers without any declarator. In the preceding example, the **enum** definition is a declaration specifier, and no declarator is supplied.

Omitting an identifier between the **enum** keyword and the enumeration list is done primarily in **typedef** declarations. In this case, any identifier following the enumeration list can be used as a type name. For example:

```
main(){
   typedef enum  { red,
                   green  = 3*2,
                   blue,
                   violet = 6,
                   puce } tname;
   // Identifer ^ is omitted, but you can
   // use tname as a type name:
   tname mycolor;
   mycolor = blue;
   }
```

MetaWare C/C++ allows you to specify **long** and **short** enumeration types. This is an extension to the C and C++ languages; see Short enumerators and long enumerators on page 48.

Structures

In C++, a **struct** defines a type. The following code is valid in C++ but not in ANSI Standard C:

```
struct X { int n; };
X a1;
```

In C++, a **struct** also defines a scope; in ANSI Standard C a **struct**, as well as an enumeration or an enumerator declared in a **struct**, is exported to the enclosing scope (see Scope access on page 19).

Unions in C++

In C++, a **union** has the following restrictions:

- A **union** cannot have virtual functions, because a virtual-function pointer cannot share memory with other user-declared union members.
- A **union** cannot have base classes, because there is no space for placing inherited base class members.
- A **union** cannot be used as a base class.
- A **union** cannot have a member whose type is a class with a constructor, destructor, or user-defined assignment operator. (Imagine two members with constructors; which overlapping member gets constructed when a **union** object is created?)

Type casting

C++ allows you to perform certain type casts that ANSI Standard C does not allow. C++ also imposes certain restrictions that ANSI Standard C does not.

Pointers to void

In ANSI Standard C, a pointer to **void** can be converted to a pointer to any other data type. This is not true in C++. A pointer to **void** must be explicitly cast to the other type.

Explicit conversion by casting

C++ allows the following explicit conversions of C++-specific elements by casting:

- You can convert an object to a class type only if you have declared an appropriate constructor or conversion function. The argument to the cast is taken as an argument to the constructor or conversion function.
- You can cast a pointer to a member to a different pointer-to-member type if both types are pointers to members of the same class.
- A pointer to a member of a derived class can be converted to a pointer to a member of a base class if the conversion in the other direction is allowed implicitly. The effect of converting from a pointer to a member of a derived class to a pointer to a member of a base class is the semantic inverse of the other conversion, and assumes that the first pointer points to a member of the base class; otherwise the semantics are undefined.
- You can cast a pointer to a base class to a pointer to a derived class, assuming the conversion in the other direction is implicitly allowed, and assuming that the base class is not a virtual base of the derived class. The effect of converting from a pointer to a base class to a pointer to a derived class is the semantic inverse of the other conversion, and assumes that the first pointer points to an object that is a sub-object of the derived class.

CAUTION Avoid casting from one pointer-to-class to another pointer-to-class, if the classes are not yet defined. If you do so, the cast does not change the pointer value. However, the same conversion occurring later might in fact add a delta to the pointer if one class is a base of the other, to reflect the location of the base object as a sub-object of the derived object. This difference in code generated for the two identical casts can produce anomalies.

Operators

Table 1 C++ operators lists operators that apply only to C++.

Table 1 C++ operators

Operator	Description
delete	Deallocates dynamic memory (unary)
new	Allocates dynamic memory (unary)
::	Global scope-resolution operator: accesses global names hidden in the current scope (unary)
::	Class scope-resolution operator: accesses names not visible in the current scope (binary)

For information about precedence of these operators, refer to the ANSI Draft C++ Standard document X3J16.

Scope access

You use the scope-resolution operator **::** to access names that are not in the current scope. For example:

```
struct s {
  typedef float T;
  };
s::T x;         // Use :: to get T within s.
```

You can use the **struct**, **class**, **union**, or **enum** keyword in a reference to a previous **class** or **enum** definition, although most of the time this is unnecessary. For example:

```
struct s {
   struct t { int z; };
   };
struct s::t x;      // x is of type s::t.
s::t y;             // This works as well.
```

The only time you must use the prefixing **struct**, **class**, **union**, or **enum** keyword is when the name you are accessing is hidden by a "competing" declaration. For example:

```
struct s {
   struct t { int z; };
   int t;           // This hides the name t
   };               //  of the struct.
struct s::t x;      // x is of type struct s::t.
s::t y;             // This no longer works, and
                    //  draws a syntax error message.
```

NOTE In general, in C++ you almost never have to use the reference-to-previous form of type specifier; you can just use the identifier instead. In ANSI Standard C, as opposed to C++, you must always use the reference-to-previous form. C does not introduce class and **enum** names as type names; C++ does.

The global and class scope-resolution operators

C++ has two scope-resolution operators.

The *global scope-resolution operator* is unary. You use it to access a global name hidden by an identical local name within the current scope:

```
int i;           // Global variable i
void init_array() {
   int i;        // Local variable i
   for(i = 0; i < MAX; i++) {
      ...
      }
   ::i = i - 1;  // Assign the value of local i to
   }             //  global i.
```

The *class scope-resolution operator* is binary. You use it to refer to a name within a specified class:

```
class circle {
   int x, y;
   double radius;
public:
   void get_coords(int a, int b);
```

```
    ...
    };
void circle::get_coords(int a, int b) {...}
```

The new operator

The **new** operator creates an object of type `type_name` and returns a pointer to the object.

The syntax for **new** is one of the following:

```
[::]new [other_args]type_name [(initializer)]
[::]new [other_args]( type_name )[(initializer )]
```

The components have the following meanings:

other_args	is an expression or list of expressions separated by commas, supplying additional arguments to **new**. You can use this only when you have declared an **operator new()** function with appropriate arguments. For example, **new** results in a call to **operator new(sizeof(T))**, whereas **new(a, b+3)** results in a call to **operator new(sizeof(T), a, b+3)**.
type_name	is any fundamental or derived type name.
initializer	is an expression or list of expressions appropriate to initialize the object you are allocating.

If *type_name* refers to a class type or an enumerated type, the type must have been previously declared. You cannot declare a class or enumerated type in a **new** expression. *type_name* cannot contain **const** or **volatile** qualifiers in its declaration.

If **new** is unable to allocate the requested storage, it returns **0** (zero).

Allocating arrays

When you allocate storage for an array, all dimensions except the first must be compile-time constant expressions. **new** allocates storage for an object of *type_name* equal to the size in bytes for **sizeof(***type_name***)**.

```
int i = 3;
// Allocate an int [3][6][9] array:
int ar = new int [i][6][9];
```

The following example demonstrates one way to allocate an array of type **char**:

```
char *str = "hello";
// A common use of new:
char *p = new char [strlen(str)+1];
```

Initializing objects with new

If you do not supply *initializer*, the allocated object has an undefined value, unless there is a constructor for it. Initialization is performed only when **new** returns a non-zero value. You cannot initialize objects of array type. You can use **new** to create an object that is an array of a class type only if the class type defines a default constructor. If there is a default constructor for the class type, the constructor is called for each element of the array.

new invokes constructors for types that have declared constructors. The order of evaluation for both the **new** operator and arguments to constructors is undefined.

If the type allocated is a class, the class-specific **new** operator, if present, is used. See Controlling storage allocation on page 130 for how to declare a class-specific **new** operator.

```
int *i;
class init{
   float f;
   char  c;
   };
init *myclass;
// Allocate an integer and initialize it to 7:
i = new int(7);
// Allocate a class object and initialize it:
myclass = new (init);
```

If the leading **::** is present, the global operator **new** is used instead of the class-specific **new** operator.

The delete operator

The **delete** operator destroys an object that was created dynamically with **new**. The type of the result of a **delete** expression is **void**. The operand of **delete** must be a pointer that was created by **new**. The effect of using any other operand with **delete** is undefined. You can always delete a pointer whose value is 0 (zero), however, with no error.

You cannot access an object after it has been deleted. The effect of such an operation is undefined. A pointer to a **const** object can never be the operand of **delete**. If the pointer operand of **delete** points to an object that has a destructor, **delete** invokes the destructor.

This is the syntax for **delete**:

```
[::]delete [[ ]] cast_expression
```

cast_expression is one of the following:

```
unary_expression
( type_name ) unary_expression
```

Deleting arrays

The form **delete[]** *cast_expression* is for deleting arrays. You must use this notation when deleting an array, or the result is undefined. You must not use this notation for deleting non-array objects, because doing so also gives undefined results.

The class-specific delete operator

If the deleted expression points to a class object, the class-specific **delete** operator, if present, is used. See Controlling storage allocation on page 130 for how to declare a class-specific **delete** operator. The optional leading **::** means that the global operator **delete** is used instead of the class-specific **delete** operator.

Keywords

Table 2 C++ keywords describes words can be used as identifiers in ANSI Standard C, but not in C++, where they are reserved keywords.

Table 2 C++ keywords

Keyword	Function	For more information, see
catch	Catch an exception.	How exception handling works on page 166
class	Declare a class.	Class declarations and definitions on page 88

Table 2 C++ keywords (con't)

Keyword	Function	For more information, see
const_cast	Cast away **const**.	const_cast and casting away const on page 179
delete	Destroy a class object.	The delete operator on page 21
dynamic_cast	Perform a dynamic cast (pointer-to-base class to pointer-to derived class).	dynamic_cast on page 179
friend	Mark a class or class member function as a friend of another class.	Friends on page 113
inline	Cause compiler to inline a function.	Inline functions on page 26
mutable	Override **const** setting of individual data members of a **const** class.	
namespace	Declare a namespace.	How to specify a namespace on page 158
new	Create a class object.	The new operator on page 20
operator	Overload an operator.	Operator functions on page 140
private	Make a class accessible only to its own and friend member functions.	Access on page 108
protected	Make a class accessible to its own member functions, member functions of derived classes, or friend member functions.	Access on page 108
public	Make a class accessible in any context.	Access on page 108
reinterpret_cast	Perform a cast of any type to any type that is big enough.	reinterpret_cast on page 180
static_cast	Perform a standard cast.	static_cast on page 178
template	Declare a template.	Declaring and using a template on page 146
this	Point to the current class object.	Non-static member functions on page 95
throw	Throw an exception.	How exception handling works on page 166
try	Specify code that can test for an error and throw an exception.	How exception handling works on page 166
typeid	Return a reference to an object of class **Type_info** or one of its derivatives.	The type object on page 183
using	Place a namespace or an element of a namespace within the current scope.	Using the names declared in a namespace on page 159
virtual	Declare a member function or base class to be virtual.	Virtual functions on page 100; Virtual base classes on page 103

static const objects

In C++, a **static const** object of an integral type can be used wherever a constant expression is required. This can replace the frequent usage of #define in C to achieve the same thing. For example:

```
static const LIM = 10;
int a[LIM*2];
```

In C++, unlike C, a **const** object declared in the global scope is assumed by default to be **static** unless you use **extern**.

Initialization

ANSI Standard C allows or requires some types of initialization that are not acceptable in C++. C++ introduces the use of constructors to initialize certain types of objects.

Initializing static entities

In C, initial values for **static** entities must be constants or constant expressions. In C++, this restriction is not present, but there is overhead when the expression is non-constant. The overhead consists of generated code that tests whether the **static** entity has been initialized. If it has not, initialization code is executed to initialize it.

Initializing structures

In ANSI Standard C, you can use an assignment operator followed by a brace-enclosed list of items to initialize a structure. Each item in the list initializes a data member of the structure, in the order the members are declared in the structure:

```
struct astruct {
   int l, m;
   int a[2];
   } thisstruct = { 1, 2, {3, 4} };
```

This brace-enclosed list cannot be used on C++ classes or structures that have constructors. For these you must use a C++ form of initialization (see Constructors on page 117). In some cases the compiler provides a constructor for a structure type if you do not, depending on the members of the structure.

Using constructors to initialize objects

In C++, a parenthesized expression list can be used to initialize objects of types with constructors:

```
class boy {
public:
   int height, weight;
   boy() { };
   boy(int i, int j) {
      height = i;
      weight = j;
      };
   };

boy tom(65, 163);
```

The listed expressions are taken as arguments to a suitable constructor for the object. If the object's class has no constructor, you can still use this form of initializer, but you can supply only one expression. The meaning is the same as if you had written = *expression*.

You cannot use the C++ form of initializer and supply *no* arguments to a constructor, because **classname X()** is instead interpreted as a function **X()** taking no arguments and returning a `classname`. Instead, specify just **classname X**; this causes the compiler to find the no-argument constructor and call it.

For C++ classes with constructors, there is a significant difference between C-style initialization of an object and the way the C++ forms initialize an object.

Here is an example of C-style initialization, which uses an assignment operator:

```
boy tom = boy(65,163);
```

In this example, the value **boy(65,163)** is a single expression that can be used to initialize **boy**-type object `tom`.

In C++, to put this value into `tom`, the class's copy constructor is called (a class always has a copy constructor; if you do not provide one, the compiler does). In this case the following operations occur:

1. A constructor is called to put **boy(65,163)** into a temporary.
2. The temporary is copy-constructed into `tom`.

Here is an example of C++-style initialization:

```
boy thom(65,163);
```

Here, the constructor taking two **int** arguments is called directly, and it constructs its result into `thom`. This form of initialization for classes with constructors is, in general, more efficient.

Where the compiler can detect that a copy constructor does nothing more than a bitwise copy, it can often optimize away the copy and construct directly into the variable being initialized; it is as if **boy tom = boy(65,163);** becomes **boy tom(65,163);**. Generally, however, this optimization is not possible.

> **NOTE** The C++ form of initialization, together with C++'s function-style form of casting, leads to C++ statements that can be interpreted as either declarations or expressions, and can make C++ programs hard to read.

Objects declared in jump statements

Jump statements pose a problem in C++ for objects with constructors declared in the scope of the jump statement. When control leaves the scope where the object is declared, the object is automatically destroyed, unless it is also declared in the scope of the target block. A jump statement cannot cause flow of control to enter a block and skip the construction of an object; for example:

```
struct s {
   s();
   ~s();
   };
f() {
   s x1;          // Constructs x1
      {
      s x2;       // Constructs x2
      goto L1;    // Destroys x2
      }
L1: goto L2;      // Invalid: skips construction of x3
      {
      s x3;       // Constructs x3
L2: goto L3;      // Destroys x3
      }
L3: return;       // Destroys x1
   }
```

Statements and declarations

C++ uses the same statement syntax as ANSI Standard C, with one difference. In ANSI Standard C, you must place all declarations in a given block before the first statement in the block. C++ allows you to place declarations anywhere they are needed, promoting the idea of declaring data items textually close to where they are first used; for example:

```
for (int i = 1; i <= 10; i++)
```

Resolving ambiguity between statements and declarations

The definition of C++ contains some cases in which it is difficult to distinguish between an expression statement and a declaration, and even some cases where text could be either. These cases occur when a fundamental or derived type name is followed by a left parenthesis. The problem is that a type name followed by a left parenthesis can be a function-style type cast. For example:

```
typedef struct { int age; } *recptr;
recptr(X)->age = 0;          // This is an expression.
recptr(Y) = 0;               // This is a declaration.
int(i);               // This is the declaration of i.
int(j) + 0;                  // This is an expression.
char(*string)(int);          // This is a declaration.
// f is a pointer-to-function taking a double.
double(*f)(double(x));       // This is a declaration.
// d is a pointer-to-double initialized with
// double(3).
double(*d)(double(3));       // This is an invalid
                             //  declaration.
```

In C++, if a text can be taken as declaration, it is so taken; otherwise, it is taken as an expression statement.

NOTE The statement **double(*d)(double(3))** is cited in the *<bAsIs>Annotated C++ Reference Manual* as a declaration. However, that is not correct, as confirmed by Bjarne Stroustrup.

You can resolve the ambiguity by examining whatever follows the right parenthesis. If the token following the right parenthesis is one of the following, the text is an expression statement:

- an operator (such as **->**)
- a function-call operator with a value as the next token
- an initializer

If the tokens following the left parenthesis are valid for a declarator, the text is a declaration.

Linkage specifications

A *linkage specification* is a mechanism that C and C++ provide to enable your program to communicate with code that is not written in the C or C++ language. Functions declared with linkage specifications are **extern**. You can declare functions as **static** within a linkage specification, but the linkage specification will be ignored for those functions. By default, if no linkage specification is given for an external function declaration, C/C++ linkage is assumed. You can use a linkage specification only at file-scope level.

Linkage from C/C++ to other languages and from other languages to C/C++ is implementation- and language-dependent.

Specifying linkage

The form for a linkage specification is one of the following:

```
extern "language_name" {[declaration]...}
```

```
extern "language_name" declaration
```

language_name is a string literal that corresponds to the name of the programming language in which the external code is written. MetaWare C/C++ recognizes the following language names:

```
C++
C
Pascal
FORTRAN
```

For example, to access C library function **strlen()**, declare it as follows:

```
extern "C" int strlen(const char *);
```

Functions

C++ allows you to do some things with functions, particularly class member functions, that you cannot do in ANSI Standard C.

- The const/volatile qualifier list
- Inline functions
- Reference parameters
- Default parameters
- Function-name overloading

The const/volatile qualifier list

In C++, an optional **const/volatile** qualifier list can follow the right parenthesis of a **virtual** (non-**static**) class member function's argument list. For information about the **const/volatile** qualifier list, which applies only to class-member functions, see Qualified member functions on page 96; see C++ classes on page 87 for more information about **virtual** and member functions.

Inline functions

The keyword **inline** is a hint to the MetaWare C++ compiler to substitute the body of a function declaration at each occurrence of a call to the function. It is only a hint, and the compiler can ignore it. Although MetaWare C++ can inline functions of arbitrary size and complexity, you can specify a limit to force the compiler not to inline functions that exceed a certain size.

See the *MetaWare C/C++ Programmer's Guide* for more information about compiler options for inlining.

The optimizing effect of inlining

Inlining is an optimization; inlining a function does not change the semantics of the function call in any way. When the compiler inlines a function, your program usually executes faster. This is because function-call overhead, such as saving and restoring registers, has been eliminated. Inlining can also

make a program smaller, if the functions being inlined are so small that the code generated to issue a call to the function exceeds the size of the function-body code.

Inlining versus macros

Inlining contrasts with the ANSI Standard C tradition of using a macro instead of a function call to increase program execution speed. If the body of the macro references a macro parameter more than once, undesirable side effects can occur when the argument is an expression containing side effects, such as ***p++**.

Multiple definitions of inline functions

You must make sure that every **inline** function has exactly the same definition in each compilation in which it appears, because if the **inline** function's definition is different in two different source files, program behavior becomes undefined.

For example, if you compile a source file using one definition of an **inline** function, where the definition of the **inline** function is in a header file, then modify the header file and compile another source file using the same function, the semantics of the **inline** function might have changed. Here is an example:

```
inline float sqr(float f) { return f*f; } // Square f.
...
float f1 = 3.7, g = sqr(f1);       // sqr is inlined.
struct s {
   float f;
   float sqr() { return f*f; }     // Square member f.
   // (Functions defined in a class definition
   //  are inline by default.)
   };
s x1 = {3.7}, g = x1.sqr();        // sqr is inlined.
```

Reference parameters

In C++ you can specify a *reference parameter*. This allows you to pass a parameter to a function, assign a value to it in the called function, and have the actual parameter's value changed. The change occurs immediately.

```
void func(double& radius) {
   radius += 3.1416;
   }
main(){
   double d=0.0;
   func(d);
   cout << d;
   }
```

In this example, *radius* is declared as a reference argument of **func()**. The call to **func()** causes `d` to have the value `3.1416`.

You cannot dereference a pointer to a function and assign a value through it. For a data item, if you dereference the pointer and assign a value through it, the value of the actual parameter is changed.

NOTE Although the reference mechanism is most widely used in parameter passing, it is generalized in C++ to include arbitrary reference variables; that is, you do not need to pass a parameter to make a reference to a variable.

Default parameters

In C++ you can declare a function with default parameters; for example:

```
int myfunct( int a, char b, double d = 5.0 );
```

The last parameter in the list for **myfunct()** is of type **double**; its default value is 5.0. This default value is used if a call to **myfunct()** omits the last actual parameter; for example:

```
int i;
i = myfunct(3, 'a'); // Passes 5.0 as third argument
```

You can still call **myfunct()** with three arguments to override the given default:

```
int i;
i = myfunct(3, 'a', 6.0); // Passes 6.0
                          //  as third argument
```

If a default parameter is given, all parameters following that default parameter must also have defaults:

```
void func1(int a,     int b = 2, int c);   // Error
void func2(int a,     int b = 2, int c=3); // OK
void func2(int a = 1, int b = 2, int c=3); // OK
```

Notice that both declarations of **func2()** are allowed. After the first declaration, **func2()** has two default parameters and could be called with a single value. After the second declaration, a third default is supplied, for the first parameter. That is, default parameters are cumulative.

A default parameter does not affect a function's definition. It affects only calls to the function. The same function could be declared in different ways in different source modules, with different default parameters.

Function-name overloading

In C++ you can declare multiple functions with the same name but different parameter lists; for example:

```
int power(int x, int p) {...}

double power(double d, int p) {...}
```

See Function overloading on page 135 for more information.

Chapter 3 — Extensions

In This Chapter

C99 Extensions

MetaWare C supports the following extensions from the ISO/ANSI C99 Standard.

Compound Literals

Compound literals let you construct aggregates by using a cast containing an initializer.

aggregate = `((`*cast*`) {`*list_of_constants*`});`

The braces amount to a constructor for a constant of the given type.

NOTE This syntax does not support the GNU feature of initializing a static structure with a compound literal.

Example 1 Array Constructed Using Compound Literals

```
char *array[] = ((char *[]) { "x", "y", "z" });
```

Example 2 struct Constructed Using Compound Literals

```
//Declaration of structs str and str1:
struct str {int a; char b[2];} str1;

// Constructing str1 with a compound literal:
     str1 = ((struct str) {x + y, 'a', 0});

// This is equivalent to writing the following:
     {
       struct str temp = {x + y, 'a', 0};
       str1 = temp;
     }
```

Hexadecimal Floating-Point Constants

Hexadecimal floating-point-constants are a direct representation of IEEE-754 floating-point values ($1.0 <= x < 2.0$ with the appropriate exponent and sign) and allow precise representation of values such as the maximal/minimal fixed-point values that are needed for many DSP algorithms.

The format is as follows:

0x[*alt_base*]*ipart*P[*plus_minus*]*exp* or

0x[*alt_base*]*ipart*.P[*plus_minus*]*exp* or

0x[*alt_base*].*fpart*P[*plus_minus*]*exp* or

0x[*alt_base*]*ipart*.*fpartP*[*plus_minus*]*exp*

where:

- *alt_base* is the MetaWare extension of *digitX* where *digit* is the alternate base to use for the number (*alt_base* applies only to *ipart* and *fpart*; *exp* is always a $base_{10}$ value)
- *ipart* is the integer portion of the constant
- *fpart* is the fractional portion of the constant
- *plus_minus* is either + (ignored) or - (negate the *exp*)
- *exp* is a $base_{10}$ value
- `0x` is required
- `P` is required (lowercase `p` is permitted as well)
- `.` is optional

The number represented is: *ipart*.*fpart* * 2^{exp}.

Example 3 Using Hexadecimal Floating-Point Constants

```
double t1 = 0x1.1p-1; // hexadecimal representation of 0.75
double t2 = 0.75;     // normal floating-point number (t1 == t2)
```

Designated Initializers (C99 Syntax)

Designated (named) initializers allow you to initialize data structures out of order or leave gaps in initialization.

Example 4 C99 Designated Initializers with a Struct

```
struct str{
   int i;
   int j;
   int k;
   }mystr = {.j = 1, .k = 2, .i = 0};
```

Example 5 Designated Initializers with an Array

```
int A[10] = {[8] = 3, [5] = 2};
```

Example 6 Designated Initializers with a Multidimensional Array

```
int array[10][20]={[8][8]=5, [5][5]=2, [15][2]=0};
```

Alternate MetaWare Syntax

Designated initializers for **struct** and **union** members may also use a syntax similar to that of the MetaWare feature Named Parameter Association. For example:

```
struct str{
   int i;
   int j;
   int k;
   } mystr = {j => 1, k => 2, i => 0};
```

As with named parameter association, use of this syntax is a MetaWare language extension, and is not portable to other compilers.

Related Topics

- Designated Initializers (GNU Syntax) on page 34
- Named Parameter Association on page 52

Case Ranges in Switch Statements (C99 and GNU Syntax)

MetaWare C/C++ supports the GNU and C99 syntax for specifying a range of values for **case** labels in **switch** statements:

```
case 'A' ... 'Z':
```

> **CAUTION** Be sure to place spaces around the `...` range indicator; otherwise, your range might be parsed wrong when you use it with integer values. For example, write `case 1 ... 5:` rather than `case 1...5:`

This feature is disabled if you specify compiler option **-Hansi** for ANSI C.

Related Topic

- Case Ranges in Switch Statements (MetaWare Syntax) on page 49

Function Names as Strings (C99 Identifier)

MetaWare C supports the identifier `__func__` to hold the name of the current function as it appears in the source.

The function name is placed in a **const char** array. The following example prints `Currently in func1` to `stdout`:

```
#include <stdio.h>
void func1(void){
   printf ("Currently in = %s\n", __func__);
   }
```

Related Topic

- Function Names as Strings (GNU Identifiers) on page 37

GNU Extensions

MetaWare C supports the following GNU extensions.

__attribute__ Syntax

MetaWare C/C++ supports an **__attribute__** syntax similar to that of GNU. The **__attribute__** syntax is allowed in both C and C++ and is largely identical between the two languages. The syntax is allowed for compilation in ANSI mode as well.

The syntax is as follows:

`__attribute__ (( `*Attr_list*` ))`

- *Attr_list* is a comma-separated list of attributes, which may include parameters, for example:

  ```
  __attribute__,
  __attribute__(parameter, parameter,...),
  ...
  ```

- The attributes may use or omit the leading and trailing underscores, e.g., **volatile** or **__volatile__**.

Unknown and unimplemented attributes are accepted and ignored with a warning.

Table 3 Attributes is a list of the currently implemented attributes and the entities to which they may be applied.

NOTE Some attributes might behave more like hints than absolute directives.

Table 3 Attributes

Attribute	Applies to	Meaning
alias	Functions Variables	Runtime alias
aligned	Functions Variables Types	Set minimum alignment (note that you must also specify **_Near** to place **aligned** variables in small data)
always_inline	Functions	Set **_FORCE_INLINE** calling convention
const	Functions Variables Types	Set **const** qualifier; make functions pure
common **nocommon**	Variables	Make common or not common; toggle `Multiple_var_defs` overrides
dllexport **dllimport**	Functions Variables	Like **__declspec(dllexport)** or **__declspec(dllimport)**
far **near**	Functions	Set **_Far** or **_Near** qualifier
interrupt	Functions	Set **_INTERRUPT** calling convention
longcall **shortcall**	Functions	Set **_Far** or **_Near** qualifier

Table 3 Attributes

long_call **short_call**	Functions	Set **_Far** or **_Near** qualifier
noinline	Functions	Turn off inlining for this function
noreturn	Functions	Set **_NEVER_RETURNS** calling convention
nothrow	Functions	Set **_NEVER_THROWS** calling convention
packed	Variables Types	Pack **struct** or unalign member
section("string")	Functions Variables	Place in section *string*
volatile	Variables Types	Set the **volatile** type qualifier
weak	Functions Variables	Mark as weak for external linkage

Examples

Example 7 Using the __attribute__ Syntax on a Variable

This example assigns variable `i` to user-defined data section `my_data`.

```
__attribute__ ((section("my_data"))) int i;
```

Example 8 Using the __attribute__ Syntax on a Function

This example declares a C++ function never returns and is always inlined.

```
__attribute__ ((noreturn, always_inline)) void func();
```

Example 9 Using the __attribute__ Syntax on a Type

This example ensures all variables of type **vint** are volatile.

```
typedef __attribute__ ((volatile)) int vint;
```

Assembly Instructions with C Expression Operands

MetaWare C/C++ supports the GNU **asm** syntax for inserting an inline assembly instruction with C operands. See the topic “Inline Assembly Code” in the *MetaWare C/C++ Programmer’s Guide.*

Designated Initializers (GNU Syntax)

Designated (named) initializers allow you to initialize data structures out of order or leave gaps in initialization.

This feature is disabled if you specify compiler option **-Hansi** for ANSI C.

Example 10 GNU Designated Initializers with a Struct

```
struct str{
   int i;
   int j;
   int k;
   }mystr = {j: 1, k: 2, i: 0};
```

Example 11 Designated Initializers with an Array

```
int A[10] = {[8] = 3, [5] = 2};
```

Example 12 Designated Initializers with a Multidimensional Array

```
int array[10][20]={[8][8]=5, [5][5]=2, [15][2]=0};
```

Alternate MetaWare Syntax

Designated initializers for **struct** and **union** members may also use a syntax similar to that of the MetaWare feature Named Parameter Association. For example:

```
struct str{
   int i;
   int j;
   int k;
   } mystr = {j => 1, k => 2, i => 0};
```

As with named parameter association, use of this syntax is a MetaWare language extension and is not portable to other compilers.

Related Topics

- Designated Initializers (C99 Syntax) on page 31
- Named Parameter Association on page 52

Conditional ?: with Omitted Operand

The middle operand in a tertiary conditional expression may be omitted. If the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression

```
exp1? : exp3
```

has the value of `exp1` if `exp1` is nonzero; otherwise, the value of `exp3`, the same as

```
exp1? exp1 : exp3
```

This feature is disabled if you specify compiler option **-Hansi** for ANSI C.

Underscored __signed__ and __unsigned__

The MetaWare compiler accepts **__signed__** and **__unsigned__** as synonyms for **signed** and **unsigned**.

__alignof__() Keyword

The **__alignof__()** keyword allows you to inquire about how an object is aligned, or the minimum alignment required by a type. Its syntax is **__alignof__(***type_or_lvalue***)**.

For example, if the target processor requires a **double** value to be aligned on an 8-byte boundary, then **__alignof__ (double)** is `8`.

If the operand is an lvalue rather than a type, the value of **__alignof__()** is the required alignment for the lvalue's type, taking into account any minimum alignment specified using the **__attribute__** syntax (see __attribute__ Syntax on page 33).

Asking for the alignment of an incomplete type generates an error.

Related Topic

- __attribute__ Syntax on page 33

Built-in Functions

See the alphabetical listing of compiler-intrinsic functions in the *MetaWare C/C++ Programmer's Guide* for information about GCC built-in functions supported.

Variants of typeof

MetaWare C supports the following forms of **typeof** to refer to the type of an expression:

- **typeof<***exp***>**
- **typeof(***exp***)**
- **__typeof(***exp***)**
- **__typeof__(***exp***)**

The **typeof** keyword is disabled if you specify compiler option **-Hansi** for ANSI C. Use **__typeof** or **__typeof__** for ANSI or ISO C.

Labels As Values

You can use the unary operator **&&** to take the address of a label defined in the current function (or a containing function). The value is a constant of type **void *** and can be used wherever a constant of that type is valid. For example:

```
void *ptr;
...
ptr = &&func;
```

To use these values, you must be able to jump to one. You can do so using something like a computed **goto** statement, **goto** `*exp;` For example:

```
goto *ptr;
```

Any expression of type **void *** is allowed.

The legitimate uses of this extension are the following:

- As a substitute for the Fortran assigned goto
- To pass an address to be printed in an error message

CAUTION Do not attempt to branch to a nonlocal label using **&&**. If you do so, the stack does not update properly, resulting in application failure.

This feature is disabled if you specify compiler option **-Hansi** for ANSI C.

Anonymous Functions

An anonymous function is the following construction:

`({`*some code*`})`

The compiler turns this construction into a nested function. The last expression in the code is the return value for the function, if it has a non-void type.

Example 13 Anonymous Function

```
//An example from the Linux kernel:

#define __get_user_nocheck(x,ptr,size)\
({ long _err=0,_val;\
```

```
__get_user_size(_val,(ptr),(size),_err);\
(x) = _val; })
```

Function Names as Strings (GNU Identifiers)

MetaWare C supports identifiers to hold the name of the current function as it appears in the source:

- `__FUNCTION__`
- `__PRETTY_FUNCTION__`

The function name is placed in a **const char** array. The following example prints `Currently in func1` to `stdout`:

```
#include <stdio.h>
void func1(void){
   printf ("Currently in = %s\n", __FUNCTION__);
   }
```

Related Topic

- Function Names as Strings (C99 Identifier) on page 32

MetaWare Extensions

MetaWare C/C++ includes the following extensions:

- Comments
- Underscores in numbers
- Specifying bases for numbers
- Keywords
- Type Qualifiers
- Preprocessor directives
- ANSI preprocessor numbers
- Data type long long int
- Short enumerators and long enumerators
- Kanji support
- Mixing statements and declarations
- Initializing automatic aggregates
- Case Ranges in Switch Statements (MetaWare Syntax)
- Labels in goto statements
- Nested functions and full-function variables
- Up-level referencing
- Full-function values
- Non-local labels
- Named Parameter Association

- Near and far objects and pointers
- Distance attributes
- Compiler-Intrinsic Functions
- Inline assembly code
- Direct register assignment

Comments

MetaWare C/C++ accepts the C++-style // comment in both C and C++ code.

Underscores in numbers

Numbers, both floating-point and integer constants, can be **`written`** with underscores (_) between the digits. Generally, the underscore takes the place of the English comma in numbers. For example:

```
1_000_000        // One million
```

Specifying bases for numbers

In MetaWare C/C++, you can specify any base (radix) from 2 to 16 for an integer constant, whereas ANSI Standard C supports only octal, decimal, and hexadecimal integer constants.

MetaWare C/C++ recognizes a second or later **x** in a hexadecimal number beginning with **0x** as a switch to a new base for interpreting the value after the **x**. The currently evaluated number becomes the value for the new base:

```
0xfffe        // Regular hexadecimal number
0x2x1001      // Binary number equal to decimal 9
0x2x1001x20   // Base 9 number with value equal to
              //  decimal 18 (20 in base 9)
0x10xfffe     // Same as 0xfffe since 0x10 is 16
```

Keywords

The following keywords are MetaWare C/C++ extensions to C and C++:

_Alias	**_Interrupt**	**pragma**
_Asm	**_Near**	**_Reversed_Endian**
_CC	**_Noalias**	**_Unaligned** (ARC, ARM, and MIPS targets)
__declspec	**_Overlay()** (ARC targets only)	**_Uncached** (ARC targets only)
_Far	**_Override**	**_Unpacked**
_Inline	**_Packed**	

Of the keywords listed, the following are type qualifiers:

_Alias	**_Interrupt**	**_Override**
_Asm	**_Near**	**_Reversed_Endian**
_CC	**_Noalias**	**_Unaligned** (ARC, ARM, and MIPS targets)

_Far **_Overlay()** (ARC targets only) **_Uncached** (ARC targets only)

_Inline

See Type Qualifiers on page 40 for more information.

Keyword __declspec

MetaWare C/C++ supports the keyword **__declspec**, which is used to assign storage-class attributes to identifiers. For example, to assign the storage-class attribute **thread** to integer variable `index`, you declare the variable as follows:

```
__declspec(thread) int index;
```

MetaWare C/C++ recognizes the following storage-class attributes:

dllexport	Specifies identifiers to be exported from a DLL
dllimport	Specifies identifiers to be imported (made available) to a DLL
thread	Specifies thread-local storage for an identifier

Keywords _Packed and _Unpacked

Whether MetaWare C/C++ aligns structure members by default is platform dependent. If your target platform supports packed structures, you can use the MetaWare C/C++ reserved words **_Packed** and **_Unpacked** to specify alignment on a per-structure basis. You can also use pragma `Align_members` to force alignment for any structure definition occurring within any program region in which this pragma is active. For example:

```
_Packed  struct {char c; int i;} x;
                          // Packed, as by default
_Unpacked  struct {char c; int i;} y;
                          // Unpacked, not default
struct {char c; int i;} z; // Unpacked, as by default
pragma Align_members(1);
struct {char c; int i;} w; // Packed, as by default
```

If your target platform does not support packed structures, keywords **_Packed** and **_Unpacked** are not implemented. See the *MetaWare C/C++ Programmer's Guide* for information.

See the *MetaWare C/C++ Programmer's Guide* for information about pragma `Align_members`.

Keyword pragma

In MetaWare C/C++, **pragma** is a reserved word. You cannot use the word **pragma** as an identifier. Pragmas are used to direct the compiler when more than one possibility is provided for an implementation-defined aspect of compilation. Subsequent to MetaWare C/C++'s choice of **pragma**, the ANSI C committee chose **#pragma** for the same purpose.

MetaWare C/C++ supports **#pragma**, and continues to support **pragma** for compatibility. We suggest using **#pragma** for portability. However, ANSI's **#pragma** cannot appear within macros, whereas MetaWare C/C++'s **pragma** can, so there is a definite advantage to using **pragma**.

Type Qualifiers

Type qualifiers are used to modify or place limits on data types. The following type qualifiers are extensions to MetaWare C/C++:

Table 4 Type qualifiers that are extensions to MetaWare C/C++

Type qualifier	Description
_Alias	Variable can be accessed by indirect means.
_Asm	Function is an assembly language macro.
_CC	Specify calling convention.
_Far	Specify a far object or pointer.
_Inline	Inline specified function.
_Interrupt	Function is an interrupt function.
_Near	Specify a near object or pointer.
_Noalias	Variable cannot be accessed by indirect means.
_Overlay()	For ARC targets only; see the *Automated Overlay Manager User's Guide*.
_Override	The specified method overrides the base-class method.
_Reversed_Endian	Variables of opposite endianness are byte-reversed at the time of access.
_Unaligned	Assume pointer misaligned (ARC, ARM, MIPS targets)
_Uncached	For ARC targets only: Excludes variable from data cache in load and store operations.

NOTE Not all of these type qualifiers are implemented for all target platforms of MetaWare C/C++.

Casting type qualifiers

Due to the characteristics of C and how type qualifiers (sometimes referred to as type attributes) work, you cannot use type qualifiers in casts according to the normal C mechanism for casting. For example, this cast would not work as expected, because the interpretation for the cast takes place after the access is completed:

```
x = (volatile int) y ;
```

To cast a type qualifier, you must cast the address of the desired variable, and then dereference to get the desired effect. For example:

```
x = * (volatile int *) &y ;
```

This requirement applies to all type qualifiers that affect external accesses.

Overlay Type Qualifier

For ARC targets, MetaWare C (but not C++) provides the **_Overlay()** type qualifier for use with the ARC Automated Overlay Manager. For information on how to use **_Overlay()**, see the *Automated Overlay Manager User's Guide*.

Access-related type qualifiers

The type qualifiers covered in this section involve access to data.

Type qualifier _Alias

When you declare a variable with the **_Alias** type qualifier, you are telling the optimizer that the variable can be accessed by indirect means; for example, through a pointer dereference.

The **_Alias** type qualifier is used to override the effect of toggle `Noalias`. Ordinarily, any global variable or pointer dereference is assumed to be aliasable.

_Alias and **_Noalias** are mutually exclusive. **_Alias** is the default.

Type qualifier _Noalias

When you declare a variable with the **_Noalias** type qualifier, you are telling the optimizer to assume that the variable cannot be accessed or modified through any indirect means, such as through a pointer dereference.

When you apply **_Noalias** to a pointer-based variable, the optimizer assumes that no other pointer references the same memory location at the same time, and that the pointer does not contain the address of any variable that is accessed in the same function.

_Alias and **_Noalias** are mutually exclusive. **_Alias** is the default.

> **CAUTION** The compiler makes no attempt to confirm that these assumptions have not been violated. If the **_Noalias** qualifier is used incorrectly, the optimizer's assumption is wrong and the compiler could generate incorrect code.

Cache-Related Type Qualifier (ARC Targets)

The **_Uncached** type qualifier allows you to exclude a variable from data cache in load and store operations, without the additional semantic implications of **volatile**. The **.di** instruction suffix is used for all loads and stores of variables declared **_Uncached**.

You can use compiler toggles to place **_Uncached** data in a separate section and align the section. The toggle defaults might vary depending on the target ARC processor; see the *MetaWare C/C++ Programmer's Guide for ARC* for details.

You can apply **_Uncached** to variables that are not bit fields just as you normally would apply **const** or **volatile**, as shown in Example 14 Legal Uses of _Uncached.

Example 14 Legal Uses of _Uncached

```
_Uncached int * p1;
_Uncached int * _Uncached p2;
int * _Uncached p3;
```

However, you cannot apply **_Uncached** to individual members of a **struct**. Usage as shown in Example 15 Illegal Uses of _Uncached causes the compiler to issue an error message.

Example 15 Illegal Uses of _Uncached

```
struct str {
   int x;
   _Uncached int y; // disallowed
   int z;
   _Uncached int a:1; // disallowed
} s1;
```

The compiler issues warnings if you mix variables designated **_Uncached** with variables and function arguments not designated **_Uncached**. The compiler issues a warning if you cast an **_Uncached** variable to a variable that is not **_Uncached**. The compiler also issues a warning if you pass an **_Uncached** variable to an unprototyped function or as part of a **va_args** list to a function such as **printf()**.

Pointer Type Qualifier (ARC, ARM, and MIPS Targets)

The **_Unaligned** type qualifier informs the compiler that a particular pointer should be assumed to be misaligned.

Declaring the **int** pointer `*p` to be **_Unaligned** declares `p` as a pointer that the compiler should always assume to be unaligned, so that it generates piecemeal aligned accesses whenever dereferencing this pointer. The definition of a function **func()** can then be written as follows:

```
void func(_Unaligned int *p)
```

All dereferences of `p` within **func()** are now performed safely regardless of the alignment of `p`.

Endianness-related type qualifier

Type qualifier **_Reversed_Endian** allows you to reverse the endian orientation of variables.

With type qualifier **_Reversed_Endian**, you can declare variables that have an the opposite endianness of the machine on which the program runs. Loading and storing these variables requires byte-reversed memory accesses.

Here are some example declarations of reversed-endian variables:

- A reversed-endian variable:

  ```
  _Reversed_endian int a;
  ```

- A reversed-endian pointer to a normal variable:

  ```
  int * _Reversed_endian a;
  ```

- A normal pointer to a reversed-endian variable:

  ```
  _Reversed_endian int * a;
  ```

- A reversed-endian pointer to a reversed-endian variable:

  ```
  _Reversed_endian int * _Reversed_endian a;
  ```

> **NOTE** The MetaWare C/C++ compiler does not enforce type checking of reversed-endian variables. You must ensure that reversed-endian variables and pointers are not assigned to normal variables and pointers, and vice versa.

Function type qualifiers

The type qualifiers covered in this section enable you to customize the compiler's treatment of certain functions.

Type qualifiers _Near and _Far

MetaWare C/C++ introduces the concepts of "near" and "far" through the type qualifiers **_Near** and **_Far**. The placement of **_Near** and **_Far** follows the same rules as those for **const** and **volatile**; for example:

```
_Far int fi;          // A far-int
_Near int frni();  // A function returning a near-int
_Far int *_Far fpfi = &fi; // A far-pointer-to-
                           //  far-int
```

See Near and far objects and pointers on page 53 for more information about accessing "near" and "far" data.

Type qualifier _Asm

With the **_Asm** type qualifier, MetaWare C/C++ provides an enhanced **asm** capability through which you can insert several assembly instructions into the compiler's output. The **_Asm** type qualifier declares a set of assembly instructions to be a C function.

For more information, see the *MetaWare C/C++ Programmer's Guide*.

Type qualifier _CC

The type qualifier **_CC** specifies the calling convention of a declared function or functionality type. Its syntax is as follows:

```
_CC(Attributes)
```

Attributes is a bit pattern in which each bit specifies a particular calling-convention attribute. It must be a calling-convention expression of the same kind permitted in pragma `Calling_convention`. If you specify more than one calling convention, the final convention is the bit-wise OR of all the constituents. For example, the following expression specifies the Microsoft Pascal calling convention: the callee pops the stack, non-volatile registers are saved if the routine uses them, and parameters are pushed from left to right (right to left requires the **_REVERSE_PARMS** attribute):

```
_CC(_CALLEE_POPS_STACK | _SAVE_REGS)
```

The return type of a function, if it has **_CC** attributes, reflects those attributes to the function being defined. For example:

```
#define MSPASCAL _CC(_CALLEE_POPS_STACK | _SAVE_REGS)
int _Far MSPASCAL f() {...}
```

`f` is a function returning a **_Far int** with calling convention **_CALLEE_POPS_STACK | _SAVE_REGS**.

NOTE The attributes of the return type reflect back onto the function being defined, so `f` is really a **_Far** function having calling convention **_CALLEE_POPS_STACK | _SAVE_REGS** and returning an **int**.

Multiple **_CC()** specifications can appear in a single declaration; the effect of **_CC(e1) _CC(e2)** is the same as that of **_CC(e1 | e2)**.

There is one restriction on this calling-convention specification: the **_CC** syntax applies only if there is a function declarator:

```
function_name(argument_list)
```

The following does not work because **x** is not a function declarator:

```
#define MSPASCAL _CC(_CALLEE_POPS_STACK | _SAVE_REGS)
...
typedef MSPASCAL x;
x f();
```

The **MSPASCAL** is lost.

CAUTION The calling convention present on a declaration of a function overrides the convention present on a subsequent definition, without any warning from the compiler. Thus, in the following code, `f` has the **MSPASCAL** calling convention:

```
void MSPASCAL f();
void f() { ... }
```

However, in this code, `f` does not have the **MSPASCAL** calling convention:

```
void f();
void  MSPASCAL f() { ... }
```

A calling convention specified by **_CC** completely overrides that given by a calling-convention pragma.

The **_CC** specification is permitted where **const** and **volatile** are permitted.

See the *MetaWare C/C++ Programmer's Guide* for information about pragma `Calling_convention`, communication protocols between functions, and platform-specific attributes of type qualifier **_CC**.

Type qualifier _Inline

Type qualifier **_Inline** directs the compiler to inline a particular function (you must also specify option **-Hi** to invoke the stand-alone inliner). For example, this code causes the compiler to inline function **func()** when you compile with option **-Hi** specified:

```
_Inline void func() { ... }
```

Type qualifier **_Inline** applied to a function overrides any **-Hi** option that might otherwise exclude that function from inlining.

NOTE You can also control the inlining of individual functions with pragmas `On_inline` and `Off_inline`. For more information, see the *MetaWare C/C++ Programmer's Guide*.

Type qualifier _Interrupt

The **_Interrupt** qualifier designates a function as an interrupt function.

The compiler inserts a special tag word at the head of the function to identify what context needs to be saved.

For more information, see the *MetaWare C/C++ Programmer's Guide*.

Type Qualifier _Override

Type qualifier **_Override** specifies that a method provides a new implementation of a method inherited from a base class. This is currently the default behavior for C++; See the *MetaWare C/C++ Programmer's Guide* for information about toggle

Macros for ARC Targets

MetaWare provides macros for ARC targets that can be used like keywords to apply certain calling-convention attributes to functions.

Macro _Interrupt1 (ARC Targets)

Macro **_Interrupt1** specifies that a function is a level-one interrupt handler that returns using register `ilink1`. It expands to **_CC(_INTERRUPT1|_DEFAULT_CALLING_CONVENTION)**.

The following is a usage example:

```
int _Interrupt1 func(void) {
    ...
    }
```

Macro _Interrupt2 (ARC Targets)

Macro **_Interrupt2** specifies that a function is a level-two interrupt handler that returns using register `ilink2`. It expands to **_CC(_INTERRUPT2|_DEFAULT_CALLING_CONVENTION)**.

The following is a usage example:

```
int _Interrupt2 func(void) {
    ...
    }
```

Macro _Preserve_flags (ARC Targets)

Macro **_Preserve_flags** indicates that a function preserves condition flags. It expands to **_CC(_PRESERVE_FLAGS|_DEFAULT_CALLING_CONVENTION)**.

The following is a usage example:

```
int _Preserve_flags func(void) {
    ...
    }
```

Macro _Save_all_regs (ARC Targets)

Macro **_Save_all_regs** causes the compiler to save all registers used by a function, both volatile and nonvolatile. It expands to **_CC(_SAVE_ALL_REGS|_DEFAULT_CALLING_CONVENTION)**.

The following is a usage example:

```
int _Save_all_regs func(void) {
    ...
    }
```

Preprocessor directives

The following preprocessor directives are MetaWare C/C++ extensions.

#assert

#c_include

#define with `vararg` macro parameters

#include_next

#print

#unassert

#warning

Preprocessor directives #assert and #unassert

Preprocessor directive **#assert** allows you to associate a value with a name that you can subsequently reference in a **#if** statement, without affecting the namespace of macros. You use the **#assert** statement to define a *predicate* (a preprocessor construct) by associating with it a token sequence:

```
#assert predicate(token_sequence)
```

For example:

```
#assert machine(386i)
#assert os(unix)
```

You test for the assertion as follows:

```
#if #predicate(token_sequence)
```

For example:

```
...
#if #machine(386i) && #os(unix)
...
#endif
```

To remove all assertions against a predicate, use directive **#unassert** as follows:

```
#unassert predicate
```

To remove a single assertion, you must supply the *token_sequence*:

```
#unassert predicate(token_sequence)
```

#assert and **#unassert** are supported in all globally optimizing MetaWare C compilers.

Preprocessor directive #c_include

You use the directive **#c_include** for conditional file inclusion. **#c_include** works like **#include**, except that if the file to be included has already been included once in the source file, it is not included again. **#c_include** is therefore a *conditional* file inclusion facility that helps you avoid duplicate declarations. Using **#c_include** also avoids file I/O and compilation overhead because the included source file is processed only once.

Preprocessor directive #define with vararg macro parameters

You use the directive **#define** with `vararg` macro parameters in the same way that you use `printf` arguments, except that the `vararg` macro parameters are used exclusively in macro definitions.

For example:

```
#define  print_it(a,b,c...) printf(a,b,c)
....
print_it("%d %d %d %d %d", 1, 2, 3, 4 5);
```

expands to

```
printf("%d %d %d %d %d", 1, 2, 3, 4, 5);
```

The third parameter (*c*) of the macro `print_it()` uses the suffix `'...'`

This is an indicator that *c* is a `vararg` parameter and will expand to as many tokens as are left in the macro call, up to the closing right parenthesis, with no regard for commas. The `'...'` must act as a suffix to the parameter name. No space is allowed between the `'...'` and the name that it succeeds.

TIP Although the `vararg` parameter must be the *last* parameter in the list, you can precede it with any number of normal parameters.

NOTE This extension is disabled when option **-Hansi** is specified.

Preprocessor directive #include_next

Preprocessor directive **#include_next** provides an extension to ANSI Standard C that implements inheritance in header files by allowing you to include multiple header files of the same name on the search path.

CAUTION Because **#include_next** is an extension to ANSI Standard C, code that uses it might not be portable to other compilers.

Using **#include_next** allows you to modify the contents of a header file (include file) without editing it directly, by creating a local version. This is useful when you want to keep some aspects of the original include file and modify others.

The preprocessor behavior for **#include** is to stop searching after finding the specified file on the search path. Simply including the old include file within itself would cause infinite recursion. Using **#include_next** obviates this problem by causing the preprocessor to begin searching the search path after the location where it first found the include file. The second include file can then modify statements made in the first, implementing inheritance.

Example

```
#include <header.h>      /* Includes header.h from
                            default search path.      */
#include_next <header.h> /* Searches the directories
                            on the search path after
                            the location of header.h
                            to find the header.h that
                            you modified.             */
```

Preprocessor directive #print

The **#print** directive works the same way as the **#error** directive, but is not treated as a compile-time error that terminates compilation. **#print** provides a convenient way of printing informational messages.

Preprocessor directive #warn

Preprocessor directive **#warn** *string* (synonym: **#warning** *string*) causes the compiler to emit a warning. Argument *string* (without quotes) is the warning text emitted.

Example

```
#warn Including header.h from c:\my_old_stuff!
```

Included in the queens demo, this example produces the following:

```
w "../demos/queens.c",L28/C1(#705): Output from #warn: Including header.h from
c:\my_old_stuff!
```

ANSI preprocessor numbers

When ANSI Standard C *preprocessor numbers* were implemented in MetaWare C/C++, the lexical syntax for a number changed to the following:

```
'.'? Digit (Letter|Digit|'.'|('e'|'E')('+'|'-'))*
```

This means, for example, that `3` and `3.3.f.e+e-` are both valid preprocessor numbers.

Preprocessor numbers allow you to construct numbers out of their components with token pasting; for example, pasting `3e` to `10` results in the floating-point number `3e10`. After all preprocessing is done the compiler diagnoses any preprocessor numbers that are not valid numbers; so the `3.3.f.e+e-` example given above is diagnosed if it has not already been removed from the token stream during preprocessing, either by simple deletion or by converting it to a string.

Preprocessor numbers invalidate the MetaWare C/C++ extension **..** that has been present in MetaWare C/C++ since its inception. For example, `case 3..5:` is now a single case with the value `3..5`, which is a valid preprocessor number but an invalid number. Therefore, to protect the original `..` extension, MetaWare C/C++ does not allow **..** in a preprocessor number except when you compile in ANSI mode.

This extension is disabled when you compile in ANSI mode.

Data type long long int

Data type **long long int** is a *signed* integer, the same size as a **long int** or larger; typically, it is eight bytes.

long long int types are not supported on all systems. Where **long long int** types are supported, the range is typically -2^{63} to $2^{63}-1$ -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807); the range of an **unsigned long long int** is 0 to $2^{64}-1$.

On platforms where **long long int**s are not supported, the compiler issues a warning if you declare one, and treats the type as a **long int**.

Short enumerators and long enumerators

In MetaWare C/C++, you can use the adjectives **short** and **long** in declarations of **enum** types. **short** indicates that the enumeration type should use as little space as possible; **long** indicates that the enumeration type should have size equal to **sizeof(int)**. For example:

```
short enum se {red,white,blue};  // sizeof == 1
long  enum le {alpha,beta,gamma};// sizeof ==
                                 //  sizeof(int)
```

short enum is the default unless overridden by toggle `Long_enums` (see the *MetaWare C/C++ Programmer's Guide* for information about compiler toggles).

Kanji support

The MetaWare C/C++ compiler supports double-byte Kanji characters in C strings.

Kanji support consists of an interpretation of a C string that differs from the usual C interpretation. When a C string is read from left to right, any occurrence of a character in the range `0x80..0x9f` or `0xe0..0xfc` is considered to be the first character of a two-byte Kanji character. The character that follows is taken without interpretation; for example, if the second character is \, it is not interpreted as the standard C escape sequence.

This method of Kanji support is a *de facto* Japanese standard. The ANSI C Standard dictates that character strings of the form `L"xxxx"` denote multi-byte character strings.

You enable Kanji support with compiler option **-Hkanji**; see the *MetaWare C/C++ Programmer's Guide* for information about this option.

NOTE The MetaWare C Library, especially its string functions, does not support Kanji. You must supply your own libraries.

Mixing statements and declarations

MetaWare C/C++ allows you to intermix statements and declarations even when you are not using C++ features. That is, you can place declarations wherever they are needed; you do not need to put all declarations in a given block before the first statement in the block.

Initializing automatic aggregates

MetaWare C/C++ permits automatic aggregates to be initialized with arbitrary non-constant expressions, where ANSI Standard C restricts initialization lists to constant expressions. For example, the following is valid in MetaWare C/C++, but not permitted in ANSI Standard C:

```
void fnc (int i) {
   int x[5] = { 0, i, 2*i, 4*i, 8*i};
   ...
   }
```

Case Ranges in Switch Statements (MetaWare Syntax)

MetaWare C/C++ permits you to specify a range of values for **case** labels in **switch** statements. You specify a case range as follows:

```
case constant_expr..constant_expr:
```

The first *constant_expr* specifies the lower bound of the **case** label; the second *constant_expr* specifies the upper bound. When you specify a range for a case label, the case label matches *expression* whenever the value of *expression* is greater than or equal to the lower bound, and less than or equal to the upper bound of the range.

For example:

```
switch (Ch) {
   case 'A'..'Z':   Scan_id();
                    break;
   case '0'..'9':   Scan_number();
                    break;
   default:         Scan_delimiter();
                    break;
   }
```

Related Topic

- Case Ranges in Switch Statements (C99 and GNU Syntax) on page 32

Labels in goto statements

As an extension to ANSI Standard C and C++, MetaWare C/C++ allows you to specify a label in a **goto** statement in a nested function if the label definition is in an outer function. If execution is currently at a **goto** statement in a nested function, you can specify a label in an outer, surrounding function as the target of the **goto**.

See Objects declared in jump statements on page 24 for information about a problem posed by jump statements in C++. See Nested functions and full-function variables on page 49 for more about function nesting.

Nested functions and full-function variables

In MetaWare C/C++, functions can be defined within functions. Such functions are called *nested*.

Up-level referencing

In the body of a nested function `N`, any name in a containing scope can be accessed. That is, the body of `N` can use names local to any function that contains `N`. This is called *up-level referencing* and any such names are said to be *up-level referenced* from `N`. The single restriction is that **register** variables cannot be up-level referenced.

You can achieve up-level referencing by making available to `N`, at each call to it, a way to reference the collection of local variables belonging to each of its enclosing functions. These collections together are called `N`'s *environment*. The function `P` immediately enclosing `N` is called `N`'s *parent*; the next enclosing function `G` its *grandparent*; and so on. The collection of local variables of each function is called its *frame*.

`N`'s environment can be implemented by passing to `N`, at each call to it, a "hidden" parameter that is a reference to `P`'s frame. If this is done for all functions, `P` will have in its frame a similar hidden parameter that links it to `G`, and so on out to the global level where functions do not need such a link. This is called the *static link* method of implementing up-level referencing, the method of choice for best efficiency and optimization.

A major difference between nested functions and non-nested functions is that the address of a nested function `N` does not entirely capture `N`'s "value"; the environment is also required. In contrast, the address of a non-nested or global function `G` entirely captures its value, because `G` has no parent and thus needs no environment. The C concept of "pointer-to-function" is therefore sufficient to capture the value of `G` but not that of `N`.

For this reason, MetaWare C/C++ disallows taking the address of a nested function, and, where C assumes **&** before any expression of type function, MetaWare C/C++ does not assume the **&** if the expression is of a nested-function type.

Full-function values

We refer to the combination of a function address and its environment as a *full-function value*, as opposed to just a "function address".

All the capabilities associated with global functions, such as passing them as parameters and storing their values into variables, are available for nested functions, although new syntax is required.

Declaring full-function variables

A variable capable of holding a full-function value, and therefore the value of a nested function, is declared as a function declaration would be, except that "`!`" follows the parenthesized formal parameter list. For example:

```
int (*ffv)()!;
```

In contrast, an ANSI Standard C variable capable of holding only a function address is declared using the pointer syntax:

```
int (*fa)();
```

Any nested function can be assigned to `ffv`. A global function `G` can be assigned to `ffv` by dereferencing it, because G is transformed to `&G` by the compiler and must be dereferenced to obtain the full-function value of `G`, not just its address: `ffv = *G;`. The environment stored in `ffv` in such an assignment is meaningless, since `G` needs no environment. Upon calling the value in `ffv`, the environment is passed to `G`, but `G` (indeed every global function) safely ignores it.

Passing nested functions as parameters

Nested functions can be passed as parameters: the full value is passed. The full-function value of a global function can be passed by dereferencing the global function; the passed environment is meaningless.

An argument can be declared to be a full-function value by using the new syntax:

```
int f(ffv) int ffv()!; { ... ffv(); ...}
```

Function constants

Only the names of function constants can be dereferenced to produce full-function values. The dereference of a pointer to a function is immediately converted back to an address by ANSI Standard C rules for function expression conversion; for example:

```
extern int sub();
main () {
   int (*fa)();
   int Nested() {...}
   main(*main);  // Passes the full value of main
   (*sub);       // Passes the full value of sub
   main(*fa);    // Passes fa, since *fa => &*fa = fa
   main(Nested); // Passes the full value of Nested
   }
```

This extension is compatible with ANSI Standard C, because the dereference of an expression of type pointer-to-function is permitted only in the context of an expression denoting a function to be called — for example, `(*fa)()` — but `(*fa)(*fa)` is not allowed in ANSI Standard C.

An example using full-function values

As an example of the use of full-function values, we present a call to a sort function that takes as parameters two functions:

```
extern void Quick_sort(
   int Lo, int Hi, int Compare(int a, int b)!,
   void Swap(int a,int b)!
   );
static Sort_private_table() {
   Entry Entries[100];
   int Compare(int a,int b) {
      return Entries[a] < Entries[b];
      }
   void Swap(int a,int b) {
      Entry Temp = Entries[a];
      Entries[a] = Entries[b];
      Entries[b] = Temp;
      }
   ...
   Quick_sort(1,100,Compare,Swap);
   }
```

Here **Compare** and **Swap** must be local to **Sort_private_table** because the table **Entries** is local to that function. In ANSI Standard C, **Entries**, **Compare**, and **Swap** would have to be moved outside of **Sort_private_table**. This works fine in this simple case, but if **Sort_private_table** were recursive, you would have to explicitly manage a stack of **Entries** arrays to get the desired effect.

Casting a full-function value

You can cast any full-function value of one type to any full-function value of another type in MetaWare C/C++, in concert with the ability to cast function addresses in ANSI Standard C. The **sizeof** a full-function type can be taken and is always greater than the **sizeof** a function address, because the former includes the environment.

Declaring full-function types

The additional syntax required to permit the declaration of full-function types is identical to the rules for Standard function syntax, except that "**!**" is allowed.

```
Declarator'
-> Extended_function_specification_declarator:
   Declarator' Parameters '!';
Abstract_declarator'
-> Abstract_declarator'? Abstract_parameters '!';
Declarator2'
-> Declarator2' Parameters '!';
```

Non-local labels

In MetaWare C/C++ a function label is visible to the function in which it is declared and to all of the descendants of that function. A label declared in a descendant overrides the declaration of a label in an ancestor.

Jumping to an ancestor's label is a valuable technique for easily unwinding a computation deep in the throes of recursion. It is a disciplined form of C's `setjmp()/longjmp()`:

For example:

```
void Wind(void Callme()!) {
   void P(int Cnt) {
      if (!Cnt) Callme();
      P(Cnt-1);
      }
   P(100);
   }
void main () {
   void Called() {
      printf("%s%s","Called from deep within"
            "recursion -- exiting.\n");
      goto EXIT;
   }
   Wind(Called);
   printf("I should not get here.\n"); return;
EXIT:
   printf("%s%s","Deep recursion unwound by"
         " a single goto.\n");
   }
```

Here **main()** invokes a procedure **Wind()** that causes local function **P()** to make 100 invocations of itself before calling function **Called()**. **Called()** disposes of all the invocations of **P()** in one fell swoop by jumping to the `EXIT` label.

The MetaWare C/C++ compiler emits the proper code to clean up the stack.

- In locally optimizing MetaWare C/C++ compilers, the operation is extremely cheap — just a few instructions — but variables in the function containing the `EXIT` label cannot be allocated to registers.
- In globally optimizing MetaWare C/C++ compilers, each unwound stack frame must be visited individually to do the unwinding so that register variables can be used in the **main()** function.

Although this example might seem frivolous, there are serious uses of non-local **gotos** in the MetaWare C/C++ compiler itself.

Named Parameter Association

Functions declared with parameter names can be called with the named parameter association syntax of Ada. Such calls refer to the parameter names rather than their positions in the argument list, so that the order of actual parameters is irrelevant.

The syntax is like that of a normal function call, except that each actual parameter expression is preceded by the corresponding formal parameter name followed by the operator =>; for example:

```
typedef enum {Red, Green, Blue} Color;
void P(int A, float B, Color C, Color D) { ... }
...
P(C => Red, D => Blue, B => x*10.0, A => y);
```

You can also start the function call using positional parameter notation and then switch to named association as you add parameters from left to right. However, you cannot switch back to positional notation for later parameters; nor is any other variation allowed. Here is an example of valid parameter notation:

```
void Plot(Xlo, Xhi, Ylo, Yhi, Xinc, Yinc)
   float Xlo, Xhi, Ylo, Yhi, Xinc, Yinc; {
   ...
```

```
   }
...
Plot(Alo,Ahi,Blo*2.0,Bhi*2.0,Yinc=>y,Xinc=>f(x+z));
```

That is, there can be an initial portion of the argument list in the positional form (a "positional prefix"), followed by named-parameter association form ("named parameter association suffix").

Thus, this is allowed:

```
f(1, 2, 3, aa => x+y, bb => q);
```

but this is not:

```
f(1, 2, 3, aa => x+y, bb => q, 7, 8, 9);
```

Constraints on using named parameters

Any function called with named parameter association must specify the names of all its arguments; for example:

```
int f(float g); // valid call
int f(float);   // not a valid call
int f();        // also not a valid call
```

The identifiers in a named parameter association form of call must collectively be the names of those parameters to the called function whose values are not supplied positionally. Values for all parameters must be supplied through either the positional prefix or through the named parameter association suffix.

Given these constraints, you can transform the function call into a purely positional form. For example, the call to **P()** in the first example in this section can be transformed from:

```
P(C => Red, D => Blue, B => x*10,0, A => y);
```

to:

```
P(y, x*10.0, Red, Blue);
```

Then the regular constraints and semantics of ANSI Standard C must be obeyed for the transformed call.

Near and far objects and pointers

The concepts of "near" and "far" are common extensions on architectures whose instruction sets can use addresses of at least two different sizes. MetaWare C/C++ introduces the concepts of "near" and "far" through the keywords **_Near** and **_Far**.

Many objects are addressed relative to a default segment number. Far references specify a non-default segment number in an address reference, and near references specify use of a default segment rather than an otherwise-implied non-default segment number.

Distance attributes

Any type can possess a near or far attribute. We call these attributes *distance attributes* and say that a type is *distance attributed*.

NOTE If you prefer using the words **near** and **far** rather then reserved keywords **_Near** and **_Far**, use #define directives to redefine these keywords.

An object with distance-attribute `X` and type `T` can be referred to as having type `X T`; for example, **_Far int**. A pointer to an object with distance-attribute `X` and type `T` can be referred to as having type

pointer-to-X-T; for example, "pointer-to-**_Far-int**". As with **const** and **volatile**, we have seen the imprecise term "far pointer" used to denote, most often, a pointer-to-**_Far**-*sometype*. Less frequently it denotes a **_Far**-pointer-to-*sometype*.

We avoid "far pointer" altogether and use the more explicit "pointer-to-**_Far**-*sometype*" or "**_Far**-pointer-to-*sometype*". With careful wording, English expressions of C attributed types can be unambiguous For example:

```
int i;                 // An int
int *pi;               // A pointer-to-int
_Far int fi;           // A far-int
_Far int *pfi;         // A pointer-to-far-int
int *_Far fpi = &i;    // A far-pointer-to-int
_Far int *_Far fpfi = &fi; // A far-pointer-to-
                           //  far-int
_Far int frff();   // A function returning a far-int
_Near int frni();  // A function returning a near-int
int *_Far frfpi(); // A function returning
                   //  a far-pointer-to-int
_Far int *frpfi(); // A function returning a
                   //  a pointer-to-far-int
```

Functions returning a distance-attributed type

Functions returning a distance-attributed type seem strange at first glance. What does it mean to return a type with the **_Far** attribute? We have said that **_Far** can specify a different addressing mode for an object, but a returned value is not an addressable object.

Rather than ignoring attributes of return types, MetaWare C/C++ interprets a function-returning-X-type... as a reflection on the nature of the addressing mechanism necessary to call the function, rather than as an implication about its return type. For example, a function-returning-**far-int** is really a **_Far**- function-returning-**int**.

In the Intel architecture, a function F can call function G that has the **_Near** attribute only if F is within a certain addressing range encompassing both F and G; if G had the **_Far** attribute it could be as far away as the architecturally supported maximum instead.

Functions **frff**, **frni**, and **frfpi** in the preceding example have attributed return types; the attribute is taken as affecting the function itself. But function **frpfi** does not have an attributed return type. Its return type is a pointer type and the base type of the pointer type is that which is attributed. Therefore, adding the attribute at the leftmost position of a function declaration generally does not cause attribution of the function. Placing the attribute nearest to the function name achieves this effect. For example:

```
char * (* _Far F())();
_Far char * (* G())();
```

F is a function returning a **_Far**-pointer-to-function-returning-pointer-to-**char**, and hence is interpreted as a **_Far** function returning a pointer-to-function-returning-pointer-to-**char**. G is a function returning a pointer-to-function-returning-pointer-to-**_Far-char**. G itself is not attributed.

Type and assignment compatibility

Two types T and T' that are compatible in the normal sense are also compatible if they are distance attributed. The same holds for assignment compatibility. Pointer types T and T' that are normally assignment compatible are also assignment compatible if either of their base types is distance attributed.

However, when values of the two types "meet" in an expression (such as assignment or comparison), conversion code might be required. For example, on Intel architectures, assigning a pointer-to-**_Near** to a pointer-to-**_Far** requires a conversion because the two types are not the same size.

For two function types, distance attributes behave like the ANSI-specified attributes **const** and **volatile**. Thus a function-returning-**_Far-int** is not compatible with a function-returning-**int**. As a (complicated) example, the cast below is necessary to assign `F` to function variable `FV`:

```
char * (* _Far (*FV)())();
char * (*        F  ())();
FV = (char * (*_Far(*)())()) F;
```

Here the cast converts the address of `F` to the same type as that of `FV`. For Intel architectures, code is added to convert the representation of the address of `F` to a value having the same size as that of `FV`'s type.

Explicit casts of pointer types can also cause conversion code to be generated if the base types have different distance attributes.

Compiler-Intrinsic Functions

MetaWare C/C++ contains a set of *compiler-intrinsic functions* that supply a number of basic trigonometric and other operations. Compiler-intrinsic functions do not need to be declared to be used.

See the *MetaWare C/C++ Programmer's Guide* for information about compiler-intrinsic functions supported for your target.

Inline assembly code

Some MetaWare C/C++ compilers allow you to insert assembly code directly into your C or C++ code, using the function **_ASM()**, the type qualifier **_Asm**, or both. See the *MetaWare C/C++ Programmer's Guide* for detailed platform-specific information about these two MetaWare C/C++ extensions.

Direct register assignment

MetaWare C/C++ supports assignment of variables directly to a specific register. See the *MetaWare C/C++ Programmer's Guide* for platform-specific information about how to use this extension.

Chapter 4 — Migrating from C to C++

In This Chapter

What is migrating?

The MetaWare C/C++ compiler supports the gradual introduction of C++ features into C programs. You can use almost all of C++, intermixing it with C, and still preserve the semantics of your C code.

C++ is almost, but not quite, a superset of C. A C++ compiler must reject some C programs. But with MetaWare C/C++, you can introduce C++ constructs in your C programs without affecting whether your C program text will compile, and without affecting the result when it does compile. We call this the *Incremental C++* approach.

C/C++ compatibility issues

The *MetaWare® C/C++ Language Reference* discusses C/C++ compatibility issues, and the ANSI C++ standardization committee has created a longer list of issues. We describe a few of the issues here to help you decide if you want to use Incremental C++.

The following examples present C code that works with a C compiler, but either fails to compile or works differently with a C++ compiler. All the examples compile and work correctly with the MetaWare C/C++ compiler's Incremental C++.

- **struct**s are scopes in C++. The following program fails to compile in C++ because `color`, `red`, `white`, and `blue` are local to the **struct**.

  ```
  struct s {
     enum color {red,white,blue} x;
     };
  // color and red are known here in C.
  enum color z = red;
  ```

  Instead, in C++ you can write the following:

  ```
  enum s::color z = s::red;
  ```

- C++ has additional keywords. The following example fails to compile in C++ because the reserved word **class** is used as a parameter name.

  ```
  type_handle make_primitive_type(type_class class,
                                  int len);
  ```

- **sizeof char** is different in C++. The following fragment prints nothing when compiled as C. It prints a message when compiled as C++, because the **sizeof** a character in C++ is typically one byte: not the same as **sizeof(int)**, as in C.

  ```
  if (sizeof('a') != sizeof(int))
     printf("different from C!\n");
  ```

- A tag is a type name in C++. The following is an error in C++ because **T** is a type name for **int *** and for the **struct**, so you have a duplicate declaration of `T`.

  ```
  typedef int * T;
  struct T { int x; };
  ```

Using incremental C++

With Incremental C++, you can use any C++ features except the following:

- templates
- class names (tags) redefined as something other than class names
- **typedefs** within classes

You can use constructors, destructors, overloaded functions, member functions, virtual functions, derived classes, virtual bases, and so forth. Some features require modified syntax, described in the rest of this chapter.

Libraries

The MetaWare C++ header files are designed for full C++, not Incremental C++, so you cannot use those files incrementally. If you need the features of the C++ library, access them through separate modules you write in full C++.

Specifying C++ level at the command line

An Incremental C++ program is a C program, and has the suffix `.c`. You use compiler option **-Hcpplvl** to enable Incremental C++ features at the level of C++ compilation you want. See the *MetaWare C/C++ Programmer's Guide* for information about using option **-Hcpplvl** to specify Incremental C++ levels.

Using special keywords and functions

You can access many Incremental C++ features by using keywords that do not intrude on the namespace of C programs, and by using slightly different syntax in some places. See Moving from incremental C++ to full C++ on page 62 for more information. Table 5 Incremental C++ keywords" shows the special Incremental C++ keywords.

Table 5 Incremental C++ keywords

Use this incremental C++ keyword	Instead of this C++ keyword
_Class	**class**
_Virtual	**virtual**
_Operator	**operator**
_Private	**private**
_Protected	**protected**
_Public	**public**
_Friend	**friend**
_Delete	**delete**
_New	**new**
_Inline	**inline**
_Anonymous	**none**

For example, you can write the following Incremental C++ program and compile it as a `.c` file with **-Hcpplvl=***1*:

```
_Class C {
   _Virtual ~C() { _Delete p; }
   _Friend struct s;
   C *p;
   };
struct s { int x; };     // ANSI C here
```

C++ features that do not involve C++ keywords are automatically available in Incremental C++, such as the initialization of static variables with non-constant expressions. For example:

```
struct s {int a,b; } X = {3,4};
int k = X.a + X.b;  // Generates initialization code
```

Accessing C++ features incrementally

You can use the following C++ features in Incremental C++:

- Alternate keywords
- Keyword **_Anonymous**
- Overloaded functions
- Special syntactic element **..**
- A **void** return type

Alternate keywords

Use **_Virtual**, **_Operator**, **_Private**, **_Protected**, **_Public**, **_Friend**, **_Delete**, and **_New** instead of the normal C++ keywords.

Keyword _Anonymous

Use keyword **_Anonymous** to specify an anonymous union. (In ANSI Standard C, there is no such thing.) For example:

```
_Anonymous union { int a,b; };
int c = a + b;    // a and b are visible
```

You can also use **_Anonymous** within a **_Class**, although it is not required; anonymous unions are accepted. With a **struct** or **union**, you must use **_Anonymous** if that is what you want:

```
_Class C {
   union {int a,b;};   // OK, anonymous union
   };
struct s {
   _Anonymous union {int a,b; };
      // _Anonymous union
      // Without _Anonymous, this would be a
      //  vacuous declaration, as in ANSI
      //  Standard C.
   };
```

Overloaded functions

Functions not contained within a **_Class** are *not* overloadable by default. To make them overloadable, surround such functions by `extern "C++" { ... }`. For example:

```
extern "C++" {
   void f() {...}
   void f(int) {...}
   };
extern "C++" void f(float) {...};
```

The preceding example declares three separate functions named `f`.

You must surround all declarations and definitions of a function `f` with **extern "C++"** for `f` to be overloadable. That is, you cannot have one declaration that is not overloadable and subsequent declarations that are.

You might consider surrounding your whole program with `extern "C++" {...}`.

Functions contained within a **_Class** *are* overloadable. You do not need to surround the **_Class** declaration with `extern "C++" {...}`.

Special syntax

You must employ the syntactic element "**..**" in several contexts to use certain C++ constructs that introduce parsing ambiguities or difficulties in the language. The "**..**" prevents these ambiguities and allows the programs to be parsed as Incremental C++ programs.

These are the contexts:

- Declaring a **const** or **volatile** function:

```
_Class s {
   void f() .. const;
   };
void s::f() .. const { /* body */ }
```

- Using a function-style cast:

```
_Class s { s(); s(int); };
s fnc(int i) { return i ? s..() : s..(3); }
```

- Using the C++ parenthesized initializer:

```
_Class s { s(int); };
main () {
   s x..(3);
   }
```

- Supplying arguments to **_New** after the type name:

```
_Class s { s(int); };
main () {
   _New s..(3);
   }
```

A void return type

To avoid a parsing problem introduced by C++, you must specify a **void** return type for constructors and for the definition of destructors. You could use a name such as `Tor`, and `#define` it to be **void**:

```
#define Tor void
_Class s {
   Tor s();
   ~s();            // Tor is not needed here.
   };
Tor s::s() { ... }
Tor s::~s() { ... }
```

This is necessary only when the constructor name is a **typedef** name. **struct**s and **union**s do not introduce **typedef** names, so you would not need `Tor` to specify their constructors:

```
struct s {
   s();         // OK
   ~s();        // OK
   };
s::s() { ... }
s::~s() { ... }
```

C++ features you cannot use in incremental mode

You cannot use the following C++ features in Incremental C++:

- You cannot use **typedef** names within structures:

```
_Class C {
   _Class D { };
   D *p;            // This works.
   typedef enum {red,white} color;
   color *c;        // This does not work.
   };
```

  Instead, move the **typedef** out of the structure.

- **_Class** names are introduced as **typedef**s, but their scope extends just as a C structure tag does. That is, the class name is not restricted to a class scope in which it might appear. If you avoid nested classes this is not a problem. If you use nested classes, the only problem is redefining the nested class name at the global scope:

```
_Class C {
   _Class D { };    // D is a typedef name.
   };
// D is still declared here, unlike full C++
int D;              // This is an error in
                    //  Incremental C++.
```

- **struct**, **union**, and **enum** tags do not become **typedef**s as they do in C++. If you want them to be **typedef**s, use **typedef**, as is common practice in C:

```
struct s { /* ... */ };
s *p;      // Invalid in C and Incremental C++
typedef struct t { /* ... */ } t;
t *p;      // Valid in C and Incremental C++
```

Moving from incremental C++ to full C++

Minor syntactic changes are required to convert an Incremental C++ program to a full C++ program. You can make it easy to change the syntax by using macros in your Incremental C++ programs. We recommend you use a set of #define statements, such as the following, including them in a suitable header file.

```
#if !__cplusplus
// Macros for Incremental C++:
#define Tor        void
#define class      _Class
#define virtual    _Virtual
#define operator   _Operator
#define private    _Private
#define protected  _Protected
#define public     _Public
#define friend     _Friend
#define delete     _Delete
#define new        _New
#define anonymous  _Anonymous
#define inline     _Inline
```

```
#define is          ..
#define init        ..
#else
#define Tor
#define is
#define init
#endif
```

Using macros `is` and `init` in place of **..** makes semantic sense, and you can use #define to get rid of them for full C++:

```
_Class s {
   void f() is const;
   s();
   s(int);
    };
void s::f() is const { /* body */ }
s fnc(int i) { return i ? s init() : s init(3); }
s x init(3);
void *p = new s init(3);
```

Chapter 5 — Iterators in MetaWare C/C++

In This Chapter

What is an iterator?

An *iterator* is a special kind of function that supplies values for the `for` variable(s) of an iterator-driven `for` loop. The `for` loop body is executed through a predefined function **yield()**, each time the iterator produces value(s) for the `for` variable(s). Recursion is possible with iterators because the function that drives the **for** loop does not have to return.

Benefits of iterators

The major benefit of iterators and iterator-driven `for` loops is that the algorithm to determine the values of the `for` variables for each loop iteration is defined separately in the iterator rather than in the `for`-loop construct itself. The same iterator can be invoked in many different `for` loops.

If, for example, you use iterator-driven `for` loops to access the elements of a data structure, iterators allow you to change the implementation of the data type without changing the implementation of the `for` loops.

- With an iterator, each `for` loop is driven by an independent computation that can maintain its environment across invocations of the loop.
- Without an iterator, the syntax accessing the structure is inside each `for` loop; therefore each `for` loop must be modified if the data structure changes, because the loops contain code specific to the data structure.

Iterator syntax and constraints

You invoke an iterator as follows:

```
for name[,name,...] <- iterator(arg_list) do
   ... // Body of the for loop, using the name
       // variables
```

An iterator is declared as follows:

```
void iterator(parameter_list) -> (yield_list) {
   ... // Body of iterator, with calls to
       // predefined function yield()
   }
```

Iterators yield one or more values. The declarative syntax for both the iterator input arguments and **yield()** types follows that of the input arguments to normal functions.

The following must match in number, order, and type:

- the for-loop variable list (*name***[,** *name*,…**]**)
- the arguments in the argument list (*arg_list*) with which the iterator is invoked
- the list of argument types with which function **yield()** is invoked (*yield_list*)

Arguments to iterators

The iterator's *parameter_list* has exactly the same constraints and semantics as the formal parameter list of other functions, except that MetaWare C/C++ additionally requires that this list use the ANSI Standard C prototype syntax. Old-style C function parameter declarations are not permitted. When the iterator is invoked in a `for` loop, the expressions passed to it must satisfy the same constraints as those of expressions passed to a function, given the same parameter list, and the parameters are passed in the same ways.

The *yield_list* must also use the ANSI Standard C prototype syntax. Here, however, the names of the parameters can be omitted, just as they can be omitted when a function is declared but not defined.

The yield list

The parameter list following the `->` in the iterator declaration is called the *yield list* (*yield_list*). The `for` variables of the `for` loop that invokes the iterator take on values of the types specified in *yield_list*, in the same order. The iterator supplies the values through a call to the function **yield()**, which is defined in the body of the iterator:

```
yield(E1,E2, ... En);
```

The following constraint must be satisfied:

> If **void yield(***yield_list***);** is a valid function declaration, then `yield(E1,E2, ... En);` must be a valid call to that function.

Using named parameters

Alternatively, the syntax of MetaWare C/C++'s named parameter association can be employed to yield the values, just as in a function call. *yield_list* must contain the names of the parameters (see Named Parameter Association on page 52). However, even if that notation is not used, it is useful to name the "yield parameters" for the sake of documentation.

Specifying a variable number of arguments

Like other functions, iterators can use the **...** notation to specify that they can take a variable number of arguments. The yield list can also use **...**, requiring the use of the `va_arg` macros within the `for`-loop body to access the remainder of the expressions yielded.

Argument types

Because the types yielded by an iterator are described by the yield list, there is no restriction on the types except that they must be types that can be passed to a normal function. An iterator can yield integers, structures, functions, and even iterators.

NOTE An iterator that computes the strongly connected components of a graph, and an example of its use, are provided in MetaWare C/C++ distributions in file `analyze.c`; this iterator's yield is itself an iterator that yields one set of strongly connected components.

Semantics of iterators

Each time an iterator executes a **yield()** statement, the body of the `for` loop is executed with its i^{th} `for` variable assuming the value of the i^{th} expression yielded, for all `1 <= i <= m`. When the iterator returns (just as a function can return), the `for` loop is terminated. A **break** or **goto** from the `for`-loop body also terminates the iteration.

The semantics of iterators and `for` loops that invoke them can be specified precisely in terms of an implementation using nested functions, as follows.

The compiler bundles up the body of each iterator-driven `for` loop and turns it into a function. Each so-bundled function is passed as an extra parameter to the iterator; the iterator is called once per `for` loop and receives as a parameter the function that used to be the body of that `for` loop. Each call to **yield()** within the iterator is translated to a call to the function that is its extra parameter.

More formally, the meaning of this code:

```
void I(Formal_parameter_list1) ->
```

```
      (Formal_parameter_list2);
   ...   // Calls to yield() in here
   }
for Name1,Name2,...Namen <- I(E1,E2,...Em) do
   ...   // for-loop body here
```

is precisely the same as the meaning of this code:

```
void I(void yield(Formal_parameter_list2)!,
                  Formal_parameter_list1) {
   ...   // Calls to yield() in here
   }

{// Would-be for loop
void For_loop_body(Formal_parameter_list3) {
   for (int i=1; i <= m; i++) {
      ...   // for-loop body here
      }
   }
I(For_loop_body,E1,E2,...Em);
}
```

where *Formal_parameter_list3* is *Formal_parameter_list2* but with the names $Name_1$, $Name_2$, ... $Name_n$ replacing the parameter names (if given) in *Formal_parameter_list2*. The **!** in the second example indicates that *Formal_parameter_list2* is a nested procedure; see Passing a nested function as a parameter on page 68.

The calls to **yield()** in the iterator `I` become calls to function `I`'s function parameter named **yield()** — the syntax is the same in both cases. Also, the `for`-loop body is made into a function and passed as an extra first parameter to the function `I`. Calling function `I` starts the loop; `I`'s return terminates the loop, unless there is a **goto** out of the loop body.

It is the function `I` that does the iterating. Iteration can occur by means of a contained loop or by `I` calling itself recursively, for example as it traverses some data structure. Each time it wants to yield something to the `for`-loop body, it does so by calling its **yield()** parameter and passing the value(s) of the `for` variable(s) to the next iteration of the loop.

Passing a nested function as a parameter

The **!** declaration guarantees we can pass a nested (non-global) function to the iterator. Because the `for` loop is guaranteed to lie within a function, when the `for`-loop body is transformed into a function, it is guaranteed to be a nested function. For example:

```
void Primes(int Lo,int Hi) -> (int ThePrime) {
      // We named the yield result.
      // Yields the primes in the interval
      // Lo..Hi
   int I,J;
   extern double sqrt(double);
   for (I = Lo; I <= Hi; I++) {
      // Asks if we can divide I evenly:
      for (J = 2; J <= sqrt(I); J++)
         if ((I/J)*J == I) goto Composite;
      yield(I);                  // I is prime.
   Composite: ;
      }
   }
...
```

```
for I <- Primes(1,100)
   do printf("%d is prime.\n",I);
```

This example has the same semantics as the following:

```
void Primes(void yield(int ThePrime)!, //Note use of !
            int Lo, int Hi) {
   // Yields the primes in the interval Lo..Hi
   int I,J;
   extern double sqrt(double);
   for (I = Lo; I <= Hi; I++) {
      // Ask if we can divide I evenly:
      for (J = 2; J <= sqrt(I); J++)
         if ((I/J)*J == I) goto Composite;
      yield(I);                 // I is prime.
   Composite: ;
      }
   }
...
   {
   void For_loop_body(int I) {
      printf("%d is prime.\n",I);
      }
   Primes(For_loop_body,1,100);
   }
```

How iterators work

Here is an example of a `for` loop written in ANSI Standard C:

```
int i;
for (i = 1; i <= 10; i++)
   printf("%d squared is %d\n",i,i*i);
```

Here is how to write this loop using an iterator:

```
void Upto(int Lo, int Hi) -> (int) {
   int i = Lo;
   while (i <= Hi) yield(i++);
   }
for i <- Upto(1,10) do
   printf("%d squared is %d\n",i,i*i);
```

The sequencing from one number to another (`1` (one) through `10` in this example) is programmed once in the **Upto** iterator. The syntax `for i <- Upto(1,10)` starts the iteration.

Each time the iterator **Upto()** calls the predefined function **yield()** with a value, that value is substituted for *i* in the body of the `for` loop, and the body is executed. When execution of the body is finished, control is returned to a point immediately after the call to **yield()**, and **Upto** continues its **while** statement.

Each call to **yield()** causes the `for`-loop body to be executed once. When **Upto** is finished invoking **yield()**, it returns just like a regular C function, and the invocation of **Upto** (and therefore the `for` loop itself) is complete. Control then passes to the statement after the `for` loop.

The syntax **->(int)** in the header of the iterator definition is all that distinguishes it from a normal function. After the **->** appears a prototype-form parameter list specifying the type(s) of the results yielded by the iterator. Here, **Upto** yields an **int** — one for each execution of the `for`-loop body.

Within the definition of an iterator, the predefined function **yield()** is defined. Outside an iterator you can use the name **yield** for any purpose except a function name.

Some uses for iterators

This section presents some programming situations where you might want to consider using iterators instead of regular `for` loops, and introduces a few of the programming problems that can be solved with iterators.

Iterating over the elements of a set

Although iterators are quite general, they are probably most useful for iterating over each element in some set. The set is often the entire contents of some data structure, but can be restricted to those elements in a data structure that meet certain conditions. However, the set does not have to be stored in a data structure; for example, an iterator can be used to provide the prime numbers up to 100 by computing and yielding them.

List processing

Here is a loop that processes each element in a list:

```
for e <- each_element(list) do {
   ... // Process element e
   }
```

In the loop, *e* is the single `for` variable. Each time through the loop, `e` is given a value yielded by the iterator **each_element**. (In this case, we would expect `e` to have a different value each time through the loop, but this is not always so.) `e` is really a parameter to the loop body. It is not visible outside the loop body and is instantiated anew each time through the loop. Assume the following declaration of an **`element`**:

```
typedef struct element {
   ...  // Some data
   struct element *next;
   } *element;
```

The iterator **each_element** might look like this:

```
void each_element(element e) -> (element) {
   while (e != 0) {
      yield(e);
      e = e->next;
      }
   }
```

Each time the predefined function **yield()** is called, control passes to the `for` loop that invoked **each_element**. The `for` variable **`element`** takes on the current value of *e* for that pass through the loop body. At the bottom of the `for` loop, control returns to **each_element** at the statement following the call to **yield()**.

Changing the implementation of a data type

Suppose in some program you make heavy use of a sorted linked list, and to speed up your code, you decide to replace the list with a binary tree. That would mean not only replacing the code that implements the list (the insert, delete, and sort functions, for example) but also finding all the places in the program where the list gets traversed. You would have to replace all the `for` loops that are some variation on the following:

```
for (listptr = head; listptr != NULL;
     listptr = listptr->next) {
   ...  // Uses list element
   }
```

On the other hand, if you use an iterator, you would still have to replace the code that implements the list, but you would not have to touch the code that traverses the list. Your loops would look something like this (both before and after you changed over to a binary tree):

```
for listptr <- traverse_list(head) do {
   ...  // Uses list element
   }
```

Traversing a tree

A **for** loop is a natural way to conceptualize accessing each item in a data structure. You can see how the process works if you write pseudo-code for some loop that processes every node in a tree, such as:

```
for each node in tree {
   ... // Processes node
   }
```

However, if you actually wrote such a loop in ANSI Standard C, you would probably not use a `for` loop. Trees are naturally traversed recursively, and the C language `for`-loop construct cannot naturally express a recursive computation. However, a MetaWare C/C++ iterator-driven `for` loop can express recursive computations as easily as any other kind; the resulting code looks almost identical to the pseudo-code:

```
for n <- each_node(tree) do {
   ... // Processes n
   }
```

The iterator **each_node** can be recursive, and a recursive algorithm can easily be used to generate values for variables.

Recursion and code clarity

In the list-traversal examples in Some uses for iterators on page 70, you might complain that all iterators do is make nice, straightforward `for` loops complicated. (We have, of course, done more than that, because now a list implementation can be changed without changing the `for` loops that process the list.) But consider a different problem: iterating through the nodes of a binary tree, where the tree is implemented using a data structure that for each node **`N`** has a pointer to the left and right subtrees of **`N`**.

A recursive tree walk

The obvious way to obtain the nodes of such a tree is by a simple recursive tree walk. However, this is not possible using ANSI Standard C `for` loops. Imagine having to translate the following pseudo-code into C:

```
Process the tree:
   do some stuff to the tree
   for each node in the tree
      print the node
   do some other stuff to the tree
```

Because you cannot use a `for` loop to access the nodes of the tree, you would probably package the node-printing process into a routine and pass that routine to be executed by a tree-walking routine:

```
typedef struct node {
```

```
   ... // Some data
   struct node *Left, *Right;
   // Left and right subtrees
   } *Node;
void Print_node(Node N) {
   printf("Node is ");
   ... // Code to print a Node
   }
void Each_node(Node N, void Doit_toit(Node N)) {
   if (N == 0) return;
   Each_node(N->Left, Doit_toit);
   Doit_toit(N);
   Each_node(N->Right, Doit_toit);
   }
Process_the_tree(Node Root) {
 ... // Does some stuff to the tree
   Each_node(Root, Print_node);
   ... // Does some other stuff to the tree
   }
```

Rearranging the computation this way allows the recursive tree walk to occur, but there is a cost: the simple `for` loop in the pseudo-code clearly expresses that a computation is being done on each element of the tree data structure; this is no longer as apparent in the body of **Process_the_tree**.

To find out what is being done you now have to look outside **Process_the_tree**, in **Print_node**. In this simple example, the choice of good names assists a great deal in promoting understanding of the code, but if you must deal with real problems, the code is more complex and the names are not perfectly explanatory.

Using iterators for the tree walk

Here is a solution to the same problem, using iterators:

```
void Each_node(Node N) -> (Node) {
   if (N == 0) return;
   // Walks the subtrees
   for L <- Each_node(N->Left)  do yield(L);
   yield(N);
   for R <- Each_node(N->Right) do yield(R);
   }
Process_the_tree(Node Root) {
   ... // Does some stuff to the tree
   for Node <- Each_node(Root) do {
      printf("Node is ");
      ... // Code to print a Node
      }
   ... // Does some other stuff to the tree
   }
```

Notice that the code for **Process_the_tree** mirrors the pseudo-code extremely closely, yet unlike the ANSI Standard C version, the algorithm that determines the successive values of `Node` in the `for`-loop body is recursive. This is a major increase in expressive power. The algorithm can be expressed in a more natural fashion, making the code easier to write, understand, and modify.

The body of **Each_node** is not very elegant (or efficient) as written. The recursive calls do nothing but re-yield a result already yielded at a deeper level of recursion.

Using nested functions (another MetaWare C/C++ extension) and the fact that calls to **yield()** can occur within functions nested within iterators, you can produce an elegant version:

```
void Each_node(Node N) -> (Node) {
   void P(Node N) {
      if (N == 0) return;
      // Walk the subtrees.
      P(N->Left);
      yield(N);
      P(N->Right);
      }
   P(N);
   }
```

Work done by the programmer in the ANSI Standard C version — packaging the body of the pseudo-code `for` loop as a function — is done instead by the MetaWare C/C++ compiler in the iterator version.

Replacing macros with iterators

Iterators are also useful when the iteration algorithm, although not recursive, is complex. Often in this circumstance you would design a macro to make up for the shortcomings of the C `for` loop.

Consider the following example, taken from the MetaWare C/C++ compiler. The problem is to sequence through the objects that overlap a given object `obj` in memory. Prior to the addition of iterators, a macro was required to simulate the iteration.

```
#define for_each_overlapping_object(o,obj,p){\
   struct obj_entry *_op = &objtab[obj];\
   obj_class_type _class = _op->class;\
   long _len = _op->len;\
   long _disp = _op->disp;\
   ushort _word = _op->un.word;\
   bool is_deref = (ea_DEREF & _op->flags) != 0;\
   bool is_adr = ((ea_ADR|ea_GLOBAL)
      & _op->flags)!= 0;\
   register struct obj_entry *p; int o;\
   for(o=1,p=objtab+1;o<=last_object;o++,p++){\
   if (is_deref && \
       ((p->flags&ea_DEREF) ||\
           (p->flags&(ea_ADR|ea_GLOBAL))!=0 &&\
            p->xlen >= _op->xlen &&\
               _op->disp < p->xlen)||\
       is_adr  && (p->flags&ea_DEREF)!=0 &&\
          p->xlen<= _op->xlen|| \
          p->class == _class && \
       (p->un.word == _word ||\
       (p->flags&_op->flags&ea_TEMP)) &&\
       (p->disp<=_disp && p->disp+p->len > ||\
        _disp+_len > p->disp && _disp < p->disp)){
```

`obj_class_type` is defined elsewhere; due to data abstraction, in the preceding example you do not need to know exactly what an `obj_class_type` is. Given an `object_index`, the macro in the example iterates over each object overlapping with that `object_index`. The macro is invoked as follows:

```
object_index obj;
...
for_each_overlapping_object(o,obj,p)
   ... do things with o and p ...
   }}}  // Required to match {'s in macro
```

The #define defines the variables o and p whose names are arguments to the macro.

With this approach, an enormous amount of code is reproduced each time the macro is invoked; and every time a change is made to the macro, every module where it appears must be recompiled. An iterator requires much less maintenance; when a change is made, only the iterator itself must be recompiled, not its "client" for loops.

Another advantage to using iterators is that the iterator can declare the types of its parameter and yielded results, providing improved type checking at the iterator invocation. The following example shows the same routine implemented as an iterator:

```
void Each_overlapping_object(object_index obj)
        -> (int o, struct obj_entry *p) {
   struct obj_entry *_op = &objtab[obj];
   obj_class_type _class = _op->class;
   long _len = _op->len;
   long _disp = _op->disp;
   ushort _word = _op->un.word;
   bool is_deref = (ea_DEREF & _op->flags) != 0;
   bool is_adr = ((ea_ADR|ea_GLOBAL) & _op->flags)
                 != 0;
   register struct obj_entry *p; int o;
   for(o=1,p=objtab+1;o<=last_object;o++,p++){
      if (is_deref &&
            ((p->flags&ea_DEREF) ||
            (p->flags&(ea_ADR|ea_GLOBAL))!=0 &&
         p->xlen >= _op->xlen &&
         op->disp < p->xlen)||
         is_adr && (p->flags&ea_DEREF)!=0 &&
         p->xlen<= _op->xlen||
         p->class == _class &&
         (p->un.word == _word ||
            (p->flags&_op->flags&ea_TEMP)) &&
         (p->disp<=_disp && p->disp+p->len >_disp ||
            _disp+_len > p->disp &&
            _disp < p->disp)) {
            yield(o,p);
         }
      }
   }
```

The iterator in this example is invoked as follows:

```
object_index obj;
...
for o,p <- Each_overlapping_object(obj) do
    ... do things with o and p ...
```

The iterator yields two results for each execution of the for-loop body: the **int** o and the `struct obj_entry *p`.

Advantages and disadvantages of iterators

You must balance the advantages of reusable code and lower maintenance against the reduced execution speed that can result from using iterators.

Reusable code

The major benefit of iterators is that the algorithm to determine the values of the `for` variables for each loop iteration is defined separately in the iterator. Unlike the ANSI Standard C `for` loop, the iterator can do an arbitrary amount of computation and can automatically maintain its environment across passes through the body of the loop, because it suspends rather than returns when it provides the loop body with values.

Once an iterator is written, it can be invoked by as many loops as desired without repeating the code that provides the values used within the loop (as compared with using many copies of standard `for` loops, which involve repeating the code for each loop).

Information hiding

The structured-programming term *information hiding* refers to placing information only where it is needed, and keeping information from where it is not needed. Information hiding promotes better and more easily maintained programs, and reduces recompilation of modified programs. Iterators promote information hiding.

For example, in a module implementing a tree data type, you might want to hide all the details of traversing trees within functions defined in that module. In ANSI Standard C, however, traversing the tree generally means exposing the tree data structure in any `for` loop sequencing through nodes of the tree. Instead, you can use iterators to localize the sequencing techniques and their required data-structure access to within the tree module itself. The `for` loop only needs to invoke an appropriate iterator and does not need to be concerned with the representation of trees.

Execution overhead

The major drawback of iterators is that they are slow, compared to standard `for` loops, because they involve the overhead of function calls. You lose speed by placing the computation of the iteration in a separate function. A loop that executes *n* times involves *n*+1 function calls related to loop overhead. The loss of speed occurs not only due to the function linkage but also because the body of the `for` loop becomes a nested function, and any variables in the body's parent accessed from the body cannot reside in high-speed machine registers, due to the current state of optimizer technology. For example:

```
int Sum = 0;
for I <- Upto(1,10) do
   Sum += I;
printf("Sum of 1 to 10 is %d\n", Sum);
```

This is implemented as:

```
int Sum = 0;
void Body(int I) {Sum += I;}
Upto(Body,1,10);
printf("Sum of 1 to 10 is %d\n", Sum);
```

Here, `Sum` cannot reside in a register because it is accessed from nested function **Body**, so all accesses to `Sum` are slowed. Therefore, if you require extreme speed, do not use iterators, especially for simple `for` loops where the iteration algorithm is straightforward and there is no need to break it out in an iterator.

Chapter 6 — C++ example program

In This Chapter

Example program

This chapter presents a program called `example.cpp` that illustrates some important features of the C++ language. These features are discussed briefly and cross-referenced to more detailed information in later chapters. In this chapter we present and discuss `example.cpp` in a series of code fragments. This simple program does the following:

- establishes an employee database and reads some predefined records from a data file `example.dat`
- enables you to select and view record headers or entire records
- enables you to add or delete records

Part 1: C++ comments, const declarations, and function prototypes

```
// File example.cpp - C++ example program
// Copyright 1998-2007 ARC® International
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
// Field-Length constants:
const Choice_len = 10;
const Agency_len = 30;
const City_len   = 15;
const Date_len   = 10;
const Dept_len   = 30;
const F_N_len    = 10;
const L_N_len    = 15;
const Phone_len  = 15;
const SSN_len    = 15;
const State_len  = 20;
const Status_len = 10;
const Street_len = 30;
const Title_len  = 30;
const Type_len   = 10;
// Function prototypes:
int get_num(int&);
int get_num(long&);
int get_num(float&);
void get_line(char*,int);
void menu();
```

Part 1: C++ comments, const declarations, and function prototypes demonstrates the C++ comment style. Double forward slashes comment text to the end of the line. Like ANSI Standard C, C++ uses the keyword **const**. **const** is the preferred way to define constants, and is meant to replace the #define preprocessor directive in most cases. Also note the use of function prototypes, a feature familiar to ANSI Standard C users. Function prototypes are required in C++. Prototyping makes programs easier to read and maintain.

The `iostream.h` and `fstream.h` header files contain I/O stream library information that C++ needs to perform standard I/O and file I/O, respectively. Streams are not files, but you can think of them as similar to file pointers in ANSI Standard C. See the *MetaWare C++ I/O Streams Library Reference* for more information about streams.

Part 2: Declaring a class

```
class employee {
protected:
   char *Soc_Sec_Num;
```

```
   char *Home_Phone;
   char *Street;
   char *City;
   char *State;
   char *Type;
   char *Last_Name;
   char *First_Name;
   char *Title;
   char *Department;
   char *Work_Phone;
   char *Start_Date;
   void print_general();
   virtual void getinfo_spec()=0;
   void getinfo_general();
public:
   employee();
   virtual ~employee();
   void print_header();
   virtual void print_report()=0;
   };
```

C++ introduces a new concept known as a *class*. In C++, classes, structures, and unions can all contain both data and functions that operate on the data. In effect, a class is simply a **struct** with functions. Classes provide a more direct association between code and data than is possible in ANSI Standard C. Part 2: Declaring a class declares **class** `employee`.

Just as C **struct** types do, classes introduce new types in C++. A *class object* is an object whose type is a class type. This is sometimes referred as an *instance* or *instantiation* of a class. In C++, object-oriented programming is accomplished by dealing with class objects.

Functions declared in a class are called *member functions* of that class. For example, **print_general()** and **getinfo_spec()** are member functions of `employee`. Member functions have special access rights to class data.

The `employee` class introduces three new C++ keywords: **public** and **protected**. A third, **private**, will be introduced in classes derived from `employee`. These keywords determine access to class members.

- **public** members (those whose declarations follow the keyword **public**) are available to any function.
- **private** members (those whose declarations follow the keyword **private**) can be accessed *only* by members of the same class.
- **protected** members (those whose declarations follow the keyword **protected**) can be accessed by member functions of the same class and by certain other classes called *derived* classes.

The practice of allowing only certain functions to access a data structure is called *encapsulation*. See Access control in C++ on page 107 for more information about access control.

Part 3: Class constructors

```
employee::employee() {
   // This constructor allocates memory for class
   // members.
   Last_Name   = new char [L_N_len];
   First_Name  = new char [F_N_len];
   Title       = new char [Title_len];
   Department  = new char [Dept_len];
   Work_Phone  = new char [Phone_len];
```

```
    Soc_Sec_Num = new char [SSN_len];
    Home_Phone  = new char [Phone_len];
    Street      = new char [Street_len];
    City        = new char [City_len];
    State       = new char [State_len];
    Type        = new char [Type_len];
    Start_Date  = new char [Date_len];
    };
employee::~employee() {
    // This destructor frees memory allocated by a
    // constructor.  This destructor is implicitly
    // called by the destructors for classes
    // temp and permanent.
    delete Last_Name;
    delete First_Name;
    delete Title;
    delete Department;
    delete Work_Phone;
    delete Soc_Sec_Num;
    delete Home_Phone;
    delete Street;
    delete City;
    delete State;
    delete Start_Date;
    };
```

The first **public** member function in `employee`, shown in Part 3: Class constructors, is a special function known as a *constructor* that allocates memory for objects of class employee and initializes them before you use them. A constructor is called in the following circumstances:

- when the program executes the declaration of an object
- when you create an object with **new**

In this example, the constructor **employee()** allocates arrays of characters to hold information about the employee. Each constructor has the same name as its class. A constructor that can be called with no arguments is a *default constructor*. The constructor defined in the `employee` class is a default constructor.

The `employee` class also contains a *destructor*. A destructor is invoked just before an object goes out of existence, so that clean-up can be performed before the object ceases to exist. A destructor's name is the name of the class, prefixed with the ~ character. For example, **~employee()** frees up the memory allocated by constructor **employee()**. A destructor for an object is called in the following circumstances:

- when the program leaves the object's scope
- when you delete an object with **delete**

See Special C++ members on page 115 for more information about constructors and destructors.

C++ introduces two new functions to deal with storage allocation: **new** and **delete**. **new** calls the constructor to allocate memory for that object. **delete** destroys an object created by **new** and frees up the storage. **delete** invokes the destructor (if any) for the object pointed to. See Controlling storage allocation on page 130 for more information about storage allocation.

Part 4: Derived classes

```
class temp : public employee {
private:
```

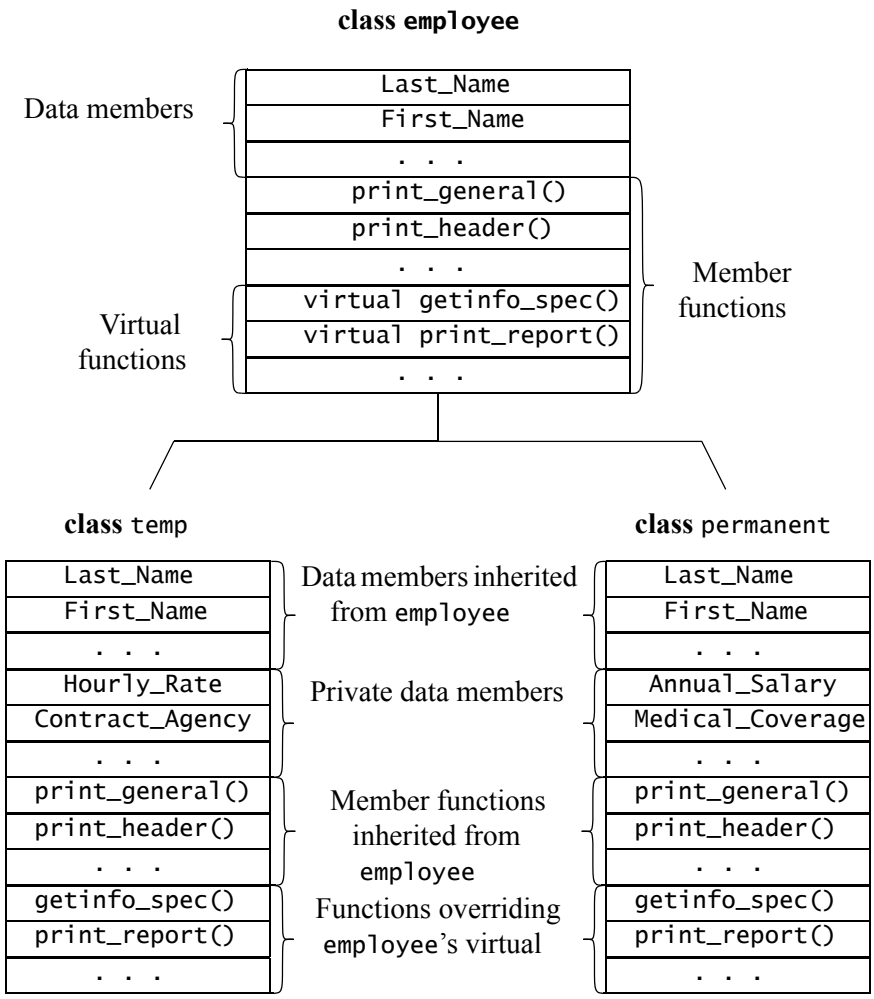
```
   void getinfo_spec();
   float Hourly_Rate;
   char *Agency;
   int Contract_Duration;
public:
   temp();
   temp(ifstream&);
   ~temp() { delete Agency; };
   void print_report();
   };
class permanent : public employee {
private:
   int Medical_Coverage;
   long Annual_Salary;
   void getinfo_spec();
public:
   permanent();
   permanent(ifstream&);
   ~permanent() {};
   void print_report();
   };
```

Part 4: Derived classes illustrates the classes `temp` and `permanent`. These classes represent data for temporary and permanent employees, respectively. As in the real world, “temps” and “permanents” are types of employees. These classes inherit the attributes of generic employees (name, address, and so on), but add their own specific attributes. Class `employee` is a *base class* and classes `temp` and `permanent` are *derived classes*. They inherit the **protected** and **public** members of class `employee`.

In addition, the `temp` class adds its own **private** members, such as `Hourly_Rate`, `Agency`, and `Contract_Duration`. Similarly, the `permanent` class adds its own members such as `Medical_Coverage` and `Annual_Salary`, which apply only to permanent employees (see Figure 1 Inheritance of data members and member functions on page 82).

See Derived and base classes on page 97 for more information about inheritance.

Figure 1 Inheritance of data members and member functions



Part 5: New class Node

```
class Node {
private:
   Node *next;
   employee *empl;
   static int records;  // static class member:
                        //  number of Nodes in list
public:
   // Default constructor:
   Node()  { empl = 0; next = 0; };
   // Second constructor with inlined code:
   Node(employee *new_emp)  {
      empl = new_emp;
      next = 0;
      };
   // Overloaded operators:
   void operator+=(employee*);
   int operator-=(int);
   // Friend functions:
```

```
   friend void headers(Node&);
   friend void view_record(Node&);
   };
int Node::records = 0;    // Initializes record count
   // Function prototypes:
void add(Node&);
void remove(Node&);
void view_record(Node&);
   // Overloaded operator +=:
void Node::operator+=(employee *new_emp) {
   // Appends a node to the linked list
   Node *last = this;
   while(last->next)
      last = last->next;
   last->next = new Node(new_emp);
   records++;
   };
   // Overloaded operator -=:
int Node::operator-=(int choice) {
   // Removes a node from the linked list
   int i=1,result;
   if(choice >= 1 && choice <= records) {
      Node *last = this;
      while (i < choice)  {
         last = last->next;
         i++;
         };
      Node *hold = last->next;
      if (hold)  {
         last->next = (last->next)->next;
         delete hold;
         records--;
         };
      result = 0;
      }
   else result = 1;
   return result;
   };
void add(Node& first) {
   // Adds a node to the linked list
   char      choice_str[choice_len],choice;
   cout << "temporary or permanent? ";
   get_line(choice_str,choice_len);
   choice = choice_str[0];
   switch(choice) {
      case 't':
      case 'T':
         first += new temp;
         break;
      case 'p':
      case 'P':
         first += new permanent;
         break;
      default:
         break;
      };
   };
```

Part 5: New class Node introduces a new class, `Node`. Each instance of `Node` is one node in a linked list of pointers to `employee` objects.

`Node` declares two operator functions, += and -=.

The operator += is declared like this:

```
void Node::operator +=(employee *new_emp);
```

This function appends the `employee` object to the linked list. The += operator is invoked in **add()** by this line:

```
first += new temp;
```

Operator functions make code easier to read because they convey their meaning more concisely than normal function names.

The -= operator function removes an employee record from the database; `choice` is the record number to delete:

```
first -= choice;
```

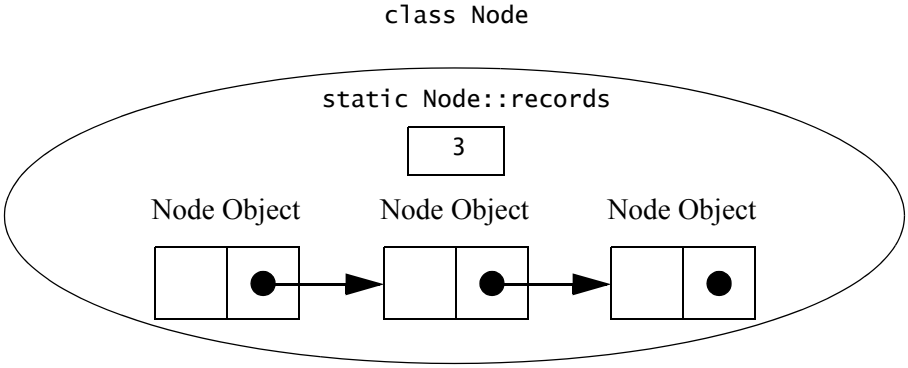
Because += and -= already exist in C++ and operate on other types (**int**, **float**, and so on), these operators are said to be *overloaded*. Overloaded operators have the same name, but are distinguishable by a unique parameter list. Different versions of += and -= are called, depending on how they are used. See Function overloading on page 135 for more information about operator overloading.

Note this line in the definition of `Node`:

```
static int records;   // static class member
```

The keyword **static** indicates that there is exactly one data member `records` for the class `Node` (Figure 2 Static data members). **static** data members are used by objects of a class to communicate with one another. `Node` uses data member `records` to keep a count of the number of nodes in the linked list.

Figure 2 Static data members



`Node` declares two **friend** functions:

```
friend void headers(Node&);
friend void view_record(Node&);
```

friend functions have the same access rights as member functions, without being declared as members. **friend** functions are used in `example.cpp` to grant non-member functions **headers()** and **view_record()** access to class `Node`. **friend** functions are also useful when one function must operate on two separate classes, and you do not want to duplicate the function. See Friends on page 113 for more information about **friend** functions.

The function **operator+=** uses the hidden *this* parameter:

```
Node *last = this;
```

Each member function has a hidden parameter, denoted by the keyword **this**.
this points to the class object whose members the function manipulates. See Non-static member functions on page 95 for more information about **this**.

The following line appears in the `employee` class:

```
virtual void print_report()=0;
```

print_report() is a *virtual function*. **virtual** function bodies are not inherited from the base class and can be overridden by derived classes. In this case, the **print_report()** function body is empty. **virtual** functions allow this call in function **view_record()**:

```
(current->empl)->print_report();
```

This call invokes **print_report()**, but the type of object pointed to by `current->empl` is unknown until run time. At run time, the program determines whether the object is `temp` or `permanent`, and calls the appropriate **print_report()**. This is known as *late binding*, and is one of the most powerful features of C++. The concept of sharing the same name throughout a class hierarchy is known as *polymorphism*. See Virtual functions on page 100 for more information about **virtual** functions.

The declaration of **getinfo_spec()** in class `employee` is set to `0` (zero) to indicate it is a *pure virtual function*:

```
virtual void getinfo_spec()=0;
```

The presence of one or more pure virtual functions makes a class an *abstract class*. Abstract classes, such as `employee`, cannot be instantiated; they can be used only as bases for inheritance. `employee` is used only as a base for the derivation of `temp` and `permanent`, not to create `employee` objects.

Chapter 7 — C++ classes

In This Chapter

Introduction

The class is the foundation of C++'s object-oriented programming model. Understanding the properties of classes is crucial to effective use of C++. With classes you can define new types where code and data are present in the same data structure and access to data is restricted to a specific set of access functions.

In discussing classes, this chapter uses the term *object* to refer to an area of storage occupied by a particular instance of a class or class member.

By restricting access to data within a class you *encapsulate* the data, as though putting a protective covering around it to prevent unauthorized access. In a **struct** or **union**, any function in the proper scope can manipulate the **struct** or **union** members. But in a class, by default, the class members cannot be read or written to except by other members of the same class.

C programmers should already be somewhat familiar with classes; a subset of classes encompasses exactly the ANSI Standard C **struct**. Think of a class as a "super-**struct**", that is, a **struct** with many more features and possibilities than plain a C **struct**. The power of classes is essential to the object-oriented design of C++. But in C++ you are not forced into programming with all that power; you can stick to the plain C subset if you wish. In fact, the **class** keyword is really largely unnecessary — you can use **struct** and **union** to perform the same functions as classes in C++ — only the default access rules for members are different.

Class declarations and definitions

This is the basic syntax for a **class** definition:

```
[_Packed|_Unpacked] {class|struct|union}
   [identifier][base_list]
   {{member_declaration|access_specifier :}...}
```

Defining a class

This is the form of a **class** definition:

```
class_head {[member_list]}
```

The components have the following meanings:

member_list is:

```
{member_declaration|access_specifier :}...
```

member_declaration is one of the following:

```
[decl_specifiers][member_declarator_list] ;
function_definition [;]
qualified_name ;
```

member_declarator_list is:

```
member_declarator [, member_declarator]...
```

member_declarator is one of the following:

```
declarator [pure_specifier]
[identifier] : constant_expression
```

class_head is one of the following:

```
class_key [identifier][base_spec]
class_key class_name [base_spec]
```

class_key is:

```
{struct|union|class}
```

base_spec is:

```
: base_list
```

base_list is:

```
base_specifier [, base_specifier]...
```

base_specifier is one of the following:

```
complete_class_name
virtual [access_specifier] complete_class_name
access_specifier [virtual] complete_class_name
```

class_name is an identifier that is the name of a previously declared class.

access_specifier is an access-control level:

```
{private|protected|public}
```

complete_class_name is:

```
[::] qualified_class_name
```

qualified_class_name is:

```
class_name [:: qualified_class_name]
```

pure_specifier is = **0**

NOTE In a **class** definition, **class** and **struct** can be used interchangeably, except that the use of **class** in a definition implies different default access rules (see Access on page 108).

Example 16 A class definition shows many facets of a class definition.

Example 16 A class definition

```
class A : public B, virtual C {
   // B is a public base class.
   // C is a virtual base class.
   // First, some declarations that are also
   //  allowed in structs and unions in ANSI C:
   int x;            // A normal member
   float f[10];      // Another normal member
   unsigned j:3;     // A bit field
   enum color {red, white, blue}; // An enumeration
   // The rest of these are allowable only in C++:
   void f();         // A function member
   public:           // An access specifier
   int sqr() {       // A function member, together
      return x*x;    //  with its definition
      }
   ;;;;;;           // Empty member_declarations
   typedef int *PI;  // A typedef member
   static char c;    // A static member
   A *next;          // Use of the class name A
                     //  declaration
```

```
    class A *next2;   // The word "class" is
                      //  redundant here, but
                      //  allowed.
    private:          // Another access qualifier
    B::g;             // Adjusting access of g,
                      //  inherited from B
    // The only allowable case of an initializer:
    virtual void func() = 0;
    // Some *** invalid *** declarations:
    auto local_var;   // Cannot use auto
    int_array[10];    // Data type is omitted;
                      //  "int int_array[10];"
                      //  would be correct.
    void func2() = 1; // 1 is not valid.  Also,
                      //  "virtual" is omitted.
    A sub_a;          // Cannot contain an instance
                      //  of an incomplete type
    };
```

Every class definition defines a new type different from all other types. The type is incomplete until the ending curly brace (}) of the definition is reached; it is then made "complete".

Empty member declarations

It is possible for a *member_declaration* to produce nothing, because all parts of it are optional, so **;;;;;;** can appear within a class definition.

Scope of class members

A class definition introduces a new scope at the occurrence of **{**, and the scope closes at the corresponding **}**. This means that names declared as members are local to the class definition. The class definition corresponds to the traditional ANSI Standard C **struct** or **union** declaration. But in C++, you can also declare and optionally define functions within the class that allow access to private names declared in the class definition. You can also adjust access to names inherited from base classes and specify so-called *static members* of a class (see Static members on page 93).

Local classes

You can declare a class within a block, such as a function definition. Such a class is called a *local class*. The name of a local class is in the scope of its enclosing block, and cannot be "seen" outside the function. If you declare a local class, its members can use only type names, **static** variables, **extern** variables, functions, and enumerators from the enclosing scope level. For instance, accessing **auto** variables declared in the enclosing function is invalid:

```
void func() {
   int i;
   enum a {b,c};
   static int j;
   struct s {
      int a[i];          // Use of i not valid
      enum a x;          // OK
      void s_member() {
         j = 3;          // OK
         x = c;          // OK
         func();         // OK
         }
      };
   }
```

Nested classes

A class declared inside another class is a *nested class*. The class name of a nested class is local to its enclosing class. You can use only type names, static members, and enumerators from the enclosing class in a nested class declaration, unless you explicitly use pointers or access operators.

NOTE Member functions of a nested class do not have any special access to members of the enclosing class, and member functions of the enclosing class have no special access to members of the nested class.

Class members

A *class member* is a declared type definition, variable, or function whose scope is restricted to its containing class. Access to a particular class member can be restricted using access qualifiers (see Access on page 108).

Rules for declaring members of a class

The following rules apply to member declarations:

- In a member declaration you cannot use **auto**, **extern**, or **register**.
- You can omit the data type in a member declaration; if you do, **int** is assumed.
- You can use the `= 0` syntax in a *member_declarator* only if the name declared is a **virtual** function type. Doing this declares a *pure virtual function*. See Virtual functions on page 100 for more information about **virtual** functions.
- The type of a member of a class type must be complete. The class type is incomplete until the ending curly brace (}) of the definition is seen; it is then made "complete". This prevents a class from containing an occurrence of itself; for example:

```
class s {
   s x;         // This is not valid.
   };
```

This declaration results in an "infinite" definition: **s** contains an **s** which contains an **s** which contains an **s**, and so on. However, a class can contain a pointer to itself, as shown in the following example:

```
class boy {
   int height;
   int weight;
   enum color {blue, brown, black,green, hazel };
   boy *next;  // A pointer to an object of this
               //  class type
   }
```

Simple members

A simple class member is the same as a member in a **struct** or a **union** in ANSI Standard C.

Rules for declaring simple members

The following rules apply to simple member declarations:

- Specifiers cannot be storage-category qualifiers or function specifiers.
- You cannot use `= 0` in a *member_declarator*.
- The member is not of a function type.
- The declarator is required in the *member_declaration*.
- When the declarator is omitted, a useful member declaration occurs only when the specifiers specify an enumeration or class definition.

Storage for simple members

If a class type is declared using **struct** or **class**, each object of the class type contains a separate copy of all simple members, and the **sizeof** value of the class type is big enough to accommodate all the members.

If a class type is declared using **union**, the **sizeof** value of the class type is large enough to accommodate the largest member, and each member shares storage with the other members. In particular, the address of each non-bit-field member is the same as any other non-bit-field member. You can think of a union as a class whose member offsets are all `0` (zero), and that is big enough to hold any of its members.

Example 17 A `struct` declaration

```
struct A {
   int x;
   float f[10];
   unsigned j:3;
   };
main() {
   A a1, a2, a3, *a4 = &a3;
   a1.x = 3;      a2.x = 4;      a4->x = 5;
   a1.f[4] = 3.3; a2.f[6] = 4.4; a4->f[7] = 5.5;
   a1.j = 1;      a2.j = 2;      a4->j = 3;
   }
```

In Example 17 A struct declaration the declaration of a **struct** is the same as in ANSI Standard C. If this example declared `A` as **union** `A` instead of **struct** `A`, the address of `x` and of `f` would be the same; that is, `x` and `f` would overlap.

Declaring bit fields

If the declarator is omitted in a bit-field declaration, no storage is allocated, but the effect is to direct placement of subsequent members at the beginning of the next bit-field container.

When the declarator is omitted, a useful member declaration occurs only when the specifiers specify an enumeration or class definition. For example:

```
struct A {
   int;              // Not useful, nothing happens
   enum color {red, white, blue};  // An enumeration
                                   //  definition
   struct B {                      // A nested class
      int x;
      };
   };
```

This example is not much different from ANSI Standard C, either. However, the difference between C and C++ here is that in C++, the type names `color` and `B` and the enumeration literals `red`, `white`, and `blue` are local to `A`. In ANSI Standard C those names are visible outside the **struct**. For example:

```
// ANSI Standard C usage
struct B x;
enum color c = red;
```

Accessing names with the scope resolution operator

In C++, the names `color` and `B` are not visible. However, they can be accessed via the C++ scope resolution operator (**::**). The following example shows how to access these names.

```
A::B x;               // A::B means name B within A
A::color c = A::red;
```

See Scope access on page 19 for information about **::**. You can make type names as visible as with ANSI Standard C by using **typedef**:

```
typedef A::B B;         // Defines B to be the
                        //  same type as A::B
typedef A::color color; // Same for color
```

Class objects in C versus C++

Each object of a class type has a copy of each of the simple members, distinct from the copies in another object. But this is no different from the C language, and in fact C++ adds nothing to C in this regard. C++ adds to C by increasing the kinds of things you can do with a class object or its members, and by allowing declarations of members that are shared across all objects of a class.

Type-defined members

When you use **typedef** in a member declaration, the effect is the same as a **typedef** declaration elsewhere, except that the declaration is local to the class:

```
class A {
   typedef float F;
   typedef A * (*X)[3];
   };
A::F f;     // f is of type float
A::X d;     // d is of type
            //  pointer-to-array-of-pointers-to-A
```

Static members

When you declare a class member **static**, and the member you are declaring is not a bit field or a function, the declared name denotes exactly one object, shared among all objects of the class. The member declaration does not allocate storage for the **static** object; a definition is required elsewhere. Because the **static** object is a member, you can access it with a class object followed by the standard **.** or **->** operator. But you can also access the **static** object without a class object with the scope-resolution operator (**::**); for example:

```
class A {
public:
   static int count;
   };
int A::count = 0;  // Required definition
main () {
   A a1, a2;
   a1.count++;
   a2.count++;
   A::count++;
   // Now A::count has the value 3
   }
```

The difference between placing `count` within the class and making it global is that within the class, `count` is class specific. Other classes can have the same member declaration, but it denotes a different `count` variable in each case; for example:

```
class A {
   static int count;
   };
class B {
   static int count;
   };
int A::count = 0, B::count = 1;
// There are two count variables here
```

static bit-field members are not allowed. **static** function members are allowed (see Static member functions on page 94). The initializer of a static data member is "in the scope of the class" — that is, the names of all class members are known and can be referred to without the use of the **.**, **->**, or **::** operators. For example:

```
class A {
   enum color {red, white, blue};
   static color C;
   };
A::color A::C = red; // red means the same as A::red
```

Member functions

A class member declaration can be a function declaration or a function definition. Such a member is called a *member function*. Other object-oriented languages refer to such a member as a *method*. If you do not supply a function body in the declaration, you must supply the function body for the member function elsewhere (if you reference the function).

Each member function is shared among all objects of a particular class type. It can be accessed via the **.**, **->**, or **::** operators.

NOTE **inline** is implied for a member function whose definition is supplied within a class definition. You cannot prevent a function from being **inline** when you supply the function body in the class definition. See Inline functions on page 26.

Static member functions

A *static member function* is a function whose declaration (and possibly definition) occur within a class declaration, and the declaration contains the storage category **static**. Here is an example of a static member function:

```
class A {
public:
   static int count;
   static void inc_count();
   };
int A::count = 0;
void A::inc_count() { A::count++; }
main () {
   A a1,a2;
   a1.inc_count();
   a2.inc_count();
```

```
   A::inc_count();
   // Now A::count has the value 3
   }
```

Unlike **static class** data objects, the function definition can be provided within the **class** definition itself; for example:

```
struct A {
   static int count;
   //  Declaration and definition of inc_count:
   static void inc_count() { A::count++; }
   };
int A::count = 0;
```

Scope of member functions

A member function name, its parameter list, and any declarations local to the member function are in the scope of the enclosing class:

```
struct A {
   static void inc_count() { count++; }
                           // ^^^^^ A:: not needed
   static int count;
   };
```

Even so, a **static** member function cannot make a simple reference to a non-**static** data member, because there is no object containing that data member. If `count` were non-**static**, the reference `count++` within `inc_count()` would not be valid.

Non-static member functions

Non-**static** member functions are much more common in C++ than **static** member functions, and are an important element of the object-oriented philosophy of C++. A non-**static** member function differs from a **static** member function in that a non-**static** member function is always supplied an object of the class in which the function was declared.

Calling a non-static member function

To call a non-**static** member function, you must have constructed an object of its class type. When the function is called, the address of that object is passed to the member function as a hidden argument. When that member function refers to the simple members of the class, it is referring to those members within the hidden object. For example:

```
#include <iostream.h>
#include <string.h>
   struct string {
   char A[512];
   void concat(const char *s);
   };
void string::concat(const char *s)
   { _strncat(A, s, sizeof A); }
main () {
   string s;
   s.A[0] = 0;
   s.concat("Hello, ");
   s.concat("world\n");
   cout << s.A;
   }
```

This is a “Hello, world” program. Within **concat()** a reference to `A` needs no **.** or **->** or **::** operator, and it refers to the `A` belonging to the hidden-argument class object. When you call a non-**static** member

function with the form `X.Y()`, `&X` is passed to function `Y` as the hidden argument; similarly, in `X->Y()`, `X` (which is an address) is passed to `Y`.

Hidden argument this

A pointer to the class object is passed in as a hidden first argument, and is named **this**. It is as if the program had been written as follows:

```
#include <iostream.h>
#include <string.h>
struct string {
   char A[512];
   };
void concat(string *this, const char *s) {
   _strncat(this->A, s, sizeof this->A);
   }
main () {
   string s;
   s.A[0] = 0;
   concat(&s, "Hello, ");
   concat(&s, "world.\n");
   cout << s.A;
   }
```

Although not as elegant as the member-function syntax, this is the way C programmers have been writing C programs with some object orientation for years. C++ gives you a syntactic boost.

When you invoke a member function of a particular class, each reference to a member of that class within the function has a pointer dereference for the class object automatically supplied. The automatic pointer dereference is equivalent to **this->***class_member* for any class member. You can also use the **this** argument directly yourself:

```
void string::concat(const char *s) {
   _strncat(this->A, s, sizeof A);
   }
```

With a little renaming, the following example shows a case where you must use **this** directly:

```
class string {
   char A[512];
   string();
   void concat(const char *A);
   };
void string::concat(const char *A) {
   _strncat(this->A, A, sizeof this->A);
   }
```

In this example, **A** was purposely named as an argument to **concat()** to "hide" the class member **A**. Remember that for member functions, the parameter list of the member function is in the scope of the class. In the example, however, the parameter list (**const char** ***A**) essentially overrides the visible member **char** `A[512]`, so **this** is necessary to access **char** `A[512]`. For a member function of class **string**, the type of the **this** parameter is ***string**.

Qualified member functions

Non-**static** member functions can be qualified with **const** or **volatile**; the qualification appears after the right parenthesis in the function declarator.

The purpose of the qualifiers is to modify the type of the **this** hidden parameter. Because there is no place where **this** is explicitly declared, we cannot use the normal C++ syntax to specify its type.

For a member function F of some class S, the type of **this** is S*** const**, that is, a constant pointer to S (you cannot modify **this**). But if the function is qualified with a **const**/**volatile** qualifier list, the type of **this** becomes one of the following:

```
const S * const
volatile S * const
const volatile S * const
```

That is, the **const**/**volatile** qualifier list modifies the type of the object to which **this** points; for example:

```
struct s {                  // Type of this:
   void f();                //  s* const
   void f() const;          //  const s* const
   void f() volatile;       //  volatile s* const
   void f() const volatile;//  const volatile s* const
   int x;
   };
```

A function qualified with **const** essentially promises not to modify the object to which **this** points. For example:

```
void s::f() const { x = 1; }    // Error.
```

Here x is really this->x, and because the type of **this** is **const** s *** const**, you cannot modify the s object.

A function qualified with **volatile** recognizes that the object for which it is being called can be modified by forces unknown to the function, so references to the object's contents are not optimized.

Unless a function is qualified with **const**, it cannot be called with **const** objects; this is a normal consequence of function-call rules that prohibit passing a pointer of type **const** T * where the actual parameter is of type T *. Similarly, unless a function is qualified with **volatile**, it cannot be called with **volatile** objects. To get around this restriction you would have to use a cast, although you should do this only when you know the called function will not modify the object (in the case of **const**) as in the following example:

```
struct s { void f(); };
void g() {
   const s x;
   x.f();          // Error
   ((s&)x).f();    // OK but dangerous: casting x to
                   //  non-const s& before call
   }
```

We recommend you use **const** for any member function that does not modify the object for which it is called. Such a member function can be called on behalf of both **const** and non-**const** objects.

Constructors and destructors are an exception: they cannot be declared **const** or **volatile**, but can be called on behalf of **const** or **volatile** objects.

Derived and base classes

An important part of object-oriented programming with C++ is the ability of a class to use information and attributes from other classes. A class B is a *base class* for another class C if B is referenced in the base list of C or any of C's bases. A class that uses information from another class in its definition is a *derived class*. The information that a derived class uses from a base class is *inherited*.

Each *base_spec* in a *base_list* (see Class declarations and definitions on page 88) specifies a base for a class **X**; each base is simply a previously defined class. `X` is said to be "derived from" the bases, and those bases are "bases of `X`".

- If **virtual** is included in the *base_spec*, the specified base is a *virtual base* of `X`.
- If an *access_specifier* is given in a *base_spec*, the base is a public, private, or protected base of `X`.
- In the absence of an *access_specifier*, if a **class** definition uses the word **class**, each base is assumed to be a private base; if the word **struct** is used, each base is assumed to be a public base; if the word **union** is used, there is no issue, because unions cannot have base classes.

Single and multiple inheritance

The term *single inheritance* means that a derived class has exactly one base class. *Multiple inheritance* means a derived class inherits information from more than one immediate base class. A derived class and all of its base classes and their base classes are a *class hierarchy*. The derived class can also override virtual member functions of its base classes. See Virtual functions on page 100 for more information about virtual functions.

Example 18 Declaration of a base class shows how to declare a base class.

Example 18 Declaration of a base class

```
enum color { red, green, blue, black, white };
class shape {
public:
   int x, y;           // Screen coordinates of center
   color background;   // Screen background color
   color shapeoutline; // Color to make shape outline
   }
class circle : shape {
   int radius;
   }
```

In Example 18 Declaration of a base class, the **class** `shape` is a *private base class* for the **class** `circle`.

Notice that `circle` inherits from `shape` some characteristics that would apply to any shape displayed on a color screen. You could use the class members in `shape` as the basis of another class called `rectangle`. In the example, the `circle` class has *single inheritance*, because only one base class is specified for it. If more than one class name followed **class** `circle` **:**, then `circle` would have *multiple inheritance*.

The members of the base classes is *inherited* by the derived class. This means they can be used after the **.**, **->**, or **::** operators just as if they were members of the derived class. Such use is not permitted if the member is ambiguous — that is, if there is more than one possible member to select.

Example 19 Inheriting members of a base class shows inheritance of members from a base class.

Example 19 Inheriting members of a base class

```
struct A {
   enum color { red, white, blue };
   color C;
   // Print member C in some fashion.
   void print_C() {
      . . .
      }
   };
struct B: A {
   color C2;
```

```
   int blue;
   };
. . .
B b;
if (b.C == b.C2) b.print_C();
B::color C3 = B::red;
```

In Example 19 Inheriting members of a base class, **struct** `B` has the following characteristics:

- It inherits `color`, `red`, `white`, `C`, and `print_C` from `A`.
- It declares an additional member `C2`.
- It overrides the name `blue` in `A`.

Overriding the name `blue` means that a mention of `blue` in the context of class `B` yields `B::blue`, not the inherited name `A::blue`; the inherited name is effectively hidden. Therefore, `b.blue` refers to `B`'s `blue`.

The expressions `b.C`, `B::color`, and `B::red` in Example 19 Inheriting members of a base class refer to the inherited names.

In `struct s2:s1 ...`, `s1` is a *direct* base of `s2`. Add `struct s3:s2...`, and `s1` is an *indirect* base of `s3`.

Access control

Use of inherited members is also governed by access control; see Access control in C++ on page 107.

A derived class can be thought of as a structure containing its own members, as well as storage allocated for direct bases of the class. C++ allows access to the base's fields without explicitly naming the bases. Example 20 Unnamed base classes is written as if the language supplied no base classes.

Example 20 Unnamed base classes

```
struct A {
   enum color { red, white, blue };
   color C;
   // Print member hue in some fashion.
   void print_C() {
      . . .
      }
   };
struct B {
   struct A A_base;   // Fake out a base.
   A::color C2;
   int blue;
   };
. . .
B b;
if (b.A_base.C == b.C2) b.A_base.print_C();
A::color C3 = A::red;
```

The difference between Example 20 Unnamed base classes and Example 19 Inheriting members of a base class is that in some places in Example 20 Unnamed base classes, `A::` replaces `B::`, `A::` is inserted, or `A_base` is accessed to get to the members of the simulated inherited class. This is close to the way inheritance is actually implemented

NOTE When a member function inherited from a base class is called on behalf of a derived class object, the **this** pointer passed is adjusted to point to the base class sub-object within the derived class.

Virtual functions

C++ allows you to declare functions in base classes that act as "place holders" for functions with the same names in derived classes. When you declare a function in a base class that has the **virtual** qualifier, and then declare a function with the same name and parameter types in a class derived from that base, the function in the derived class "overrides" the **virtual** function in the base class. You can use **virtual** for any non-static class member function.

The semantics of a virtual function are as follows:

- If a **virtual** function `f` in a base class `B` is overridden in a derived class `D`, any call to `B::f` on behalf of a `B` object contained within a `D` object actually calls `D::f` on behalf of the `D` object.
- A function `D::f1` overrides a virtual function `B::f2` in the following situations:
  - `f1` and `f2` are the same name and have the same type and number of parameters; for example, `f1 = f2 = func`.
  - `f1` and `f2` are both destructors; for example, `f1 = ~B` and `f2 = ~D`. See Destructors on page 124 for more information.
  - `f1` and `f2` are conversion functions with the same name; for example, `f1 = f2 = operator int`.

Example 21 Virtual functions illustrates a use of virtual functions.

Example 21 Virtual functions

```
#include <stdio.h>
struct point { int X, Y;
   point(int x, int y): X(x), Y(y) {}
   };
struct shape {
   virtual void draw() {
      printf("Nothing interesting here.\n");
      }
   };
static void draw_many(shape *A[], int cnt) {
   for (int i = 0; i < cnt; i++) A[i]->draw();
   };
struct square : shape {
   point upper_left, lower_right;
   void draw() {            // Code to draw a square
      printf("I am drawing a square!\n\t"
         "upper_left=(%d,%d), lower_right=(%d,%d)\n",
         upper_left.X, upper_left.Y,
         lower_right.X, lower_right.Y);
         }
   square(point UL, point LR) : upper_left(UL),
                                lower_right(LR) {}
   };
struct circle : shape {
   point center; double radius;
```

```
   void draw() {            // Code to draw a circle
      printf("I am drawing a circle!\n\t"
            "center=(%d,%d), radius=%f\n",
            center.X, center.Y, radius);
      }
   circle(point C, double R) : center(C), radius(R) {}
   };
struct line : shape {
   point left, right;
   void draw() {            // Code to draw a line
      printf("I am drawing a line!\n\t"
            "left=(%d,%d), right=(%d,%d)\n",
            left.X, left.Y, right.X, right.Y);
      }
   line(point L, point R) : left(L), right(R) {}
   };
main () {
   circle C(point(1,3), 4);
   square S(point(1,2), point(3,4));
   line   L(point(3,3), point(4,5));
   shape *A[] = {&C, &S, &L};
   draw_many(A,3);
   }
```

Constructors

To make Example 21 Virtual functions an actual runnable program, we had to provide constructors. A *constructor* is a member function whose name is the same as that of the class. In Example 21 Virtual functions, `point(1, 3)` constructs a `point` object whose `X` and `Y` values are `1` (one) and `3` respectively; `circle C(point(1, 3), 4)` constructs a `circle` whose center and radius values are `point(1, 3)` and `4`, respectively; and so forth. See Constructors on page 117 for more information about constructors.

Example 21 Virtual functions shows a basic `shape` class with a **virtual draw()** function that does not do anything realistic; it is intended to be overridden by a **draw()** function in a derived class. Nonetheless, we can still define a **draw_many()** function that draws many shapes, calling `shape::draw()` on each one. The three derived classes define three realistic shapes: a square, circle, and line; each class provides an overriding draw function that can draw that specific shape.

When **draw_many()** is passed three `shape` pointers (`&C`, `&S`, `&L` get converted to a pointer to the base class `shape`), the call to **draw()** within **draw_many()** invokes the functions in `C`, `S`, and `L` instead, and the **this** pointer for those three calls is adjusted back up to the derived class from the base class. This makes it possible to write **draw_many()** without having to know all possible classes that might be derived from `shape`. Operationally here is what happens:

`A[0]` is assigned the pointer to the `shape` base within `C`.

`A[1]` is assigned the pointer to the `shape` base within `S`.

`A[2]` is assigned the pointer to the `shape` base within `L`.

Within **draw_many()**:

`A[0]->draw()` adjusts `A[0]` to pointer to `C`, then calls `circle::draw()` on behalf of `C` (passing pointer to `C` as **this**).

`A[1]->draw()` adjusts `A[1]` to pointer to `S`, then calls `square::draw()` on behalf of `S` (passing pointer to `S` as **this**).

`A[2]->draw()` adjusts `A[2]` to pointer to `L`, then calls `line::draw()` on behalf of `L` (passing pointer to `L` as **this**).

If you run this program, you get the output:

```
I am drawing a circle!
   center=(1,3), radius=4.000000
I am drawing a square!
   upper_left=(1,2), lower_right=(3,4)
I am drawing a line!
   left=(3,3), right=(4,5)
```

How does `A[i]->draw()` "know" the pointer adjustment and the right function to call? The answer is that each object with any virtual member functions contains a pointer to a virtual-function table (see Virtual functions on page 100). That table is referenced whenever a call to a virtual function **f()** is made. The entry in that table for **f()** records the pointer adjustment and which function to call. The table has as many entries as there are virtual functions in a class; each function is assigned a distinct, compile-time-known location in that table. The pointer to that table is installed in the object when the object is created at run time (see Constructors on page 117).

In Example 21 Virtual functions, you can think of `A[0]->draw()` as implemented by:

```
vp = &A[0]->vftab[draw_index];
vp->func(A[0]+vp->adjust);
```

where `vp` is a pointer to a virtual-function-table entry, and `draw_index` is the compile-time-known location for the virtual draw function within the table. `vp->adjust` is the value that converts the pointer to the base shape within a `C` to the pointer to the `C` object. The declaration of the `C`, `S`, or `L` object installs the appropriate virtual-function table within the `shape` sub-object of `C`, `S`, or `L`.

Virtual calls and polymorphism

The use of the virtual-function table in a call to a virtual function is often referred to as the *virtual call mechanism*. The virtual call mechanism provides C++ with *polymorphism*: a function call can invoke one of a set of different functions, depending on the type of the object on behalf of which the function is called. For example, if the type is a `circle`, the circle's **draw()** function is called; if the type is a `square`, the square's **draw()** function is called; and so on.

Virtual-function tables

The *set of virtual-function tables for a class* `X` is the collection of tables designed to override any virtual functions in any base class(es) of `X`, and to remap virtual functions in `X` to themselves. These tables are plugged into the virtual-function-table pointers for `X` and all its bases. There can be a different virtual-function table for `X` and for each of its bases (although there are compiler optimizations that allow many of these tables to be "shared"). This set of virtual-function tables allows base-class functions to be overridden by functions that are as "high up" as `X` in the class hierarchy. The virtual-function table for `X` itself remaps each virtual function in `X` to itself.

When an object of class `C` is created, all these virtual-function tables are installed within `C` and its bases as appropriate. See Constructors on page 117 for a more precise specification of the timing of this installation.

The virtual call mechanism is *skipped* when the **::** operator is used in conjunction with the **.** or **->** operator to access the function:

```
A[i]->shape::draw();
```

Here the `shape::` prefix means "really call the **draw** function in `shape`". This would produce:

```
Nothing interesting here.
```

for each drawing in Example 21 Virtual functions.

"Sticky" virtual

If member function **f** in a derived class `X` overrides a virtual function in a base, the **virtual** specifier is implied for `X::f`. That is, **virtual** is "sticky"; once said, you cannot get rid of it among overriding functions.

Pure declarations

A virtual function in a class used as a base must be defined somewhere, or must be declared *pure*. A pure declaration can be given for a virtual member function, and consists of supplying "`= 0`" in the *member_declarator*:

```
struct shape {
   virtual void draw() = 0;
   };
```

This is a useful alternative to the way **draw()** was written in Example 21 Virtual functions, because it eliminates the function body that will never get called anyway:

```
virtual void draw() {
   printf("Nothing interesting here.\n"); }
```

The result is that if **draw()** is called for a `shape` object that is not part of a derived object, the value `0` (zero) is called — resulting in a memory fault on most machines. But if you design your program so that never happens, you can usefully omit the dummy function body by using the pure declaration.

Abstract classes

A class is called *abstract* if it has at least one pure **virtual** function. No objects of an abstract class can be created except as a base of a non-abstract derived class. You cannot declare an object of an abstract class type. An important use of an abstract class is to provide an interface without providing any implementation; an implementation can be provided in a derived class.

Virtual base classes

A base class is *virtual* if the **virtual** key word is supplied when the base class is mentioned in the definition of the derived class. Such a base class is known as a *virtual base*. Do not confuse **virtual** functions with **virtual** bases. The meanings of the keyword are unrelated in the two contexts.

If a base class `B` occurs more than once as a direct or indirect virtual base class of `D`, all such occurrences refer to exactly one copy of the base class. Example 22 Multiple virtual base classes shows this.

Example 22 Multiple virtual base classes

```
struct s1 { };
struct s2 { int f; };
struct s3:s1, virtual s2 { };
struct s4:s1, virtual s2, s3 { };
struct s5:s4, s2 { };
```

In Example 22 Multiple virtual base classes, `s2` is mentioned as a virtual base class of `s3` and of `s4`, and as a non-virtual base class of `s5`. In class `s4` there is exactly one base `s2`, shared between `s4` and `s4`'s base class `s3`. In class `s5` there are two occurrences of `s2`: one as part of `s4` and one as part of `s5`. The declaration of `s5` is unusual in that there is no way to refer to `s2` members of `s5`, because all such references are ambiguous. However, it is possible to refer to the `s2` members of `s4` with the `s4::` scope resolution operator. For example:

```
s5 x;
x.f = 1;            // Ambiguous:  Which s2::x?
x.s4::f = 1;        // OK, the single s2 in s4
```

Virtual base classes are interesting only in the context of multiple inheritance. In the above examples they allow local sharing of information among the `s3` and `s4` classes.

Virtual base pointers

Virtual base classes complicate conversion of a derived class pointer to a base class pointer, and thus complicate calling a member function inherited from a virtual base class (which requires **this** to refer to the base class). In terms of the implementation, any class with a direct virtual base class contains a pointer (called a *virtual base pointer*) that points to that virtual base class. The conversion from the derived class pointer to the base class pointer essentially just accesses that virtual base pointer.

When an object is created, enough room is allocated for just one copy of a virtual base class. The virtual base pointer for that object and for any base classes containing the virtual base class are initialized at the creation (see Constructors on page 117). Pursuing the previous examples, consider the declaration of an `s4` object:

```
s4 x;
```

The space allocated accommodates one `s2` object. `s4`'s and `s3`'s virtual base pointers for `s2` are both initialized to point to that `s2` object. Therefore, any reference to `this->f` (`f` is in `s2`) in a member function of either `s4` or `s3` refers to the same `int f`.

Pointers to members

You can declare pointers to members of a class. A pointer-to-member declaration uses the following syntax from *ptr_operator* in its *declarator*:

```
{identifier ::}... *
```

The class containing the members to which the pointer-to-member points is identified by this syntax, which is basically a *type_specifier* referring to a previously declared class. For example:

```
struct s {
   int i, j, k;
   float f;
   };
int s::* p;
```

In this example `p` is "pointer-to-member-of-**s**-of-type-**int**". The **s::** shows that the pointer points to members of **s**, and the leading **int** tells you that those members are of type **int**.

When to use pointers to members

We already have pointers in C, so what good is it to tie down pointers to point to specific class members? Does that make any sense?

Consider the following problem. Write a function to take an array of **s** objects and initialize the **i** member of each array element to 0 (zero):

```
void init_0(s A[], int elements) {
   for (int n = 0; n < elements; n++)
      A[n].i = 0;
   }
```

This is straightforward. But suppose you want to choose whether to initialize the `i`, `j`, or `k` member. You could pass **init_0** a "choice" parameter:

```
void init_0(s A[], int elements, int which ) {
   for (int n = 0; n < elements; n++)
      switch(which) {
         case 0: A[n].i = 0; break;
         case 1: A[n].j = 0; break;
         case 2: A[n].k = 0; break;
         }
   }
```

But every time you add a new member to **s**, you have to change **init_0**. An alternative is use a pointer-to-member. A pointer-to-member is essentially a way to represent the name of a class member.

```
void init_0(s A[], int elements, int s::*p) {
   for (int n = 0; n < elements; n++)
      A[n].*p = 0;
   }
s A[10];
main () {
   init_0(A, 10, &s::i);  // Initialize the i field
   init_0(A, 10, &s::k);  // Initialize the k field
   }
```

Pointers to data members

With respect to implementation, a pointer-to-(data-)member is essentially just the offset from the beginning of any class object to that member; it is really just a way to represent the ANSI Standard C **offsetof** macro in a strongly typed manner. When `f` uses the **.*** operator on `A[n]` and `p`, it takes the base address of `A[n]`, adds the offset contained within `p`, and arrives at the address of the particular member of `s`; that member is then assigned the value `0` (zero).

Because of the strong typing, you cannot pass `init_0 &s::f`, because `&s::f` is of type pointer-to-member-of-**s**-of-type-**float**, and **init_0** takes pointer-to-member-of-**s**-of-type-**int**.

Pointers to member functions

You can also have a pointer-to-member of a function type This allows you to take the address of a non-static class member function; this is different from the standard C pointer-to-function, because the latter cannot point to non-static class member functions.

For example:

```
struct s {
   int fnc1(), fnc2();
   };
int (s::* p)() = &s::fnc1;
```

Here `p` is a pointer-to-member-of-`s`-of-type-function-returning **int**. We initialized it to `&s::fnc1`. We can later supply an object of type **s** and call the pointed-to function:

```
main () {
   s x;
```

```
   (x.*p)();   // Same as x.fnc1()
   p = &s::fnc2;
   (x.*p)();   // Same as x.fnc2()
   }
```

Pointers to static member functions

Pointer-to-member functions are not appropriate for taking the address of static member functions. Static member functions do not need the hidden argument of the class type that is passed to non-static member functions. Thus straightforward C pointers-to-functions can point to static member functions:

```
struct s {
   int fnc1(), fnc2();
   static int fnc3();
   };
int (* p)() = &s::fnc1;  // Not valid!
int (* q)() = &s::fnc3;  // OK
```

Limitations of pointer-to-member comparison

When a pointed-to member function is a virtual function, a comparison between two pointer-to-member functions might not yield the results you expect.

If one source file sets the value and a different source file does the comparison, the comparison fails because different function thunks are used to implement the call to the virtual function.

File `hdr.h`:

```
struct s {
   virtual fnc();
   };
```

File `1.cc`:

```
#include "hdr.h"
extern void (s::*p)();
extern sub();
main () {
   sub();              // Set the pointer
   if (p==s::fnc)
      printf("OK\n"); // Will not print OK
   }
```

File `2.cc`:

```
#include "hdr.h"
void (s::*p)();
sub() {
   p = s::fnc;
   }
```

Chapter 8 — Access control in C++

In This Chapter

Introduction

Access control determines who can access a member declared in a class or inherited from another class. For example, a class can implement a data member and allow only member functions that manipulate that data member to be visible; the data member itself can be hidden by denying anyone other than the class member functions access to that data. Access control allows you to specify access to individual members of a class, and can discriminate which among all other classes can access those members.

Access

Each member of a class has one of the following accesses, in diminishing order of "usability":

public	A public name is accessible in any context.
protected	A protected name owned by a class X can be used by a member function defined for X or any class immediately derived from X, and by any friend of X (see Friends on page 113).
private	A private name owned by class X is accessible by a member function defined for X and by any friend of X.
inaccessible	An inaccessible name is not accessible in any context.

Example 23 Accessing a private member

```
struct B {
public:
   int i_pub;     // i_pub is public.
   void bfunc();  // bfunc is public, too.
protected:
   int i_prot;    // i_prot is protected.
private:
   int i_priv;    // i_priv is private.
   };
void B::bfunc() {
   i_priv++;      // OK:  Member function can access
                  //  private members.
   i_prot++;      // OK:  Member function can access
                  //  protected members.
   }
void anyfunc() {
   B b;
   b.i_priv++;    // Error; cannot access
                  //  private member
   b.i_prot++;    // Error; cannot access
                  //  protected member
   b.i_pub++;     // OK:  Anyone can access
                  //  a public member.
   b.bfunc();     // OK, for same reason;
                  //  increments i_priv
   }
```

In Example 23 Accessing a private member, **bfunc()** is provided as the only way an "outsider" can increment the private member **i_priv**. In this manner the implementor of class `B` can completely control both the reading and the writing of its data members.

Using access specifiers

The access of a member is specified through a combination of the following:

- an *access_specifier* within a *class_definition*; that, is one of the following:

  ```
  private:
  public:
  protected:
  ```

- the access specifier for a base class: you can specify **public**, **private**, or **protected** for each base class in the list of base classes for a derived class (a base is a **public**, **private**, or **protected** base of another class)
- an access override (see Access override on page 111)

The *access_specifier* specifies the access control for declarations physically following it in the class declaration (unless the declaration is instead interpreted as an access override), up until the next *access_specifier*. For **struct** or **union**, **public:** is implied at the beginning of the **struct** or **union** declaration. For **class**, **private:** is implied.

Determining access of inherited members

When a class derives from a base class, the members of the base class are inherited as members of the derived class. The access of those members in the derived class is determined from the base access specifier and the access of the member in the base class. You can view the base access specifier as a filter that transforms the access of a name in the base class to a new access as the name becomes a member of the derived class. In the absence of a base access specifier, **public** is implied if the derived class is a **struct** or **union**, and **private** is implied otherwise. Here is how to determine the access of inherited members in a class `D` derived from a base `B`:

- If `B` is a **public** base of `D`, any **public** or **protected** member of `B` is a **public** or **protected** member, respectively, of `D`; that is, the access is not changed — and any **private** or inaccessible member of `B` becomes an inaccessible member of `D`.
- If `B` is a **protected** base of `D`, any **public** member of `B` is "demoted" to a **protected** member of **D**, the access of a **protected** name is unchanged, and any **private** or inaccessible member of `B` becomes an inaccessible member of `D`.
- If `B` is a **private** base of `D`, any **public** or **protected** member of `B` is "demoted" to a **private** member of `D`, and any **private** or inaccessible member of `B` becomes an inaccessible member of `D`.

To summarize, any derivation makes **private** and inaccessible members inaccessible, and demotes any other access to a level no more "accessible" than the level of the derivation. See Table 6 Access of members of base and derived classes.

Table 6 Access of members of base and derived classes

Access in base class	Access in derived class when base access is:		
	public	**protected**	**private**
public	**public**	**protected**	**private**
protected	**protected**	**protected**	**private**

Table 6 Access of members of base and derived classes (con't)

Access in base class	Access in derived class when base access is:		
	public	**protected**	**private**
private or inaccessible	inaccessible	inaccessible	inaccessible

To determine whether an attempt to use a class member is valid, this is all you need to know:

- the class to which the member belongs
- the access of the member (**public**, **protected**, **private**, or inaccessible)
- the accessor: who is accessing the member

To which class does the member belong?

In any use of a class member, you must take care to identify the class to which the member belongs. Because of C++'s inheritance, the same name can be used for a member of the class in which it is defined, and also for a member of some derived class that inherited it. When you write:

`T::x`

x is a member of `T`, even though it might have been inherited from some base class. For:

`b.x`

x is a member of `b`'s class. For:

`p->x`

the class of `*p` is the owner. Finally, for just:

`x`

there is usually an implied **this->**, as in a member function. **x** then is a member of the type of ***this**.

In any of these cases, **x** might be defined originally in some base class, but it is being referenced as a member of a derived class.

Consider class `D` with `B` as a private base class:

```
struct D: private B {
   void dfunc();
   };
// Recall that as members of D,
//  i_priv is inaccessible,
//  i_prot is private, and
//  i_pub and bfunc are private.
void D::dfunc() {
   i_priv++;      // Error: this->i_priv is
                  //  an inaccessible member of D.
   i_prot++;      // OK:  this->i_prot is
                  //  a private member of D.
   i_pub++;          // OK:  this->i_pub is
                  //  a private member of D.
   B b;
   b.i_priv++;    // Error: cannot access
                  //  private member of B
   b.i_prot++;    // Error: cannot access protected
                  //  member of private base class B
   b.i_pub++;     // OK
   int B::*p = &B::i_prot; // OK: can access
```

```
                          //  protected member of
                          //  base
   }
```

Here `D::dfunc` is accessing **i_prot** once as a member of `D` and twice as a member of **B**. For the unadorned reference to `i_prot, this->` is implied, and so **i_prot** is considered a member of `D` (the type of ***this**). Because **i_prot** is a private member of `D`, and **dfunc()** is a member function of `D`, the access is allowed.

For the references `b.i_prot` and `B::i_prot`**, i_prot** is a member of `B`. The access is allowed for a different reason: the protected members of a base are accessible by an immediately derived class (`D`).

Name overloading and access

In C++ a single function name can have different parameter lists. This is called *overloading* (see Function overloading on page 135 for more information). Because names can be overloaded, access is determined for a particular member of a class, not just by the member's name.

```
struct G {
public:
   int f(int);
protected:
   int f(double);
private:
   int f(char *);
   };
G g;
```

In the preceding example:

- **g.f(1)** is an access to a **public** name owned by **G**.
- **g.f(1.0)** is an access to a **protected** name owned by **G**.
- **g.f("1")** is an access to a **private** name owned by **G**.

Access override

Access override allows you to preserve the access of a **public** or **protected** name inherited from a base class, despite the access specifier attached to the base-class name in the class definition. Any member declaration with no *decl_specifier*s and for which the declaration is of the form *identifier***::***simple_name* is an *access override* and does not actually declare any member of the class. The *identifier***::** in the access override must denote one of the base classes.

Again, taking class `B` in Example 23 Accessing a private member and a sample derivation `D` from `B`, we have:

```
struct D : protected B { };
```

The names in `D` have the following access rights:

```
i_priv: inaccessible
i_prot: protected
i_pub , bfunc: protected
```

You might want the public members of `B` to remain public. You can do so by using access override:

```
struct D : protected B {
   public:  B::i_pub;  B::bfunc;
   }
```

The names in **D** now have the following access rights:

```
i_priv: inaccessible
i_prot: protected
i_pub , bfunc: public
```

Deriving `E` from `D`, we keep the protected member of `D` protected despite a **private** derivation:

```
struct E : private D {
   protected: D::i_prot;
   };
```

The names in `E` have the following access rights:

```
i_priv: inaccessible
i_prot: protected
i_pub , bfunc: private
```

Otherwise, **i_prot** would have been **private**.

Access override for overloaded functions

An access override for a name adjusts the access to all declarations of that name in the base class. For an overloaded function, the access is adjusted for all functions of that name in the base class, provided they have the same access in the base class:

```
struct H {
   public:    int f(int), f(double);
   protected: int g(int);
   public:    int g(double);
   };
struct DH: private H {
   public:     H::f;  // OK, preserve public access
   protected:  H::g;  // Not valid!  Different
                      //  accesses for g
   };
```

Note that you can only preserve the same access, not change it. Here is an example of an invalid attempt:

```
struct DH: private H {
   protected:  H::f  // Not valid; cannot change
                     //  public to protected
   };
```

Multiple access through virtual base classes

Due to sharing between virtual base classes, a name can be accessible through more than one path. The access granted is the one that allows the greatest accessibility.

```
struct V: { int x; };
struct D1: virtual public  V { };
struct D2: virtual private V { };
struct E: D1, D2 { }
```

Here, **`E::x`** has public access in **`E`** because there is a path through **`D1`** to obtain **`V`** publicly. The path through **`D2`** is ignored.

Friends

A *friend* is a function that can access members of a class that are not **public**, even though the function is not a member of the class.

Example 24 Friends of class attack_pieces

```
class position {
public:
   int where[2];
   };
class attack;
class attack_pieces {
   friend attack;// Declares attack to be a friend
   int queens_rook[2];
   int queens_knight[2];
   int queens_bishop[2];
   int queen[2];
   int kings_bishop[2];
   int kings_knight[2];
   int kings_rook[2];
   int in_check(int *);
public:
   attack_pieces();
   analyze_knights(int,int);
   };
class attack {
public:
   void move_knight(int *, int *);
   void move_bishop(int *, int *);
   void move_rook  (int *, int *);
   void move_queen (int *, int *);
   };
class move_piece : attack {
   int legal_moves[100];
   int check_legal( int where[]);
   }
```

In Example 24 Friends of class attack_pieces, every function in **class** `attack` is a friend of **class** `attack_pieces`. The functions of `attack` can access and change all the private members of `attack_pieces`. The declaration **friend** `attack` declares the **class** `attack` as a friend of `attack_pieces`. The meaning of a class as a friend is that all the class functions are friends of the class declaring the `friend` class. However, the name **move_knight** is not in the scope of `attack_pieces`, nor are any of the other member functions of `attack`; therefore, you cannot reference them as though they were member functions of `attack_pieces`.

```
main(){
   position whereami;
   attack_pieces mine, yours;
   attack mymove, yourmove;
   // The following is not valid because mine does not
   //  have access to attack member functions:
   mine.move_knight((int *)whereami.where[0],
                    (int *)whereami.where[1]);
   }
```

The function **check_legal()** is not a friend of `attack_pieces`, even though it is derived from **class** `attack`.

The property of being a friend is not necessarily reflexive; in the example, `attack` is a friend of `attack_pieces`, but `attack_pieces` is not a friend of `attack`. Nor is friendship transitive; if `A` is a friend of `B`, and `B` is a friend of `C`, it does not follow that `A` is a friend of `C`.

You can make an individual member function of `attack` a friend of `attack_pieces`, rather than all functions in `attack`, as illustrated in Example 25 Only move_knight() is a friend of class attack_pieces.

Example 25 Only move_knight() is a friend of class attack_pieces

```
class position {
   int where[2];
   };
class attack {
public:
   void move_knight(int *, int *); // This is the
                                   // only friend.
   void move_bishop(int *, int *);
   void move_rook  (int *, int *);
   void move_queen (int *, int *);
   };
class attack_pieces {
   friend void attack::move_knight(int *, int *);
   int queens_rook[2];
   int queens_knight[2];
   int queens_bishop[2];
   int queen[2];
   int kings_bishop[2];
   int kings_knight[2];
   int kings_rook[2];
   int in_check(int *);
public:
   attack_pieces();
   analyze_knights(int,int);
   };
```

In Example 25 Only move_knight() is a friend of class attack_pieces, only the function **move_knight()** is a friend of `attack_pieces`. The other member functions of `attack` have no access to **private** or **protected** members of `attack_pieces`. If a **friend** declaration references an overloaded name or operator, only the function that is referenced by the argument types becomes a **friend**.

Any **friend** function whose body is defined in a **class** declaration is automatically **inline**.

Chapter 9 — Special C++ members

In This Chapter

Introduction

You create objects with classes, and classes have special member functions designed to protect the integrity of such types. For example, class constructors (see Constructors on page 117) are special member functions that operate in conjunction with other C++ language rules to ensure that a class object is not used until it is properly initialized. In C you can specify a stack object with the functions:

```
// Implement stacks with integer "handles":
typedef int stack;
stack create_stack();
void push(stack S, int value);
int pop(stack S);
```

You can then use this stack object as in the following function:

```
void f() {
   stack S;
   S = create_stack();
   push(S, 1);
   if (pop(S) != 1) error("What?!");
   }
```

However, if you forget the call to **create_stack()**, *S* is uninitialized, which causes calls to **push()** and **pop()** to fail. But if the stack is an instance of a **stack** class, you can define a constructor member function for stacks. Such a constructor is always called when a stack object is declared, which ensures that a stack object is always initialized before it is used.

Table 7 C++ special members lists the kinds of special members in C++ and indicates their purpose.

Table 7 C++ special members

Special member	Purpose
Constructor	Properly initialize an object
Destructor	Properly "destroy" an object when the object ceases to exist; destruction can include, for example, memory deallocation
Copy constructor	Control what happens when a copy of an object is made; a copy constructor is a special kind of constructor
Operator `=()`	Control what happens when something is assigned into an object
Conversion function	Define automatic conversions of a class object to another type, the same way C++ automatically converts from (for example) **double** to **int** when necessary
Operator **new**	Control heap allocation of an object
Operator **delete**	Control heap deallocation of an object

All special members are invoked automatically without specifically referring to them. For example:

```
void f() {
   stack S1, S2;  // Invokes constructor twice
   S1 = S2;       // Invokes stack::operator =
   stack S3 = S1; // Invokes copy constructor
   stack *p = new stack;
                  // Invokes stack::operator new
                  //  and the constructor
   delete p;      // Invokes stack::operator
                  //  delete
   int i = S1;    // Invokes stack::operator int
   }
```

The member is invoked only if it is declared for the stack class. Notice that in the example, no specific class member is named; each member is invoked automatically by the compiler. In that sense these members are “special”.

Special member functions obey all the usual access rules. For example, if a stack constructor is private, stacks cannot be declared within functions that are not members of the stack class. If you try to declare a stack in a function that is not a member of the stack class, you get an access error when the private constructor is called.

Constructors

A *constructor* is a member function that creates class objects. A constructor always has the same name as its class. A class can have any number of constructors, or even none at all. A constructor’s declaration cannot specify a return type.

If a class type has a constructor, creation of an object of that class type invokes a suitable constructor. The term *suitable constructor* means that one constructor from among those declared will be called. Overload resolution (see Overload resolution on page 136) is used to choose such a constructor. The arguments to the contructors are taken from the context in which the object is being created.

Creation of a class object by a constructor can occur in the following ways:

- declaration of a class object as a local or global variable
- explicit use of operator **new**
- explicit call of a constructor
- creation of a temporary class object by the compiler
- as a member of a class that is being created
- as a base of a class that is being created

Creating a class with a constructor ensures that an object is initialized before any use is made of the object. Essentially, the constructor takes “raw memory” and turns it into a valid class object. The special behavior of constructors is different from anything you can easily mimic. The rules are carefully designed to ensure that no member function is called for a class object until that class object is fully initialized.

Characteristics of constructors

A constructor must be a non-**static** member. The **this** parameter passed to the constructor is the pointer to the area of storage that the constructor turns into a valid object. A constructor for a given class differs from all other non-**static** member functions of the class, because the **this** parameter passed to a constructor points to storage that has not been turned into a valid object, whereas for all other member functions, **this** points to a valid class object.

A constructor cannot be declared **const** or **volatile**. A constructor cannot be **virtual**. You cannot take the address of a constructor.

Constructor arguments

You can provide arguments to the constructor in the following contexts:

- in a declaration of an object with a parenthesized expression list
- in a **new** expression with optional parenthesized expression list

- in a constructor initializer (see Constructor initializers on page 121) with a parenthesized expression list

An explicit invocation of a constructor includes zero or more argument expressions, just like the invocation of any member function.

No other contexts exist for which arguments to a constructor are present.

The default constructor

For any cases of object creation that demand a constructor where it is not possible to specify any arguments, zero arguments are implied. This means that overload resolution will be forced to find a constructor that is callable with no arguments. Such a constructor is called a *default constructor*. For example:

```
struct stack {
   stack();         // Default constructor, takes no arguments
   stack(int);      // Not a default constructor
   stack(int,int);  // Also not a default constructor
   };
void somefunc(stack S);
void f1() {
   stack S1;        // Calls default constructor stack()
   stack S2(3);     // Calls stack(int)
   stack S3(3,3);   // Calls stack(int,int)
   stack S4("xyz"); // Error: Overload resolution; cannot find a suitable
                    //  constructor.
   stack S5(4.5);   // Calls stack(int) after converting 4.5 to an int
                    //  via standard conversions
   stack *p1 = new stack;      // Calls stack()
   stack *p2 = new stack(10);  // Calls stack(int)
   stack *p3 = new stack(1,2); // Calls stack(int,int)
   somefunc(stack());  // Explicit call to stack()
   somefunc(stack(1)); // Explicit call to stack(int)
   somefunc(stack(1,2));      // Explicit call to stack(int,int)
   }
```

If a class has no default constructor, but one is needed (because work such as initializing virtual-function tables is necessary for constructing an object), the compiler generates such a default constructor, with **public** access.

What constructors do

When a class contains members or bases that are classes, the construction of an object of that class type requires that the members or bases are constructed as well. In addition to your code for a constructor, the MetaWare C/C++ compiler inserts other code that causes the class type's members and bases to be appropriately constructed. All such inserted code occurs ahead of your source code in a constructor.

Setting aside the issue of virtual bases for the time being, this is what a constructor for a class does:

1. It calls a suitable constructor for each direct base of the class (in left-to-right order in a case of multiple inheritance).
2. It calls a suitable constructor for each member of the class.
3. It installs virtual-function tables for this class and its bases (see Virtual-function tables on page 102 for information about virtual-function tables).
4. It executes your constructor code.

The reason the bases and members of the class are constructed before executing your constructor code is that your code might refer to a member or a base's member. These members must be properly initialized for your code to work correctly. C++ thus guarantees that the member or base objects are properly initialized before any attempt to access them in your constructor code.

Constructing a direct base

Construction of a direct base invokes a constructor defined in the base's class. In turn, the constructor in the base class, following the rules in What constructors do on page 118, constructs its own bases and members. In this manner, all members, bases, and members of bases are constructed before your constructor code is executed.

Constructors and virtual functions

If a constructor for class X calls a **virtual** member function of X, the call is always to the actual function in X, not to a function in a class derived from X. It would be improper for a constructor to call an overriding member function in a class derived from X. This is because, due to the order of construction, that derived class is not fully constructed yet, and so the behavior of any overriding function might be unpredictable.

Example 26 A virtual function used in constructor code shows how a **virtual** function is used in constructor code.

Example 26 A virtual function used in constructor code

```
struct s1 {
   s1() {
      f1();           // Calls s1::f1 always
      }
   virtual void f1() { x = 1; }
   void init_s1() {
      f1();           // Can call overriding function
      }
   int x;
   };
struct s2 {
   s2() {
      f2();           // Calls s2::f2 always
      }
   void f2() { x = 2; }
   void init_s2() {
      f2();
      }
   int x;
   };
struct s3: s1 {
   s3() {
      f3();           // Calls s3::f3 always
      f1();           // Calls s3::f1, not s1::f1
      }
   void f1() { s1::x = 30; }  // Overrides s1::f1
   void f3() { x = 3; }
   void init_s3() {
      init_s1();      // Calls init_s1 for base s1
      y.init_s2();    // Calls init_s2 for member y
      f3();
      f1();
      }
   int x;
```

```
   s2 y;
   };
void main() {
   s3 obj;
   obj.init_s3();
   }
```

When `obj` is declared to be **class** type **`s3`**, in Example 26 A virtual function used in constructor code, the generated code does the following:

1. It calls constructor `s1::s1()` for base **`s1:`**, which installs virtual tables for **`s1`** that remap **`s1`**'s functions to themselves.
2. It calls **`f1()`**; this means call `s1::f1()`.
3. It calls constructor `s2::s2()` for member **`y`**: which installs virtual tables for **`s2`** that remap **`s2`**'s functions to themselves.
4. It calls **`f2()`**; this means call `s2::f2()`.
5. It installs virtual tables for **`s3`** that remap **`s3`**'s functions to themselves, and remap `s1::f1` to overriding `s3::f1`.
6. It calls **`f3()`**; this means call `s3::f3()`.
7. It calls **`f1()`**; this means call `s3::f3()`, not `s1::f1`, because the latter has been overridden.

So, the non-constructor functions are called in this order:

```
s1::f1()
s2::f2()
s3::f3()
s3::f1()
```

Installing the virtual-function tables in step 5 reinstalls virtual-function tables for base **`s1`**. In general, base-class virtual-function tables are overwritten during construction as many times as the length of the derivation chain from the base to the topmost object being declared.

By contrast, suppose you write your own mechanism for initialization by defining the **`init_`*`xx`*`()`** functions in the classes in Example 26 A virtual function used in constructor code. The call in **`main()`** to `obj.init_s3()` behaves differently from the construction of `obj`, even though you try to write the **`init_`*`xx`*`()`** functions to mimic the constructors. The reason is the timing of virtual-function table installation. By the time your program calls **`init_s3()`**, the virtual-function tables installed are those for **`s3`**, so that **`s1`**'s functions have been overridden. This causes the generated code to do the following when calling **`obj.init_s3()`**:

1. It calls **`init_s1()`** for base **`s1`**, which calls **`s3::f1()`**, which overrides `s1::f1`.
2. It calls **`init_s2()`** for member **`y`**, which calls `s2::f2()`.
3. It calls **`f3()`**. This action calls `s3::f3()`.
4. It calls **`f1()`**. This action calls `s3::f1()`.

So, the non-constructor functions are called in this order:

```
s3::f1()
s2::f2()
s3::f3()
s3::f1()
```

The first function called is an overriding one, whereas in the case for constructors, the first function called is **`s1::f1()`**.

Constructor initializers

A constructor for a class automatically constructs its direct bases and its members. Overload resolution always chooses default constructors, because no arguments are given to the base and member constructors. However, a *constructor initializer* is a mechanism you use to specifically supply arguments to the constructors, so that overload resolution can choose something other than a default constructor.

After a constructor's declarator, and preceding its definition (`{...}`), you can supply a colon and a list of one or more initializers separated by commas:

Each initializer takes the form of an identifier followed by a parenthesized expression list. The identifier must refer to a member, a direct base, or a virtual base. The expressions in the list are taken as the arguments for the constructor to be used for the base or member referenced by the identifier. If the member referenced is not a class object, the list must have exactly one expression, and this expression is taken as the initial value for the member.

> **NOTE** If a member and a base have the same name, the member is preferred in the constructor initializer, and there is no way to specify the base. Giving a base and a member the same name is not a recommended practice.

The constructor initializer does not change the order of initialization of bases and members for constructors. It just allows other than the default constructor to be chosen for the initialization of a base or member.

Consider adding the following two constructors to Example 26 A virtual function used in constructor code:

```
s1::s1(int i) { x = i; }
s2::s2(char *s) {
   printf("whaddaya know:%s\n",s);
   }
```

This addition allows you to modify the definition of the constructor in **`s3`** to call each of these two non-default constructors rather than the default constructors called in the following example:

```
struct s3: s1 {
   s3() : s1(5), y("called from s3") {
      ...
      }
```

So, when an **`s3`** object is constructed, `s1::s1(int)` is called for the base **`s1`**, and `s2::s2(char *)` is called for the `y` member — rather than calling `s1::s1()` and `s2::s2()`, as is done without the constructor initializer.

A constructor initializer is also useful for initializing non-**static const** members and reference members of a class, as in the following example:

```
int i;
struct s {
   s() : x(i), c(3), k(7.2) { }
   int &x;
   const int c;
```

```
   float k;
   };
```

This initialization of field **k** could have been moved into the body of **s()** with an assignment statement `(k = 7.2)`, but this is not possible for the **const** or reference member.

Virtual base classes and constructors

Virtual bases are a special case for constructors. There is only one copy of any virtual base, so that base is initialized by the "topmost" constructor being called. Constructors called by the topmost constructor for initializing bases are "instructed" that they must not construct any virtual base; this assures that only the topmost constructor constructs the virtual base. After all, the topmost object being constructed is the one that must allocate the storage for the virtual base anyway.

Constructors that are not topmost are "instructed" not to initialize a virtual base by being passed a hidden parameter that tells them whether or not they are a topmost constructor. Given this action, the procedure for a constructor for a class **X** can be rewritten (in pseudo-code) as follows:

```
constructor_code(boolean I_am_topmost) {
   0. if (I_am_topmost) {
      Initialize all virtual base pointers for all bases.
      For each base in a depth-first, left-to-
      right traversal of the class graph, construct
      each virtual base and pass FALSE as the value
      for I_am_topmost.
      }
   1. Call a suitable constructor for each direct,
      nonvirtual base of X in left-to-right order.
      Pass FALSE as the value for I_am_topmost.
   2. Call suitable constructor for each member of X.
      Pass TRUE as the value for I_am_topmost.
   3. Install the set of virtual-function tables for
      class X (this includes virtual-function tables
      for X's bases; see Virtual functions on page 100).
   4. Execute your constructor code.
   }
```

A constructor initializer can specify the initialization for a virtual base because a topmost constructor initializes all virtual bases. This implies that any constructor initializer for a virtual base will be ignored if that initializer is part of a constructor that is not topmost. For example:

```
struct V {
   V(char c) { ch = c; }
   char ch;
   };
struct A: virtual V {
   A(): V('a') { }
   };
struct B: virtual V {
   B(): V('b') { }
   };
struct X: A, B, virtual V {
   X() : V('c') { }
   A a;
   };
main () {
   X x;  // Uses V('c') for base V, ignores V('a')
         //  in A() and V('b') in B()
         // Uses V('a') for member a
```

```
   printf("%c %c\n", x.ch, x.a.ch); // Prints c and a
   }
```

Expanding what happens when **x** is constructed reveals the following steps for constructors for members of **X** (refer to the `constructor_code` procedure in the preceding pseudo-code):

```
0. Since I_am_topmost is TRUE:
      Initialize the virtual base pointers within A,
      B, and X to point to the location of V.
      Construct V passing argument 'c'.
            4. Execute V::V()'s user-written code
               ("ch = c").
   1. Construct A with A::A():
         0. Since I_am_topmost is FALSE, do nothing.
         4. Execute A::A()'s user-written code.
      Construct B with B::B():
         0. Since I_am_topmost is FALSE, do nothing.
         4. Execute B::B()'s user-written code.
   2. Construct member a with A::A():
         0. Since I_am_topmost is TRUE:
               Initialize the virtual base pointers
               within A and V (as constituents of
               member a), to point to the location
               of the V within a.
               Construct V within a, passing argument
               'a'.
                  4. Execute V::V()'s user-written
                     code ("ch = c").
               4. Execute A::A()'s user-written code.
   4. Execute X::X()'s user-written code.
```

The **printf** statement at the end prints out the characters **c** and **a**.

Copy constructors

Some constructors, called *copy constructors*, are used to control the semantics of copying a class object in so-called "initialization" contexts. A *copy constructors* for a class is any constructor that can be called with a single argument of the same class type. The first argument of a copy constructor must be a reference type, as in the following example:

```
// Valid copy constructors for class X:
X::X(const X&);
X::X(X&, float f = 1.2)
// An erroneous copy constructor:
X::X(volatile X)
```

The initialization contexts are contexts in which an object is copied, except for assignment. Specifically, these are the contexts:

- forms of user-specified initialization in which '=' is used
- function argument passing
- function value return

Example 27 When objects are copied

```
struct s {
   s();
   s(s&);
   s(int);
   };
```

```
s f(s);     // f takes and returns an s.
s x;
s y = x;    // Calls copy constructor to copy x into y
            //  (that is, y takes the value s(x))
f(y);       // Calls copy constructor to copy y to f's
            //  argument (that is, the argument takes
            //  the value s(y))
s f(s parm) { return parm; }
            // Calls copy constructor to copy parm to
            //  return result (that is, s(parm) is
            //  returned)
s z(1);     // Does NOT call copy constructor;
            //  calls s(1) to construct z
```

Compiler-generated copy constructors

The compiler generates a **public** copy constructor if you do not declare one in a class. If all bases and members of a class have copy constructors accepting **const** arguments, the generated copy constructor takes a single argument of type **const** *class_name***&**. Otherwise, it takes a single argument of type *class_name***&**, and therefore copying a **const** *class_name* object is not possible.

The semantics of a generated copy constructor are to copy each member and base via the appropriate copy constructor for the member or base. In practice, the MetaWare C++ compiler generates fast bitwise copy semantics for copying class objects when you have not defined any copy constructors.

Initializing declarations

When you specify an initialization that uses the parenthesized list of initializing expressions, copy constructors are not used, as exemplified by the declaration of **z** in Example 27 When objects are copied. Compare the following initializing declarations:

```
s z1(1);
s z2 = s(1);
s z3 = 1;
```

The first declaration calls the **`s(int)`** constructor directly to construct **`z1`**. The second one calls the **`s(int)`** constructor to construct **`s(1)`**, and then calls the copy constructor on the result to construct **`z2`**. The third is the same as the second, except that the compiler provides the conversion of `1` to `s` via the constructor.

Eliminating constructor calls

Using the assignment operator = directs the program to construct the value specified in a temporary object, and then copy the temporary to the variable being initialized with a copy constructor. If the compiler generates the copy constructor itself, and if it notices the semantics of the copy constructor are bitwise copy, the compiler can eliminate the call to the copy constructor and make `s z2 = s(1)` as efficient as `s z1(1)`, directly calling **`s(int)`** to construct **`z2`**. But where the semantics of the copy constructor are not bitwise copy, the optimization is not possible.

Destructors

A *destructor* for a class does the reverse of what a constructor does. It takes a valid class object and does anything necessary to destroy the object. For example, it might free up heap storage that was allocated during construction. A destructor for a class is a member function whose name is the same as the class name, prefixed with the character ~. For example:

```
class buf {
   void *storage;
```

```
public:
   ~buf();     // This is a destructor declaration.
   buf(int);   // This is a constructor.
   };
buf::buf(int howmuch){
   storage = new char[howmuch];
   int *i = 3;
   }
buf::~buf(){   // This is a destructor definition.
   delete [] storage;
   }
void func() {
   buf B(1024);
   ...
   }     // Storage is reclaimed by the destructor.
```

Here the destructor reclaims the storage allocated by the constructor.

Just as a constructor constructs bases and members and then executes user-written code, a destructor must execute user-written code and destroy bases and members. The order in which a destructor does these things is exactly the reverse of the order in which a constructor does them. This guarantees that the order in which objects are destroyed is the reverse of the order in which they are constructed. A destructor must free the storage associated with an object that was allocated via operator **new**.

A destructor can be **virtual** (in contrast to a constructor).

A destructor declaration cannot specify a return type or any parameters. However, no overload resolution is ever needed for a destructor, because it takes no arguments. Thus there can be at most one destructor for a class.

Calling a destructor

Destructors can be invoked as follows:

- for a local variable that has gone out of scope
- for a global variable when program termination occurs
- through use of operator **delete**
- when a temporary object created by the compiler is no longer needed
- for a member of a class that is being destroyed
- for a base of a class that is being destroyed

This list parallels the list of cases when a constructor can be called. Here is pseudo-code for any destructor, along with the two hidden parameters that identify whether the destructor call is a "topmost" call, and whether the context of the call is the operator **delete**:

```
destructor_code (boolean I_am_topmost,
                    boolean I_should_delete) {
   0. Install the set of virtual-function tables for
      class X (this includes virtual-function tables
      for X's bases; see Virtual functions on page 100).
   1. Execute the user-written destructor code.
   2. Call the destructor for each member of X.  Pass
      TRUE as the value for I_am_topmost and FALSE for
      I_should_delete.
   3. Call the destructor for each direct, non-virtual
      base of X in left-to-right order.  Pass FALSE as
```

```
   the value for I_am_topmost and for I_should_
   delete.
4. if (I_am_topmost) {
      For each base in a depth-first, right-to-left
      traversal of the class graph, construct each
      virtual base and pass FALSE as the value for
      I_am_topmost and for I_should_delete.
      }
5. if (I_should_delete), free the storage
   associated with the object by calling:
      X :: operator delete
   or, if there is no such operator, calling the
   global operator delete.
}
```

Compiler-generated destructors

If you do not specify a destructor for a class and one is needed, the compiler generates a **public** destructor. The generated destructor has, of course, none of your code.

Destructors and virtual member functions

Just as with constructors, the virtual-function tables for a class are set up before your destructor code is called. This means that any virtual member functions of a class that your code calls always call the actual function in a class. This subtlety about a destructor is important to remember, because a destructor cannot call an overriding member function in a derived class. This is because, due to the order of destruction, that derived class might be partially destroyed, and so the behavior of any overriding function might be unpredictable.

Characteristics of destructors

Other points to remember:

- A destructor cannot be declared **const** or **volatile**.
- A destructor must be a non-**static** member.
- You cannot take the address of a destructor.

User-defined conversions

Type conversions of objects to and from class objects can be specified by constructors and by conversion functions.

Such conversions, also called *user-defined conversions*, are used in addition to C++'s standard conversions. User-defined conversions can be applied to the following expressions:

- an argument to a function
- an initializer value
- a function return value
- an explicit type conversion (cast)
- an operand of an expression
- the *expression* in:

- **if(***expression***)**
- **switch(***expression***)**
- **while(***expression***)**
- **for (***S***;** *expression***;** *e2***)**

User-defined conversions are applied only if necessary. Their use obeys the usual access-control rules. At most one user-defined conversion is implicitly applied to a single value.

Conversion by constructor

There are two kinds of user-defined conversions. The first is conversion by constructor. A constructor for a class `X` accepting a single argument specifies a conversion from that argument type to the `X` type.

For example:

```
struct s {
   s(int);
   s(char *, float = 1.2);
   };
s f(s&, s) {
   return 10;     // Converts 10 to s via s(int)
   }
s x;
s y = f(3,        // Converts 3 to s as 1st arg to f
        "string"  // Converts "string" to s as second argument to f
        );
s y = 1;          // Converts 1 to s via s(int)
s z[3] = {10, y, 30}; // Calls s(int) for 10 and 30, and compiler-generated copy
                      //  constructor for y.
```

Conversion functions

The second kind of user-defined conversion is by a conversion function. A *conversion function* is a function with a name of the form **operator** followed by a sequence of one or more type specifiers, followed by zero or more *****s. Neither a return type nor argument types can be specified for a conversion function. A conversion function must be a non-**static** member of a class. The conversion specified is from the class of which the function is a member to the type given by the name of the conversion function. In contrast to constructors, conversion functions can convert to a non-class type, and can specify a conversion from class `Cfrom` to class `Cto` without having to include in `Cfrom`'s definition a conversion constructor. For example:

```
struct s2 { };
void takes_s2(s2&);
struct s {
   operator char *();    // Converts s to char *
   operator const s2();  // Converts s to const s2
   s(int);               // Can convert an int to an s
   };
char * f() {
   s x;
   if (x) ...;           // Converts x to char * before testing
   while (x) ...;        // Ditto
   s2 z = x;             // Converts x to s2, then uses bitwise copy
                         // (compiler-generated default) to copy result into z
   printf("%c",x[3]);    // Converts to char *
                         // before subscripting
   takes_s2(1);          // ERROR
```

```
   return x;             // Converts to char * before returning
   }
```

Notice that in the erroneous case, two user-defined conversions would have made it possible: first converting **1** to an **s** via constructor **s(int)**, and then converting **s** to an **s2** via the conversion function. But only one user-defined conversion is applied, and so this is erroneous.

User-defined conversion functions are inherited, and they can be virtual.

Converting to a known type

There are two possible contexts. The first is when a type you are converting to ("destination type") is known, and the second is when no destination type is known. A destination type is known in the following contexts:

- an argument to a function (the destination type is the argument type)
- an initializer value (the destination type is the type of the object being initialized)
- a function return value (the destination type is the return type)
- an explicit type conversion (the destination type is the cast type)

These rules apply when a destination type is known:

1. Apply user-defined conversions first to see if you can convert exactly to the destination type.
2. If not, see if you can convert to the destination type, ignoring any **const** and **volatile** qualifiers.
3. If not, see if it is possible to convert to a type that can in turn be converted (without user-defined conversions) to the destination type.

For example:

```
struct s {
   operator int();
   operator const int();
   };
s x;
int i = x;     // Converts via operator int()
const j = x;   // Converts via operator const int()
int k = int(x) > 1 ? ... : ... ; // Converts via
                                 // operator int()
```

Converting to an unknown type

A specific destination type is not known in the following contexts:

- an operand of an expression
- the *expression* in:
  - **if(***expression***)**
  - **switch(***expression***)**
  - **while(***expression***)**
  - **for (S** *expression***; e2)**

In these cases, conversion is attempted to either an integral type, a floating-point type, a pointer type, or a combination of these types, depending on the context.

Overloaded operators and user-defined conversions

Because a call to an overloaded operator can require a user-defined conversion, ambiguities can arise between user-defined operators and pre-defined C++ operators. For example:

```
struct s {
   operator +(int);
   operator int();
   };
s x,y;
int z = x+y;      // Error: ambiguous
```

Either of the following solutions is valid:

```
x.operator +(y)
```

or:

```
int(x) + int(y)
```

Both employ user-defined conversions. The compiler responds with a diagnostic message.

Conversion by constructor

Whenever conversion by constructor is considered, overload resolution is used to choose a constructor from among many. If overload resolution fails in finding a single constructor, conversion by constructor is not considered. For example:

```
struct s {
   s(int);  s(float);
   };
s x = 1;      // Converts via s(int)
s y = 1.2f;   // Converts via s(float)
```

Using C-style casts for conversion functions

Not all conversion functions can be called by the function-style syntax of casting; sometimes old C-style syntax is necessary:

```
struct s {
   operator int();
   operator const int();
   };
s x;
int i = int(x);       // Converts via operator int()
int j = (const int)x; // Converts via operator const int()
```

So, although C++ discourages use of the C-style cast, it is useful in naming certain conversion functions. Alternatively, you could more clumsily use a **typedef**:

```
typedef const int CI;
struct s {
   operator int();
   operator CI();
   };
s x;
int i = int(x);  // Converts via operator int()
int j = CI(x);   // Converts via operator CI()
```

Controlling storage allocation

Operators **new** and **delete** allow control over storage allocation.

Creating objects

When an object is created with the **new** operator, an **operator new()** function is implicitly called to obtain the storage necessary. If the object being created is of some class type `C`, `C::operator new()` is invoked if it exists. The usual access and overloading rules apply.

The following rules apply to **operator new()**:

- It must be declared with its first argument of type **size_t** (defined in the header `stddef.h`).
- It must return **void***.
- It can have any other parameters besides the **size_t** first argument.

Here is an example of how to declare **operator new()**:

```
struct s {
   void * operator new(size_t amount);
   void * operator new(size_t amount, char *msg);
   };
```

The first argument to **new** is the size of the object being allocated. Read about the **new** expression in The new operator on page 20 to see how other arguments are passed to **new**.

operator new() should return the address of the storage allocated.

You might provide a class-specific **operator new()** to efficiently zero out the storage allocated to the class:

```
void * s::operator new(size_t amount) {
   return calloc(1,amount);
   }
```

Destroying objects

When an object is destroyed with the **delete** operator, an **operator delete()** function is implicitly called to free the storage as necessary. If the object being destroyed is of some class type `C`, `C::operator delete()` will (usually) be invoked if it exists. The usual access rules apply. Overloading is not necessary because there can be only one **operator delete()** for a single class.

The following rules apply to **operator delete()**:

- It must be declared with its first argument of type **void ***.
- It can have an optional second argument of type **size_t** (defined in the header `stddef.h`)
- It must return **void**.

Here is an example of how to declare **operator delete()**:

```
struct s {
   void operator delete(void *ptr);
   // OR:
   // void operator delete(void *ptr, size_t size);
   };
```

We say “usually invoked” because what really happens depends on whether a destructor exists for class `C` and whether that destructor is **virtual** and overridden.

Destroying objects when no destructor exists

Consider the simple case where no destructor exists for a class `C`, but `C::operator delete()` is declared. Then the **delete** expression calls `C::operator delete()` with the pointer to the object being deleted as the first value and, if **delete** is declared with two parameters, **sizeof(C)** is passed as the second argument.

```
s *p;
delete p;   // Calls s::operator delete(p)
```

Calling delete when there is a destructor

Consider the case where a destructor exists for class `C`. Recall that destructors must call the operator **delete** if they are told to free the storage associated with an object. Thus, for an object of class `C`, if **~C()** exists, the **delete** expression calls the destructor for that object, passing it the value `TRUE` for the hidden parameter *`I_should_delete`*. The destructor itself calls the operator **delete** defined in the class containing the destructor.

```
struct s {
   void operator delete(void *ptr);
   ~s();
   };
s *p;
delete p;  // Calls s::~s() with TRUE for I_should_delete
```

Here the delete expression does *not* directly call the **delete** operator. It calls the destructor for the class, telling the destructor to delete the object.

Calling the destructor instead means that if `C`'s destructor was overridden by a derived class `D`, it is `D`'s destructor and `D`'s **delete** (if any) that will be called; `C`'s **delete** will never be called:

```
struct s {
   void operator delete(void *ptr);
   virtual ~s();
   };
struct s2 : s {
   void operator delete(void *ptr);
   ~s2() ;
   };
s *p = new s2;  // Creates an s2, put pointer in p
delete p;       // Calls s2::~s2() with TRUE for I_should_delete
```

In this example, the destructor is virtual and is overriden by **`s2`**'s destructor. Therefore **`s2`**'s operator **delete** is called, not **`s`**'s. This is indeed appropriate behavior, because the pointer originated from an allocation of **`s2`**, not from an allocation of **`s`**.

Special operators are static member functions

Operators **new** and **delete** are implicitly static member functions, even if you do not declare them **static**. For **new**, this is because there is no object yet created to be passed as the **this** parameter. For **delete**, it is because the process of deletion invalidates the **this** parameter. Because the functions are **static**, they cannot be **virtual**.

Assignment operator

The assignment operator for a class `C`, `C::operator =`, is invoked whenever an assignment (=) is made into an object of type **C**.

Example 28 Assigning a string

```
struct str {
   str& operator =(char *str);
   str& operator =(str&);
   str() { string = 0; }
   char *string;
   };
// Assigns string s into this->string, by value:
str& str::operator =(char *s) {
   if (string) delete string;
   string = new char[strlen(s)+1];
   strcpy(string,s);
   return *this;
   }
str& str::operator =(str& other) {
   // Checks for self-assignment; if so, does nothing:
   if (&other == this) return *this;
   // Now calls operator =(char*) to finish the job:
   return *this = other.string;
   }
str x, y,z;
void f() {
   y = "hello";
   x = y;            // x.str::operator=(y)
   x = "hi";         // x.str::operator=("hi")
   y = x = "fnc";    // y.str::operator=
                     //  (x.str::operator("fnc"))
   }
```

In Example 28 Assigning a string, class `str` contains a string. The user-defined assignment operators can copy into that string from another string (`operator =(char *)`) or from another `str` object (`operator = (str&)`). In either case the copying is done by value — that is, the string content, rather than the pointer to the string, is copied.

`operator =(char *)` reclaims any storage in the `str` object being assigned, allocates enough space to hold the copy, and then makes the copy. `operator =(str&)` checks whether the object for which it is called and the argument are the same, in which case nothing is done; otherwise, it calls the other assignment operator to do the actual string copy.

The assignment operators shown in Example 28 Assigning a string return ***this**. This is recommended practice, because it allows for expressions such as:

```
y = x = "fnc"
```

This is equivalent to:

```
y.operator=(x.operator=("fnc"))
```

and is what you would expect with right-to-left assignment.

Special default assignment operator

A "default" assignment operator for a class `C` is one that can take a single argument of class `C`. Such an operator is the one invoked when an object of class **C** is assigned from another object of class `C` or class derived from `C`. The default assignment operator = is considered special: if you do not provide one, the compiler generates one (unless `C` is an anonymous union). The compound assignment operators +=, -=, *=, and so forth, are not considered special in this way.

If all bases and members of a class **C** have assignment operators accepting **const** arguments, the generated default assignment operator has a single parameter of type **const C&**. Otherwise, it has a single parameter of type **C&**, and therefore assigning from a **const** `C` object is not possible. The return type of the generated default assignment operator is **C&**, and the operator returns ***this** as its result. The semantics of a generated default assignment operator is to copy each member and base via the appropriate assignment operator for the member or base. In practice, the compiler generates fast bitwise copy semantics for this copying when there are no user-defined assignment operators involved.

In a manner similar to constructors, compiler-generated assignment code takes care of assigning bases and members, and recognizes whether it is the topmost assignment operator, as shown in the following pseudo-code:

```
assignment_code(boolean I_am_topmost) {
   0. if (I_am_topmost) {
         Initialize all virtual base pointers for
         all bases.
         For each base in a depth-first, left-to-right
         traversal of the class graph, assign each
         virtual base and pass FALSE as the value
         for I_am_topmost.
      }
   1. Call the assignment operator for each direct,
      non-virtual base of C in left-to-right order.
      Pass FALSE as the value for I_am_topmost.
   2. Call the assignment operator for each member
      of C.  Pass TRUE as the value for I_am_topmost.
      This includes the virtual-function pointer
      since it is a (compiler-hidden) member of the
      object being copied.
   }
```

For example:

```
struct s1 {
   operator =(s1&);
   };
struct s2 {
   operator =(s2&);
   };
struct s3 : s1, virtual s2 {
   s1 mem_s1;
   }
void f() {
   s3 x, y;
   x = y;
   }
```

The compiler-generated assignment operator for **x** = **y** first assigns virtual base **`s2`** using `s2::operator=()`, then assigns base **`s1`** using `s2::operator=()`. **`mem_s1`** is then assigned using `s1::operator()`.

CAUTION Unfortunately, whereas compiler help in copying the contents of virtual (versus non-virtual) bases is essential in user-defined copy constructors, no such compiler help is available in user-defined assignment operators. Thus, the use of user-defined assignment operators in classes with virtual bases is hazardous and best avoided.

Chapter 10 — Function overloading

In This Chapter

Introduction

Unlike C, C++ allows multiple declarations for a function in the same scope. The function name is then said to be *overloaded*. When the function name is subsequently used, *overload resolution* selects one declaration from the many declarations.

Overload resolution

C++ provides two kinds of overload resolution: argument match and function match.

Argument-match overload resolution is by far the more common of the two overload resolution processes, and occurs in a function-call context. This matching chooses a function by comparing the types of the arguments passed with the parameter types for each candidate function declaration. The return type of the function is not considered during argument-match overload resolution.

Function-match overload resolution chooses a function by comparing the type of a given function with each candidate, until an exact type match is found.

The following example illustrates both kinds of resolution.

Example 29 Argument-match and function-match overload resolution

```
void f(int);
void f(float);
void f(char *);
// Argument-match overload resolution:
void g() {
   f(1);       // Calls f(int).
   f(1.2f);    // Calls f(float)
   f("str");   // Calls f(char *)
   }
// Function-match overload resolution:
void (*h1)(float)  = f;  // f(float) chosen
void (*h2)(int)    = f;  // f(int) chosen
void (*h3)(char *) = f;  // f(char *) chosen
void (*h4)(double) = f;  // Error: no exact match
```

Two function declarations by the same name are allowed when the parameter types are not identical; the return types do not matter:

```
void f(int);
int  f(int);  // Not valid: same parameter
              //  types as previous f
```

Some functions with different parameter types cannot be distinguished from each other in argument-match overload resolution because they accept the same parameter types. For example:

```
void f(int);
void f(int &);         // Parameter types differ, so allowed
void g() {
   f(1);               // Cannot tell which f; both take an int
   }
void (*h)(int &) = f;  // Selects f(int &)
```

However, function-match overload resolution can tell the difference.

The reason the return type is not considered in argument-match overload resolution is that using the return type would make it harder for compilers — and much harder for programmers — to recognize which function is chosen by overload resolution.

Argument-match overload resolution

When you call a given function with a given set of arguments, argument-match overload resolution chooses the best function from among all candidate functions that can possibly be called. A function is best unless it is bested by another function. A function is *bested* if, for any argument expression, another function has a better "score" on matching the expression's type to that of the parameter type of the function. There must be exactly one best function, or else the call is not valid. Two kinds of error can result:

- no best function
- multiple best functions

Example 30 Choosing the "best" function

```
void f(double);
void f(float);
void g(double,int);
void g(int,double);
void test() {
   f(1);     // Error: there are two "best" functions,
             //  f(double) and f(float); both require
             //  the same conversion from 1 to a
             //  floating-point number.
   f(1.2f);  // f(double) has a worse score than
             //  f(float), because converting 1.2f to
             //  double is worse than not converting
             //  it at all; hence f(float) wins.
   f(1.2);   // f(float) has a worse score than
             //  f(double) because converting 1.2 to
             //  float is worse than not converting
             //  it at all; hence f(double) wins.
   g(1,1);   // Error: there are no "best" functions.
             //  g(double,int) is bested by
             //  g(int,double) when considering the
             //  first argument, and vice-versa for
             //  the second argument.
   g(1,1.2); // g(int,double) scores better than
             //  g(double,int) for both arguments.
   }
```

Argument type matching

The type *T* of each argument expression is examined in turn and compared with the corresponding parameter type *A* of each candidate function. Each argument receives a score as follows:

Score 0 : Best possible score. If *T* equals *A*, or *T* is converted to *A* by any of the following trivial conversions on a type *X*:

```
X => X &
X => const X
X => volatile X
X => const volatile X
```

In other words, converting a type by adding **const** or **volatile** or by turning it into a reference type is the best possible conversion, if a conversion is necessary.

Score 1: If the conversion from *T* to *A* requires any of the conversions for score 0, plus:

```
X & => const X &
X & => volatile X &
X & => const volatile X &
X * => const X *
X * => volatile X *
X * => const volatile X *
```

In other words, adding **const** or **volatile** to the base type of a pointer or reference type is worse than score 0 (zero).

Each of the higher scores below specifies what more is permissible in addition to what is allowed by previous scores.

Score 2: Add integral promotions or conversions from **float** to **double**.

Score 3: Add standard conversions, excluding those standard conversions mentioned in Scores 4 and 5.

Score 4: Add conversion from `C*` to `B*` or from `C&` to `B&`, where `B` is a publicly derived base of class `C`.

Some conversions in this category can be better than others, but a simple scoring mechanism cannot be used to compare them.

The conversion `C*` to `B*` is better than `C*` to `A*` if `A` is a base of `B`; essentially, `B` is "closer" to `C` than is `A` so `C*` to `B*` is preferred over `C*` to `A*`. However, if `A` is not a base of `B`, both `C*` to `B*` and `C*` to `A*` have the same rank and neither is preferred. The same argument holds for `C&` to `B&` versus `C&` to `A&`.

Score 5: Add conversion of *X** to one of the following:

```
void *
const void *
volatile void *
const volatile void *
```

Score 6: Add a single user-defined conversion.

Score 7: If there is no parameter *A*, but there is an ellipsis in the function, so that *T* is accepted because the function declaration contains "**...**".

No sequence of conversions is considered if it is longer than another sequence that achieves the same effect. For example, `int->float->double` is not considered because `int->double` achieves the same effect and is shorter.

For the purposes of scoring, a function with *n* trailing default parameters is considered to be *n*+1 functions with differing numbers of parameters. For example:

```
void f(int, char = 'c', float = 1.2);
```

is considered to be these three candidates:

```
void f(int);
void f(int,char);
void f(int,char,float);
```

and so can match `f(1)`, `f(1,'x')`, and `f(1,'x',3.3)`.

For the purposes of scoring, a non-static member function is considered to have an extra first parameter type — the type pointed to by the **this** pointer. In converting the argument, no temporaries are introduced, nor are any user-defined conversions attempted to achieve a type match.

For the purposes of scoring, an ellipsis (**...**) in the function declaration matches any sequence of argument types, each then having the ellipsis score as already described.

```
void f(double);
void f(float);
void g(double,int);
void g(int,double);
void test() {
   f(1);       // Error:  Both functions score 3 on
               //  converting 1 to double or float.

   f(1.2f);    // f(double) scores 2 and f(float)
               //  scores 0 on the argument, so
               //  f(float) wins.

   f(1.2);     // f(float) scores 2 and f(double)
               //  scores 0 on the argument,
               //  so f(double) wins.

   g(1,1);     // Error: g(double,int) scores (3,1)
               //  and g(int,double) scores (1,3).
               // So each bests the other in
               //  one of the arguments.

   g(1,1.2);   // g(int,double) scores (1,2)
               //  and g(double,int) scores (3,3),
               //  so g(int,double) clearly wins.
   }
```

Function-match overload resolution

In function-match overload resolution, a function name is used without any arguments, and it is converted to another function type. Function-match overload resolution chooses the declaration of the name that exactly matches that other function type.

Function-match overload resolution occurs in these contexts:

- a function pointer being initialized
- a function pointer on the left side of an assignment
- an argument in a function call where the corresponding parameter is a function pointer
- a function return expression having a function pointer type

If an exact match is not possible, function-match overload resolution fails with no solution.

```
void f(int);
void f(float);
void f(char *);
// Examples of function-match overload resolution:
//  typedef for simplicity
typedef void (*func_taking_int)(int);
func_taking_int test() {
   void (*h1)(float) = f;  // f(float) chosen
   void (*h2)(char *);
   h2 = f;                 // f(char *) chosen
   extern void g( void (*)(int) );
   g(f);                   // f(int) chosen
   return f;      // f(int) chosen as return result
   }
```

Overloading and scope

C++ allows multiple declarations for a function, as long as they are in the same scope. However, C++ does not allow a function in one scope to overload a function in another scope. For example:

```
void f(int);
test() {
   extern void f(char *);
   f("hi");         // OK, calls f(char*).
   f(1);            // Error:  no function callable
   }
```

At the call **f(1)**, there is just one candidate function: **f(char*)**. The global **f(int)** is hidden, not overloaded by **f(char*)**.

Operator functions

In C++ you can define your own operators by declaring functions that allow the language's built-in operators such as + to work on types that you define. These are called *operator functions*. The name of an operator function has the form **operator** followed by one of the C++ operators in the following list.

```
new     delete
+       -       *       /       %       ++      --
^       &       |       ~       <<      >>
=       +=      -=      *=      /=      %=
^=      ~=      |=      !=      >>=     <<=
<       >       ==      <=      >=
&&      ||      !
,       ->*     ->      ()      []
```

The following operators *cannot* be declared:

```
.       .*      ::      ?:      sizeof
```

Both the unary and binary forms of **+**, **-**, *****, and **&** can be declared; whether a function is a unary or binary operator depends on whether it takes one or two arguments. For example:

```
class s {
   int operator -(int); // Binary:  Remember the hidden "this" parameter.
   int operator &();    // Unary:  The argument is the hidden "this" parameter.
   };
int operator+(int,s&);  // Binary
double operator *(s&);  // Unary
```

An operator function permits defining, say, the operation + on a string class — perhaps to indicate string concatenation:

```
string s1(80), s2(80);
string operator + (string &parm1, string &parm2);
. . .
string s3(160) = s1 + s2;'
```

Declaring an operator function

An operator function can be declared in either of two ways:

- as a non-static class member function
- as any other declaration (that is, *not* within a class), provided at least one of its parameters is a class type or a reference to a class type

The distinction is due to the way an operator function is chosen in an expression. When you write a binary operator expression such as:

```
x + y
```

the C++ built-in binary addition operator is used if neither argument is a class type. Otherwise, the built-in operator cannot possibly apply, and so the following two scopes are examined for a declaration of an operator function to be called:

- the nearest scope that is not a class scope
- the class scope where *x*'s type was declared, if **x** is a class object

If there is an operator in both scopes, overload resolution is used to select which operator is used. Even though the operator function might not be overloaded in either scope, overload resolution must be used because candidates come from two different scopes.

Similarly, when you write a unary operator expression — for example, **+x** — the same two scopes are considered.

Overloading operator functions

Just as for non-operator functions, you can also overload operator functions. In general, the overloaded operators in both scopes are candidates from which overload resolution chooses a single operator.

Because operator functions are just functions with special names, you can still call them with the same syntax you use to call any function. In fact, you can use the operator function name just as you would use any other function name:

```
operator+(x,y);
x.operator+(y);   // operator+ must be a member function.
int (s::*add)(s&) = x.operator+;
g(operator+);     // Passes operator+ as an argument
```

Calling operator functions implicitly

The only thing special about an operator function is that it can be called implicitly in an expression such as **x + y**. Essentially, **x + y** is equivalent to either **operator +(x, y)** or **x.operator+(y)**, depending on where the operator function is found. You could write **operator +(x, y)** or **x.operator+(y)** yourself, but neither means the same thing as writing **x + y**, where the compiler chooses between the two.

In Example 31 Overloading an operator, **operator*** is overloaded in the scope of the complex class and in the global scope.

Example 31 Overloading an operator

```
struct complex {
   complex operator +(int);          // Allows complex+int
   complex operator +(complex);      // Allows complex
                                     //  +complex
   complex();                        // Constructor
   complex(double,double);           // Constructor
   };
complex operator *(complex&,int);    // Allows complex*int
complex operator *(complex&,double); // Allows complex*double
complex operator +(int,complex&);    // Allows int+complex
test() {
   complex c1(1, 3), c2(-1, 6);
   complex c3;
   c3 = c1 + c2;  // Equivalent to c1.operator+(c2)
   c3 = c1 + 1;   // Equivalent to c1.operator+(1)
```

```
c3 = c1 * 2.7; // Equivalent to operator*(c1, 2.7)
c3 = c2 * 1;   // Equivalent to operator*(c2, 1)
c3 = 1 + c1;   // Equivalent to operator+(1, c1)
c3 = 1 * c2;   // Error: no operator found
}
```

Operator functions versus built-in operators

The usual properties that apply to built-in operators do not necessarily apply to operator functions. For example, **1** + **x** is the same as **x** + **1** for the built-in +. However, as shown in Example 31 Overloading an operator, **c1** + **1** and **1** + **c1** call different functions. And although you could write **c2 * 1**, no operator function applied for `1 * c2.` `++x` is the same as `x += 1` for built-in ++ and +=, but this might not hold for operator functions ++ and +=.

Where to place an operator function

You have two choices of position for an operator function. You normally place an operator function within a class if one of the following conditions applies:

- The function requires access to the private data of the class.
- The function is more efficient if it can assume details of the class representation that only member functions know.
- The function is **operator=**, **operator[]**, **operator()**, or **operator->**. These must be non-static member functions.

Binary operators

The expression **x** *OP* **y** is interpreted either as **x.operator** *OP***(y)** or as **operator** *OP***(x, y)**, with overload resolution choosing among possible candidates. The binary **operator=** is defined for a class, unless you define it. See Assignment operator on page 131 for more information.

Prefix unary operators

OP **x** is interpreted as either **x.operator** *OP***()** or **operator** *OP***(x)**, with overload resolution choosing among the candidates.

Postfix unary operators

Operators ++ and -- can be both prefix and postfix, and so merit special attention. Postfix operator functions ++ and -- are declared with two parameters, the second of which is type **int**. When called, `0` (zero) is passed as the second argument:

```
class s { ... };
operator ++(s&);         // Prefix
operator ++(s&, int);    // Postfix
s x;
++x;                     // Calls prefix
x++;                     // Calls postfix
```

Postfix function-call operator

The function call:

```
function_expression ( arguments )
```

is considered an *n*-ary operator with *function_expression* as the first operand and the arguments as the remaining operands, for a total of *n* operands. **operator()** must be a non-**static** member function, so the *function_expression* can be only of a class type. *function_expression* `(` *arguments* `)` is equivalent to:

```
function_expression.operator() ( arguments )
```

You could use **operator()** to implement a multi-dimensional array:

```
class _3Darray {            // Of ints, say
   int *storage;
   int one, two, three;     // Dimensions
   int& operator()(int x, int y, int z) {
      return storage[x*two*three + y*three + z];
      }
   };
_3Darray x;
. . .
x(1, 2, 3) = 1;     // Calls x.operator()(1, 2, 3).
```

Postfix subscript operator

Operator **[]** is considered a binary operator. **x[y]** is interpreted as **x.operator[](y)**. **operator[]** must be a non-static member function.

You could use **operator[]** to implement an associative array:

```
class AA {
   int& operator[](int);
   };
AA x;
. . .
x[1] = 1;
x[4000] = 2;
// At this point there might be only two objects stored, at subscripts 1 and 4,000.
```

Postfix dereference operator

Operator **->** is considered a unary operator. **x->y** is interpreted as **(x.operator->())->y. operator()** must be a non-static member function. **operator->** must therefore return something that can be used as a pointer.

Smart pointers

You could use **operator->** to implement so-called "smart pointers"; that is, pointers that do something extra when dereferenced, such as make sure that the dereferenced area is paged in from secondary storage. For example:

```
class object {    // That is paged to disk
   int field;
   };
class smart_ptr {
   object *p;
   long disk_location;
   object * operator->() {
      // Make sure x.p is in memory.
      . . .
      return p;
      }
   };
smart_ptr P;
. . .
int i = P->field;
```

Make sure **operator->** does not return the same type that it takes:

```
class loop { loop operator->(); int field; };
loop x;
x->field;
```

```
// Means (x->operator->())->field, which means
//  ((x->operator())->operator())->field, which means
//  (((x->operator())->operator())->
//  operator())->field, . . .
```

In This Chapter

Introduction

Templates are a powerful feature of C++. With templates you can define generic families of functions or classes. That is, using a template, you can declare a **class** whose members can assume various types, based on parametric substitution, or you can define a family of functions whose arguments and local variables can assume various types. Unlike macros, instantiations of templates are accompanied by full type checking.

NOTE The *<bAsIs>Annotated C++ Reference Manual* makes reference to “function templates” and “template functions.” A *template function* is an instantiation of a function template. The terms “class templates” and “template classes” are related the same way. A *template class* is an instantiation of a class template.

Declaring and using a template

This is the form of a template declaration:

```
template <arg[, arg]...> declaration
```

arg is one of the following:

```
class identifier
```

```
argument_declaration
```

If *arg* is of the form **class** *identifier*, the corresponding argument in a class object or function declaration must be a fundamental or derived type, and in the body of the class object or function declared, *identifier* stands for that type.

If *arg* is of the form *argument_declaration*, it is the same as an argument for a function, and the corresponding argument in a class object or function declaration can be any of the following:

- a constant expression
- the address of an object with external linkage
- the address of a static class member

declaration is either the name of the class template or a function declaration.

This is the syntax for instantiating a template with a class-object or function declaration:

```
template_name<arg[, arg]...> declaration
```

where *template_name* is the same as *template_name* in the template declaration, and *arg* is either a fundamental or derived type, or an expression. *declaration* is either the name of the class object or the declaration of the function.

An example: declaring and using a class template

Example 32 Defining a class template declares a class template and several instances of that class.

Example 32 Defining a class template

```
template <class node> class tree {
   node *left_child, *right_child;
   int size;
public:
   tree();
```

```
   node& operator[](int);
   node& element(int i) { return left_child[i]; }
   };
tree<int> oak[20];
tree<unsigned> pine[30];
tree<unsigned> pinon[10];  // Same element type as pine
```

In Example 32 Defining a class template, `tree` is a template class in which type `node` can assume any fundamental or derived type. Each instantiation of `tree` forms a class object different from any other class object declared using this template. For example, `oak` is an array of class objects, and within each class object defined for `oak`, elements having the type `node` in the template definition assume the type **int**. But `pine` and `pinon` are arrays of class objects whose `node` members assume the type **unsigned**.

Template type equivalence

Two instances of a template are of the same type if the names are the same, and if their argument lists are the same or differ only in constant expressions that evaluate to the same value; for example:

```
template <class mytemp,
          int size1,
          int size2> class mybuf {
   mytemp *buf1;
   buf1size[size1];
   buf2size[size2];
   mytemp *buf2;
public:
   mybuf();
   }
mybuf<char,120,80> line1;
mybuf<char,3*40,160/2> line2;
```

This example declares a class template and two objects that use it. The argument lists for the two objects `line1` and `line2` are different, but they both use the same class name (`mybuf`), and their argument lists contain constant expressions that evaluate to the same values. `line1` is therefore of the same type as `line2`.

Function templates

A function template allows you to declare a family of logically related but separate functions. These functions can perform the same action, but operate on different data structures. To declare a function template, make the *declaration* portion of the template declaration a normal C++ function declaration.

For example, suppose you define a class that implements a binary-tree data structure. Each class object contains a pointer to a left child node and a right child node. You can then define a family of functions to manipulate the tree, as shown in Example 33 Defining a family of functions.

Example 33 Defining a family of functions

```
tree<int> oak, willow;
tree<float> spruce;
template <class node> void treeop(tree <node> N) {
   node *p1, *p2;
   ...
   }
void func() {
   treeop(oak);      // Calls treeop(tree<int>),
```

```
                    //  passing oak
   treeop(spruce);   // Calls treeop(tree<float>), passing spruce
   treeop(willow);   // Calls treeop(tree<int>), passing willow
   void (*fp)(tree<float>) = treeop;
      // Finds the function created by the call to
      //  treeop(spruce) and assigns it to fp
   fp(spruce);       // Same as treeop(spruce)
   void (*fp2)(tree<long>) = treeop;
      // Instantiates treeop(tree<long>) and assigns it to fp2
   }
```

The template definition of **treeop** in Example 33 Defining a family of functions declares an unbounded set of functions, each named **treeop** and taking a single parameter of type `tree` (see Example 32 Defining a class template). When the compiler encounters **treeop(*oak*)**, it provides a definition for that **treeop** function that can use the `tree<int>` parameter *oak*:

```
void treeop(tree<int> N) { int *p1, *p2; ... }
```

Notice that the parameter *node* has assumed the type **int**. The compiler inferred this because the formal parameter was `tree<node>` and the actual parameter was `tree<int>`.

Later, when it encounters **treeop(*spruce*)**, the compiler provides a similar definition that uses the `tree<float>` parameter *spruce*:

```
void treeop(tree<float> N) { float *p1, *p2; ... };
```

When **treeop(*willow*)** is encountered, the compiler has already found a **treeop** function that can be called with `tree <int>`, and so does not provide another definition.

To initialize function pointer `fp`, the compiler uses `treeop<float>`, which is already defined. To initialize `fp2`, the compiler instantiates a `treeop<long>` function and assigns its address to `fp2`.

How the compiler chooses a function

The presence of templates modifies the compiler's behavior in searching for functions to call or to take the address of. Here is what happens, in order:

1. The compiler tries to find a function that can be called without any type conversions — this is called an "exact match".
2. If there is no exact match, the compiler looks for a function template that can be instantiated and which then provides a function that can be called with an exact match.
3. If no such function template is found, the compiler performs a normal function search as if there were no template functions. This may involve overload resolution, argument conversion, and so forth.

So, by the time the compiler encounters the call **treeop(*willow*)** (for example) in Example 33 Defining a family of functions, a function already exists (by previous template instantiation at the call to **treeop(*oak*)**) that could be called with an exact match, so the compiler does not instantiate another function definition.

Given these rules, in Example 33 Defining a family of functions you could provide your own definition of a "specialized" **treeop()** function, in case you do not like what would be instantiated from the template definition:

```
void treeop(tree<double> N) {
   double D;
   ...
   }
```

```
tree<double> apple;
tree<char *> ibm;
void func() {
   treeop(apple);    // Calls non-template
                     //  treeop(tree<double>)
   treeop(ibm);      // Calls template
                     //  treeop(tree<char *>)
   }
```

Here the specific definition of `treeop(tree<double>)` causes the compiler to find this function first at the call to **treeop(***apple***)** (by the "exact match" rule) and so there is no need to instantiate one from the template definition. This works even if your definition appears *after* the call to **treeop(***apple***)**. You might use specific definition of functions to write a faster version of the general template function for certain argument types.

Declaring a function object more than once

You can declare function objects using a function template many times, and the compiler makes sure that only one instance for each set of distinct template arguments is generated. You declare a function template object when you call a template function or take its address.

Class templates

A class template allows you to declare a family of logically related but separate class objects whose member functions perform the same actions, but operate on and return different data types.

See Example 32 Defining a class template for an example of how to define a class template and declare instances of it.

Rules for declaring a class template

The following rules apply to class template declarations.

- A template declaration can appear only at file scope.
- A class template name must be unique in a program. It cannot refer to any other template, class, function, object, value, or type in the same scope.
- For a non-type template argument, the type of an *arg* must match that given in the template declaration for that argument.
- You can use the *template_name* only by instantiating it.

Template member functions

When you declare a template class, any member function of the class is implicitly a template function. Each such member function has the template arguments of the class of which it is a member as its own template arguments.

Example 34 Defining subscript member function operator [] (int)

```
template <class node> class tree {
   node *left_child, *right_child;
   int size;
public:
   tree(int s) { size = s; }
   node& operator[](int);
   node& element(int i) { return left_child[i]; }
   }
```

```
tree<int> oak(20);
// The implicit template function, which has
// the same template arguments as class tree:
template <class node> node& tree<node>::
   operator[](int i){
      if ( i < 0  || size <= i)
         cout << "Tree: subscript error.";
   return left_child[i];
   }
```

In Example 34 Defining subscript member function operator [] (int), a subscript member function **operator[](int)** is defined. It has the same arguments as its **template** class declaration.

Notice, however, the syntax for saying that you are defining a member function of a template class: `tree<node>::operator[](int i)`. The `tree<node>::` specifies that the function being defined is a member of the general `tree` template.

Template friends

A function that is a friend of a **template** class is not automatically a **template** function.

```
template <class job> class todo {
   friend void next_job();
   friend todo<job>* override(todo<job>*);
   friend todo* mistake(todo*); // Not valid
   };
```

In this example, the function **next_job()** is a **friend** of all `todo` classes. In addition, each `todo` object has an appropriately typed function called **override()** as a **friend**. But the declaration of **mistake()** is not valid, because the type `todo` does not exist. There are only **template** types, such as `todo`**<int>**.

Static members and variables in a template

Unlike normal classes, in a **template** class each generated class or function has its own copy of any **static** variables or members. This is because the members can assume different types.

For example:

```
template <class myclass> class uclass {
   static myclass aname;
   }
uclass<int> ii;
uclass<char*> cc;
```

In this example, `ii` has a **static** class member **aname** that has type **int**. `cc` has a **static** class member **aname** that has type **char***.

Template definitions and definition replacement

There can be exactly one definition for each class or function template of a particular name in a program. The compiler, when it instantiates a class or function template, uses the template definition unless you have provided a specific class or function definition to replace the template definition.

If you define a function that is not a template function, but its type and parameters exactly match an instantiation of a template function with the same name, the definition of the non-template function takes precedence over the definition of the matching template function.

Likewise you can provide a specific class definition to take precedence over the definition of a class template instantiation. For example:

```
class mybuf< char *, 100, 200> {
   ...
   };
mybuf<char*,100,200> line3;
```

Here the type of `line3` does not cause a template instantiation of `mybuf` (see Example 33 Defining a family of functions), but instead uses the specific class you have provided. You might use a specific class definition to obtain efficiency (for some template arguments) greater than would be afforded by instantiation of the general template.

The single-copy problem

When you use templates, you must ensure that, for each distinct set of parameters to a template, there is only a single copy of an instantiated function or static class data member. We call this the "single-copy problem."

The default operational mode of the MetaWare C/C++ compiler is to always generate instances of function templates and classes when they are used in a source module — one instance in the source module is generated per set of parameters to the template.

The MetaWare C/C++ compiler currently supports the following solutions to the single-copy problem:

- Merging global template instantiations, also referred to as the *COMDAT solution*. This is the default solution for template instantiation and virtual-function table generation when you use the ARC® International ELF linker. See Using COMDAT to merge global instantiations on page 151 for a discussion of this solution.
- Controlling instantiations at compile time. This is an alternate method for template instantiation. Use this alternate solution if you are not linking with the ARC® International ELF linker. Specify option **-Hnocomdat** when you use this solution. For more information, see Template instantiation issues on page 189.

Using COMDAT to merge global instantiations

The ELF (Executable and Linking Format) and OMF (Object Module Format) files generated by the MetaWare C/C++ compiler allow the linker to choose one of the multiple copies of a structure and discard the others, so that only one copy appears in the final linked program. This "common data" approach is known as the *COMDAT solution* to the single-copy problem. The COMDAT solution is the MetaWare C/C++ compiler's default solution for template instantiation and virtual function table generation. This solution is the default because option **-Hcomdat** is on by default.

When COMDAT support is enabled, these toggles are turned on:

- `Inline_common`
- `RTTI_common`
- `Template_common`
- `Vtable_common`

A related toggle, which option **-Hcomdat** does not turn on, is `Class_common_always`. (For more information about these toggles, see the *MetaWare C/C++ Programmer's Guide*.)

If you plan to use a linker that does not support COMDAT sections, you must specify option **-Hnocomdat** and adopt the manual instantiation techniques outlined in Template instantiation issues on page 189.

> **NOTE** In cases where there is a conflict between a template function and a user-defined function, both with the same name and argument list, the linker gives precedence to the user-defined function and discards the template function.

Limitations of COMDAT

The COMDAT solution does not allow you to do the following:

1. Declare apocryphal some class template *T* in Module 1 and provide definitions for its member functions; then
2. Declare class template *T* in Module 2 without member function definitions, and instantiate template *T* in Module 2.

Problems with matching const parameters

Instantiating a function template requires an exact match of the function formal-parameter types and the actual-parameter types. This requirement can be a nuisance.

For example, suppose you generalize the ANSI Standard C **bsearch()** routine with a template:

```
template <class T> T *bsearch(
   const T key,
   const T *base,
   int nmemb);
```

Because you used **const**, your generic search routine "promises" never to "clobber" the elements being searched, or the key. This is a valuable promise. But when you try to call **bsearch()** the call fails, because `123` is not a **const**, and `a` is not of type pointer-to-**const**:

```
extern int a[10];
int *solution = bsearch(123, a, 10);
```

In fact, to successfully call **bsearch()**, you must write this:

```
extern int a[10];
int *solution = bsearch(
   (const int)123,
   (const int *) a,
   10);
```

One way to alleviate this problem is to define an additional **inline** template that allows non-**const** parameters, but which calls the "real" template:

```
template <class T> inline T *bsearch(
   T key,
   T *base,
   int nmemb) {
      bsearch(
         (const T)key,
         (const T*)base,
```

```
        nmemb);
   }
```

This template allows the call to **bsearch(*123*, *a*, *10*)** to succeed. The second template is called, which turns around and calls the first. Because the second template is inlined, there is no performance penalty.

To permit a non-**const** parameter to be "matched" with a **const** parameter in a function template, turn on toggle `Template_trivial_conversions`. This toggle (on by default) allows *trivial conversions* (as defined in the *<bAsIs>Annotated C++ Reference Manual*) when arguments are matched to function templates. This conversion allows the call to **bsearch(*123*, *a*, *10*)** to work without introducing the intermediate **inline** template.

Using newer function template matching rules

By default, the compiler currently implements function template matching rules consistent with the October, 1996 version of the ANSI C++ draft standard.

- The *<bAsIs>Annotated C++ Reference Manual* rules say that if you can find an exact match on a non-template, you must call that non-template rather than considering any templates.
- The October, 1996 ANSI C++ draft standard says that you must consider both equally.

To implement the older function template matching rules (rules consistent with the *<bAsIs>Annotated C++ Reference Manual*), you must turn off toggle `Function_template_deduction`.

> **NOTE** When toggle `Function_template_deduction` is on, it causes some old C++ programs to break.

The following example conforms to the *<bAsIs>Annotated C++ Reference Manual* specification but not to the October, 1996 ANSI C++ draft standard:

```
struct s { int operator+(int y); };
template <class T> operator+(T x, int y);
main () {
   s x;
   x+1;  // Ambiguous: Is '+' the  s::operator+ or the template operator+(int,int)?
   return;
   }
```

In this case, the **operator+** in the class and in the template match equally well, so the expression `x+1` is ambiguous.

Not defining a template's unreferenced member functions

When a program instantiates a class template that was declared or defined with toggle `Define_unreferenced_template_members` turned off, the only member functions that are defined are those member functions that are referenced. Such a reference to the member function can be (for example) by a call, or by the function's address being taken due to its presence in a virtual function table.

Prohibiting the definition of unreferenced member functions permits otherwise ill-formed function bodies to be "not compiled".

If a class template is used in more than one compilation unit, turning on toggle `Define_unreferenced_template_members` might result in the same function definition occurring in two distinct compilation units. Toggle `Template_common` (on by default) prevents linker errors in this situation.

> **NOTE** Toggle `Define_unreferenced_template_members`, as of this writing, does not affect the definition of unreferenced static member data. However, it will do so in a future version of the compiler.

In the following code, **main** calls the copy constructor of `B<A>`, rather than calling the constructor **B()**:

```
struct A { A(int) {} };
template<class T> struct B {
// Calls T's default constructor T()
   B(): data(T()) {}
// Calls T's copy constructor
   B(const T& d): data(d) {}
   T data;
   };
main () {
   extern A a;
   B<A> y(a);
   }
```

In this example, if unreferenced member functions were defined (that is, if the toggle were turned on) and **main** were to call **B()**, an error would result because **A** has no default constructor.

With toggle `Define_unreferenced_template_members` turned off, if **main** were to call **B()**, no error would result.

Deferring the definition of a template class

In later versions of the ANSI C++ draft document, the language specifies that the compiler must defer the definition of a template class until its definition is "needed".

Refer to section 14.7.1 in the September, 1996 version of the ANSI C++ draft standard:

Unless a class template specialization has been explicitly instantiated (14.7.2) or explicitly specialized (14.7.3), the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type.

The MetaWare C++ compiler defers the definition of a "needed" template when toggle `Defer_template_defn` is on. This toggle is off by default, however, because it affects other template programming styles adversely. For more information about toggle `Defer_template_defn`, see the *MetaWare C/C++ Programmer's Guide*.

The following example illustrates the concept of when the definition of a template class is needed. In this code, the **typedef** `QB` refers to `Q<B>`, but `Q<B>` is not yet needed:

```
template <class T> struct Q {
   T x;
   };
class B;
typedef Q<B> QB;
```

In this context, Q<B> establishes Q<B> as a template instantiation but does not elaborate the definition of Q<B>.

If the following statement is added, the compiler is obliged to define Q<B>, because the size of *x* is required to define *x*:

```
QB x;
```

At this point, the compiler generates a diagnostic message. Why? Because class B is not yet defined, the definition of member *x* (of type B) of Q<B> fails. This failure can be prevented if the definition of B is provided before the definition of *x*:

```
class B { ... };
QB x;   // Succeeds, because B is defined before x
```

There are two reasons for the deferral:

1. To prevent instantiating a template class until it is needed.
2. To allow certain forms of pre-declarations (as shown in the preceding example, where class B is pre-declared but not defined).

The definition of **x** in the preceding example requires a completely-defined object type QB, so QB is then instantiated.

NOTE The current version of the MetaWare C++ compiler defers the instantiation only in the **typedef** context. These contexts will increase in number in later versions of the compiler.

For example, Q<B> *p; does not require the instantiation, but the compiler performs the instantiation anyway.

Chapter 12 — C++ namespaces

In This Chapter

When to use namespaces

The global namespace that C++ inherited from C causes a problem: global names of functions, variables, types, enumerators, and so forth declared in one third-party library may clash with names in another third-party library, or in your own application.

For example, a graphics library from vendor A might have a method called **Curve::rotate(int degrees)**, and so might the math library from vendor B. Your only recourse is to change the name of the class `Curve` in one library or the other. Sometimes the change is difficult or impossible to make.

C++ namespaces help solve this problem.

Encapsulating library names

Using namespaces, you can encapsulate library names in a library namespace with a long name derived, say, from the name of the library vendor. Vendor A's **rotate** function is then known as:

```
A_graphics::Curve::rotate
```

and vendor B's as:

```
B_math::Curve::rotate
```

The external (mangled) names given to the linker for such encapsulated library names includes the namespace name, so each name is distinct to the linker.

Encapsulating names in your own code

Even if you are using a library that pre-dates namespaces, you can use namespaces in your own code to encapsulate your own identifiers that clash with those of the library.

How to specify a namespace

Names are introduced into a namespace by means of a **namespace** definition:

```
namespace identifier {
   ... list of zero or more declarations
   }
```

Namespaces are global. A **namespace** definition is syntactically a declaration, but it can appear only at the global scope or within another **namespace** definition. All names declared directly within a **namespace** definition, except those introduced by a **using** declaration (see The using declaration on page 159), are "owned" by the namespace.

Namespaces can be extended. If `identifier` is not declared in the current scope, it is declared as a namespace. If it is already declared, it must be declared as a namespace, and the new **namespace** definition "continues" the namespace:

```
namespace A {          // Declares A as a namespace
   int f() { ... }
   typedef int T;
   }
...
namespace A {          // Adds more to namespace A
   int g();
   }
```

This continuation allows a single large namespace to be defined over multiple header files.

A library vendor can choose a very long name for a namespace to minimize potential clashes with other vendors:

```
namespace Distributed_System_Object_Model_Library {
   ...
   }
```

Aliasing namespaces

Because it is unwieldy to use such a long name to refer to names in the namespace, you might want to use a namespace aliasing definition to create a shorthand name:

```
namespace DSOML =
   Distributed_System_Object_Model_Library;
```

Now `DSOML` is declared as a namespace, and it stands for the longer name in the scope of the alias definition.

Using the names declared in a namespace

You can explicitly denote the names declared in a namespace by means of the existing qualified name syntax for classes. For example, `Colors::invert` denotes member **invert** of namespace `Colors`:

```
namespace Colors {
   enum color {red, white, blue};
   typedef int color;
   color invert(color);
   extern color c;
   };
...
func() {
   Colors::color C = ...;
   Colors::invert(C);
   }
```

A name declared in a namespace can be defined outside the namespace, if it is not already defined within the namespace. You use the qualified name to denote the member being defined:

```
Colors::color Colors::invert(color c) { ... }
Colors::color Colors::c = red;
```

Just as with classes, the text following the qualification is considered to be in the scope of the namespace. So, although you must use `Colors::color` for the return type, namespace members `color` (the parameter type) and `red` (the initializer) can appear without qualification.

The using declaration

A **using** *declaration* declares a name *Name* from a namespace as a current declaration in the scope in which the **using** declaration appears. The declaration of *Name* is an alias of its declaration in the namespace. A **using** declaration uses a qualified name to refer to *Name*. For example:

```
func() {
   using Colors::red, Colors::white, Colors::blue;
   Colors::color x = red;
   using Colors::invert;
   x = invert(x);
   }
```

NOTE Although MetaWare C++ allows the comma-separated list shown in this example, as of this writing the ANSI X3J16 draft standard does not.

Overloading functions with a using declaration

Just as with normal declarations, a **using** declaration can introduce a function that overloads other functions in the same scope. And just as with normal declarations, a **using** declaration can cause a duplicate declaration if a non-function of the same name has already been declared:

```
func() {
   extern int invert(int);
   using Colors::invert; // invert is now overloaded.
   // OK if invert is a function
   using Other_namespace::invert;
   Colors::color C = ...;
   invert(C);           // 1: Selects from 3 inverts.
   typedef int T;
   using Other_namespace::T; // Error!
   }
```

Unlike normal declarations, **using** declarations can overload functions with the same argument types. The resulting ambiguity is detected at the point of use, rather than at the point of declaration. For example, if `Other_namespace::invert` has the same signature as `Colors::invert`, then at **1:**, when the call is made to **invert(***C***)**, an error message results.

Precedence of non-using over using declarations

If there is an ambiguity between two functions with identical argument types, where one function is introduced by a **using** declaration and the other by a non-**using** declaration, the non-**using** declaration takes precedence over the **using** declaration.

```
extern int invert(Colors::color);
using Colors::invert;   // invert is now overloaded.
Colors::color C = ...;
invert(C);              // 2: Selects from 2 inverts.
```

Here, at **2:** the call is made to the **extern int** function rather than to the function declared with the **using** declaration. No diagnostic is produced, and this silent choice could easily become a point of disagreement among language designers.

Where to place using declarations

You can place a **using** declaration anywhere any other declaration can appear. A **using** declaration within a namespace *N*, just like any other declaration in *N*, causes the name of the identifier being declared to be declared in *N* as well, so *N***::***name* can refer to the **using**-declared name. However, the imported name is still "owned" by its original namespace, so being **using**-declared does not affect its external mangled name. For example:

```
namespace TV {
   using Colors::color;
   color pixels[1024*1024];
   }
```

You can now write `TV::color` or `using TV::color`, even though `color` is owned by namespace `Colors`.

Partly because **using** declarations allow a name to be declared in multiple namespaces, and partly because **using** declarations are just aliases, distinct **using** declarations can import the same declaration.

Such an otherwise invalid duplicate declaration is allowed with **using**, and has the same effect as if the duplicate were not there.

```
using Colors::color;
using TV:color;         // OK; same as Colors::color
```

This duplicate declaration is more useful with functions, where you can use a namespace solely for constructing a collection of functions:

```
namespace Collection1 {
   using A::f, B::f, C::f;
   };
namespace Collection2 {
   using B::f, C::f, D::f;
   };
...
... Collection1::f(...) ...
... Collection2::f(...) ...
```

Now you can also import both collections without fear of duplication:

```
using Collection1::f, Collection2::f;
... f(...) ...  // Selects from A::f, B::f, C::f, and D::f.
```

A **using** declaration does *not* allow you to select a single, individual function:

```
using N::f(int);      // NOT ALLOWED
using N::f(char);     // NOT ALLOWED
```

MetaWare C/C++ also supports **using** *N***::***, which effectively performs a **using** on every identifier declared in *N*.

The using directive

A **using** directive is quite different from a **using** declaration. The directive imports an entire namespace at once, but not as declarations in the scope containing the directive. Instead, within the scope containing the directive, the names are treated as if they were declared in the smallest enclosing non-namespace scope. (This is the global scope for namespaces declared globally or within global namespaces.) Furthermore, apparent duplicate declarations resulting from such treatment do not occur until the point of use of a name.

A **using** directive begins with **using namespace**, followed by the (potentially qualified) name denoting the namespace.

```
func() {
   using namespace Colors;
   color C = red;
   invert(C);
   }
```

Because the names are treated as if they were declared globally, local declarations of the same name override global declarations:

```
func() {
   using namespace Colors;
   using Other_namespace::invert; // 1: Local
                                  //  declaration
   color C = red;
   invert(C);                     // 2: Calls 1:
   }
```

The local declaration at **1:**. takes precedence over the global declaration, so the function call at **2:**. invokes the function declared at **1:**. However, consider the following situation:

```
func() {
   using namespace Colors;  // 1: invert declared
                            //  globally
   using namespace Other_namespace; // 2: Another
                                    //  global invert
   color C = red;
   invert(C);               // 3: Chooses between
                                  invert at 1: and
                                  invert at 2:
   }
```

Here, the members of both namespaces are introduced globally; therefore, at **3:** there is the choice of two globally declared **invert()** functions.

As with **using** declarations, ambiguity between two functions with identical argument types is resolved in favor of a function not introduced by a **using** definition. Likewise, ambiguity that cannot be resolved by overload resolution is reported at the use of the name, not at the occurrence of the **using** directive.

Names introduced by a **using** directive are ignored when an expression uses explicit qualification. Thus, even though the names are treated as if they were declared globally, a reference `::global_name` does not refer to any names introduced by a **using** directive:

```
namespace A { int glob; }
using namespace A;
int glob;     // Two globs declared globally
func() {
   glob++;    // Ambiguous:  A::glob or ::glob?
   ::glob++;  // Refers to global, non-namespace glob
   A::glob++; // Refers to namespace A's glob
   }
```

Namespaces, friends, and external declarations

friends first declared in a class are declared in the smallest enclosing non-class, non-function-prototype scope. With the addition of namespaces, this means a **friend** declaration can be "injected" into the namespace:

```
namespace A {
   class C {
      friend func();  // This is A::func.
      };
   }
int A::func() { ... }
```

The same holds for an **extern** declaration first declared within a function in a namespace. Rather than being a global **extern**, the declaration is owned by the namespace.

Unnamed namespaces

If you omit the identifier in a namespace definition, you are referring to an "unnamed" namespace. Each compilation unit has one of these unnamed name spaces, which is unique for that compilation unit, and a **using** directive for it is automatically assumed. The effect is that you can enclose your local data in the unnamed namespace without fear of it clashing with local data in other compilation units.

For example:

```
namespace {
   int a, b, c;
   void f() { ... }
   }
```

This is equivalent to the following:

```
namespace UNIQUE {
   int a, b, c;
   void f() { ... }
   }; using namespace UNIQUE;
```

where `UNIQUE` is a compiler-chosen name unique for each compilation unit.

Status of namespaces

In general, the MetaWare C++ implementation of namespaces follows the preliminary proposal for namespaces, 93-055.

The MetaWare C++ implementation does not regard classes as namespaces, and does not use internal linkage for names in unnamed namespaces. It is not necessary to preserve clashes among different compilation units because unnamed namespaces are unique, so none of the contained identifiers can ever clash when a program is linked.

The MetaWare C++ implementation does not allow templates to be namespace members.

Chapter 13 — Exceptions in MetaWare C++

In This Chapter

Detecting errors

When a function **f()** detects an error, a typical coding practice is to have **f()** either return a null value, assuming such a value exists, or set a global value (such as `errno`) indicating a problem. Then any client of **f()** must check the return value or global value to see if an error occurred. Furthermore, the client might have to transmit the error code to the function that called it, which then transmits the error code to the function that called it, and so forth. Every function in the calling chain must inspect an error code or global value until some function **g()** is reached that can deal with the error situation and dismiss it.

Such programming practice is complicated and prone to error.

How exception handling works

C++ exceptions are designed to handle errors in such a way that **f()** can *throw* an exception when an error occurs, and **g()** can *catch* the exception and deal with the error. None of the functions in the calling chain between **f()** and **g()** need to do anything about the error situation; the communication is directly between the source of the error and the function that deals with it.

Throwing, trying, and catching

When a function detects an error, it can *throw* an exception. In C++, the code for this is as follows:

```
throw expression
```

This throw should occur within a block of code (called a *try block*) constructed to test and catch the exception.

The value of *expression* is communicated to the *catch* point, so you can place information in the expression that tells the catch point what happened. **throw** *expression* is an expression of type **void**, and has the same precedence as the assignment operator '='. For example:

```
struct File_error {
   const char *file_name;
   File_error(const char* name) {
      file_name = name;
      }
   };
...
void open(const char *name) {
   if (attempt_to_open_name_fails)
      // Communicate name to catch point by constructing and throwing
      // a File_error object containing name
      throw File_error(name);
   }
```

To catch an exception, you must embed code that can potentially cause the exception in a try block. The try block is followed by a sequence of one or more *handlers* that specify what kinds of exceptions can be caught, catch them, and perform whatever actions are needed to deal with the error:

```
try {
   // Code that can throw an exception or calls code that can do so
   open(some_file);
   }
catch (File_error &F) {
   printf("File error in file %s\n",
           F.file_name);      // Catch File_errors
```

```
   }
catch (Floating_overflow &O) {
   // Catch floating-point overflows
   }
```

Formally, a try block has this syntax:

```
try compound-statement
```

followed by one or more handlers, each of this syntax:

```
catch (exception-declaration) compound-statement
```

Each handler takes a single argument, because the *exception-declaration* is just like an *argument-declaration*, except that you cannot specify a storage class or a default value.

The type of a handler's exception declaration specifies what sort of exception it can catch. Basically, it can catch anything thrown that is of the same type; or if the handler specifies a base type or pointer to a base type, the thrown type can be a class or a pointer to a class derived from the base. This is just the standard C++ conversion from derived to base. See What a handler can catch on page 173 for the complete list of rules.

What happens when an exception is thrown

When an exception is thrown, the exception-handling run time looks at the handlers in the order they are listed for each try block that has been entered but not exited, from most recently entered to first entered. Control is transferred to the first handler whose type allows it to catch the exception. The search continues until such a handler is found, to the end of the program if necessary. Once an appropriate handler is found, the search stops; for each exception thrown, there can be no more than one catch. When control enters the handler, the exception becomes active and remains active until handling is complete and control exits the handler. For example:

```
void throw_something() { throw 1; };   // 1:
void func3() {
   try {                               // 2:
      throw_something();
      }
   catch(char *p)  { /* ... */ }       // 2a:
   catch(int *p)   { /* ... */ }       // 2b:
   }
void func2() {
    try {                              // 3:
       func3();
       }
   catch(float f) { /* ... */ }        // 3a:
   catch(int i)   { /* ... */ }        // 3b:
   }
void func1() {
   try {                               // 4:
      func2();
      }
   catch(double d) { /* ... */ }       // 4a:
   }
main () { func1(); }
```

In this example, when `1` is thrown at `1:`, the exception-handling run time has available to it the handlers in the three active try blocks at `2:`, `3:`, and `4:` in that order. It checks the handlers at `2a:`, `2b:`, `3a:`, and `3b:`. Handler `4a:` is not considered; searching stops at `3b:` because `3b:` catches the integer thrown.

throw *expression* initializes a temporary object of the static type of the expression (**int** in the preceding example). This copy is used to initialize the appropriately typed handler parameter; the mechanism is similar to function-argument passing. The initialization can require another copy, called the *parameter copy*, if the handler parameter is not the same type as that of the first copy made. The exception-handling run time is allowed to optimize away copies if doing so does not change the program's behavior (except for using copy constructors to copy the object).

After a handler completes, program control goes to the statement following the sequence of handlers.

NOTE The compiler does not inline functions containing a **throw**. It places copies of all inline functions containing a **throw** in any module that calls the **inline** function. If you want inlining to occur, you must place the **throw** in a static member function and call this static member function from the **inline** function.

Unwinding: automatic clean-up with destructors

C++ automatically causes destructors to be called to perform clean-up on automatic objects; you do not need to write specific handlers for this purpose. This clean-up of objects is called *unwinding*.

As control passes from a **throw** to a handler, destructors are invoked for all automatic objects constructed since the handler's **try** block was entered. Any object or array of objects that is partially constructed has destructors executed only for its fully constructed subobjects. So, if a constructor for an object allocates a resource, the destructor can free the resource as exception handling *passes through* the site of the object's declaration:

```
void f() {
   try {
      C x;
      throw 77;
      }
   catch(int) { ... }
   }
```

In this example, the destructor (if any) for object `x` is run before control enters the handler.

A constructor for an object is considered to have fully executed when all user-written code in the constructor has run. A destructor for an object is considered to have run as soon as control enters the destructor.

The definition for the constructor makes clear sense, but the destructor requires a bit more analysis. If you are in the middle of user-written code in a destructor, and an exception is thrown, you should not re-enter that same destructor for the same object during an unwind; half the destructor code has executed, so the object is in an indeterminate state. Hence the definition: as soon as user code in a destructor starts to execute, the object is considered completely destroyed for purposes of unwinding.

Additional clean-up during unwinding

Although the functions between the **try** and the **catch** do not need to do anything about the error situation, you might want a particular function to do clean-up if an exception *passes through* it. One way to do this is to write, in that function, a try block that catches any exception, code to perform the clean-up, and a **throw** command to *rethrow* the exception once the clean-up is finished. This way, you

can perform your own clean-up in addition to the automatic clean-up provided by the unwinding process. For example, in the program below, the **malloc**-ed space is not reclaimed after the throw:

```
void computation() {
   if(some_error)
      throw 1;
   }
void func2() {
   void *p = malloc(a_lot);
   // Does something with p
   computation();
   }
void func1() {
   try { func2(); }
   catch(int i) { /* ... */ }
   }
```

The thrown integer is caught in **func1()**, and the pointer allocated in **func2()** is not deallocated.

You can manually deallocate the pointer by writing a catch clause that catches any exception, performs your own clean-up, and then rethrows the exception to be caught by the handler for which it was originally intended. In the following example, **func2()** does this:

```
void func2() {
   void *p = malloc(a_lot);
   try {
      // Does something with p
      computation();
      }
   catch(...) {         // Catches anything

      // Clean-up code, such as freeing storage
      free(p);          // Frees memory in the pointer
      throw;            // Rethrows current exception
      }
   }
```

Here, the handler in **func2()** catches the integer thrown in **computation()**, does its own clean-up, freeing the allocated pointer, and then rethrows the integer, which is caught by the original intended handler located in **func1()**.

In general, **throw** without an expression rethrows the currently active exception. The *exception-declaration* "**...**" can catch anything. Thus, no matter what is thrown, the combination **catch(...)** and **throw;** catches the exception, passes control to whatever clean-up code you specify, and then rethrows the exception to be caught by some other handler. If a parameter copy was made, a rethrow causes destruction of the copy. The parameter of any subsequent handler is initialized from the original copy, or from a parameter copy generated to match that parameter's type.

Because the exception handling makes copies, you cannot count on changes you make to a handler argument being reflected in a rethrown value caught by a subsequent handler.

Stacked and nested exceptions

An *active* exception is one for which handling is not complete. The *current* exception is the one being handled.

Active exceptions conform to a stack discipline. When a handler is entered, the current exception, if any, is stacked and the exception caught by the handler becomes the current one. When a handler finishes, the exception on top of the stack, if any, is popped and becomes the current one once again.

Therefore you can dynamically nest exception handling, including rethrows:

```
void f() {
   try {
      throw 77;          // 1: Throws int exception
      }
   catch(int i) {
      try {
         throw "string";// 2: Throws string exception
         }
      catch(char *p) {
         ...;            // 3: Catches string exception
         }
      throw;             // 4: Rethrows int exception
      }
   }
```

Here, the **throw** at `4`**:** rethrows the **int** value **77**, which is popped off the stack and becomes the current exception as soon as control leaves the handler at **3:**. If you had a **throw** at `3`**:**, it would have rethrown the string value **string**.

If you rethrow when there is no active handler, the exception-handling run time reports an error.

Exceptions during unwinding

Before control enters a handler `H`, destructors called during the unwind process can throw exceptions. This is valid as long as the exception is also caught before `H` is entered. If an exception is thrown during destruction that attempts to enter or skip past `H`, the run-time function **terminate()** is called:

```
struct C { ~C() { throw “skip”; } };   // 3:
void f() {
   try {
      try {
         C x;
         throw 1;                      // 1:
         }
      catch(int i) {                   // 2:
         }
      }
   catch(const char *) {               // 4:
      }
   }
```

Here, **1:** throws an integer, and in searching for a handler, the exception-handling run time finds **2:**. Before transferring to **2:**, object **x** is destroyed. However, at **3:** the destructor throws a string. Rather than have the string caught at **4:**, the exception-handling run time detects an attempt to skip over **2:**, and calls **terminate()**.

terminate() is also called if no handler is found for a thrown exception or when the exception mechanism finds the stack corrupted.

The default behavior of **terminate()** is to call **abort()**. You can change the function that **terminate()** calls by calling **set_terminate()** with a function pointer; then **terminate()** calls the function pointed to instead of **abort()**.

set_terminate() returns a pointer to the function **terminate()** would have called, and records your function as the one to call. In this way, you can program a chain of functions to be called when

terminate() is invoked. It is an error if the function you supply returns to **terminate()** or throws an exception.

Partially constructed and destroyed heap objects

An issue not discussed in the ANSI X3J16 draft on exception handling is the treatment of partially constructed and partially destroyed heap objects. The sentiment is that partially constructed heap objects should be destroyed; therefore, if a **new** operation is not complete, the destructors for objects constructed so far are run as part of exception unwinding:

```
int cnt = 0;
struct S {
   S() { if (++cnt == 3) throw 1; }
   ~S();
   };
void f() {
   try {
      S *a = new S[3];
      }
   catch(int i) {
      // ...
      }
   }
```

Here the constructor for **S** throws an exception after two objects have been constructed. The third element of the array is not constructed, so the **new** operation is unfinished, and exception handling runs the destructors on `a[0]` and `a[1]` before entering the handler. It also frees the storage allocated by **new**.

A somewhat more controversial decision that remains to be resolved is what to do about partially destroyed heap objects. We expect the consensus to be "finish the destruction", which is what MetaWare C++ has implemented. A delete operation that has started but has not finished is completed as part of unwinding:

```
int cnt = 0;
struct S {
   S() { }
   ~S() { if (++cnt == 2) throw 1; }
   };
void f() {
   try {
      S *a = new S[3];        // This completes
      delete p;               // This throws
      }
      // Deletion completed as part of unwinding
      //  before control enters catch(int i)
   catch(int i) {
      // ...
      }
   }
```

The destructor throws an exception after `a[2]` and `a[1]` are destroyed; only `a[0]` remains. The exception-handling run time calls the destructor to destroy `a[0]` before entering the handler.

NOTE There is an apparent inconsistency in the treatment of heap-based objects. Completely constructed heap objects are *not* automatically destroyed as the exception-handling run time unwinds, but partially constructed and partially destroyed objects are, because it is difficult for a user to program the destruction of objects that are in a partially constructed state.

Objects constructed via a completed **new** operation are not destroyed because it is impossible for the exception-handling run time to know what completely constructed heap objects should be destroyed; the pointers to such objects could have been copied anywhere.

Function exception specifications

In a function declaration, you can list the set of exceptions that a function might directly or indirectly throw. The list is known as an *exception specification*, and takes this form:

```
throw ( argument_decl_list )
```

The exception specification can appear in the same place as a function declaration's optional *cv_qualifier_list*.

The list of argument types in an exception specification must have no default values and cannot use "**...**". The exception specification is checked at run time only after a handler has been found for a thrown exception. If the function directly or indirectly throws an exception that could not be caught by a handler of the form **catch(***T***)**, where *T* is one of the exception-specification types, a function named **unexpected()** is called. For example:

```
void f() throw(A*, int) {
   // ...
   }
```

is effectively the same as:

```
void f() {
   try {
      // ...
      }
   catch (A*) { throw; }
   catch (int) { throw; }
   catch (...) { unexpected(); }
   }
```

The default behavior of **unexpected()** is to call **terminate()**. You can program the behavior of **unexpected()** by calling **set_unexpected()** and passing it a function pointer. **set_unexpected()** returns the pointer that **unexpected()** would have called, and records your function as the one to call. In this way you can program a chain of functions to be called when **unexpected()** is invoked.

Unlike **terminate()**, **unexpected()** can throw an exception. Handlers for this exception are searched for starting at the call of the function whose exception specification was violated. Thus an exception specification does not guarantee that only the listed types will be thrown. After the call to **unexpected()**, the function behaves as if it had no exception specification.

Miscellaneous rules for handling exceptions

The following rules should help you write effective exception-handling code:

- In a sequence of handlers, it is an error for handler *H1* to precede handler *H2*, if *H1* always catches an exception that *H2* can also catch. This erroneous situation ensures that *H2* will never run:

```
struct base { ... };
struct derived: base { ... };
try {
   // ...
   }
catch(base *bp) { }
catch(derived *dp) { }  // Error: Control can never enter this handler.
```

- A throw can appear as conditional operands of **?:**; the resulting type of the **?:** operator is the type of the other conditional operand:

```
x ? throw something : 1  // Type is int.
```

- You cannot jump into a try block or a handler, but you can jump out of either.
- Exceptions are not handled for globally declared objects.

What a handler can catch

Here is the full list of rules for what a **catch** clause can catch. A handler can catch a thrown object under the following circumstances:

- The thrown object can be copied at the throw point (for classes, this means that a default copy constructor is accessible; for non-classes, this requirement is irrelevant).
- A handler whose type is *T***&** can catch the same things that a handler of type *T* can catch.
- A handler whose unmodified type is *T* can catch a thrown object whose unmodified type is *E* if any one of the following is true:
  - *T* and *E* are the same type.
  - *T* is an accessible base class of *E* at the throw point.
  - *T* is a pointer type and *E* is a pointer type that can be converted to *T* by a standard pointer conversion.
- A handler whose *exception-declaration* is "**...**" can catch anything.

These rules are slightly different from, and less limiting than, those in the current ANSI X3J16 draft. Most of the X3J16 rules will undergo change.

Using exception handling

This section describes how to compile classes that are exception-aware, and how to use them with existing classes that are not exception-aware.

NOTE MetaWare C/C++ embedded development distributions do not include exception-aware run-time libraries, because it is unlikely that developers of embedded applications will want to use them.

To obtain exception-aware libraries, you must rebuild the supplied MetaWare C/C++ run-time library source code with option **-Hexcept** specified. You can then explicitly link your application with those libraries.

Generating exception-aware classes

An important part of exception handling is the clean-up of partially constructed or destructed objects. To properly achieve this clean-up in the MetaWare C++ implementation of exception handling, constructors increment a global "subobject count" by one and destructors decrement the count by one. For example, after construction of an object with a total of three subobjects, the subobject count is incremented by three.

A class is *exception-aware* if its constructors increment the global subobject count and the destructor decrements the count.

So that code with exception-handling capability can properly interoperate with earlier code, the MetaWare C++ compiler provides a way to declare exactly which classes are exception-aware (see Making a class be exception-aware on page 174).

Making a class be exception-aware

By default, classes are not exception-aware. You make a class be exception aware by turning on toggle `Exception_aware_class` before the class declaration. Within a class hierarchy, exception-aware classes can be intermixed with non-exception-aware classes.

You should make a class be exception-aware if either of the following is true:

- You do not want the destructor to run on an unconstructed object; for example, if the destructor accesses a pointer allocated during construction.
- You want the destructor always to run on a constructed object; for example, if the destructor frees storage allocated during construction.

Clean-up of completely constructed objects

For a completely constructed object, no portion of which is exception-aware, no destructors are run during clean-up. This is an advantage, because it saves time during clean-up.

For a completely constructed object, a portion of which is exception-aware, all destructors are run during clean-up.

Clean-up of partially constructed objects

For partially constructed objects, the presence of non-exception-aware classes introduces imprecision in the clean-up.

A partially constructed object is considered to be fully constructed if all its exception-aware subobjects have been constructed. All destructors are run, even though the object is partially constructed.

For example, if all but the most-derived constructor completes, and the most-derived class is not exception-aware, the object appears fully constructed.

```
#pragma On(Exception_aware_class)
struct A { ~A(); };
struct B { ~B(); };
#pragma Pop(Exception_aware_class)
struct C:A, B { ~C(); C() { throw 1; } };
C x;
```

Even though **C**'s constructor throws the exception, **x** is taken as fully constructed and the **C** subobject is fully destructed.

If you are concerned about a destructor running on a non-exception-aware class, be sure that the most-derived class is exception-aware. In that way, you ensure that if all the exception-aware objects are constructed, the object is fully constructed.

For a partially constructed object with some exception-aware subobjects not constructed, a non-exception-aware subobject is not destructed unless its construction was followed by the successful construction of an exception-aware subobject.

For example:

```
#pragma On(Exception_aware_class)
struct A { ~A(); };
#pragma Pop(Exception_aware_class)
struct B { ~B(); };
struct C { A a1; B b; A a2; };
C x;
```

If the construction of the second `A`-type class, `a2`, throws an exception, the `b` subobject is not destructed, even though it was fully constructed.

Interacting with previously compiled classes

Because exception handling is new, there exist classes (such as those found in third-party libraries) that are not exception-aware. This is why none of the standard classes in the MetaWare C++ run-time library (such as **iostreams**) are exception-aware by default. To make them exception-aware would introduce problems of binary incompatibility with any third-party libraries that might make use of those classes.

Option **-Hexcept** makes it safer to use toggle `Exception_aware_class`, and also enables you to choose between the exception-aware version of the run-time library and the version that is not exception-aware. Specifying option **-Hexcept** turns on toggle `Exception_aware_class`. See the *MetaWare C/C++ Programmer's Guide* for more information.

Wrapping header files

Header files for third-party libraries that do not contain exception-aware classes must not be compiled with toggle `Exception_aware_class` turned on (whether or not you use option **-Hexcept**).

You must protect these header files by "wrapping" them with other include files that turn off and on toggle `Exception_aware_class`. To see how this works, specify **-Hon**=*`List`* when you compile.

To direct the compiler to wrap an include file, insert a character string *xxxx* in the file's name, then specify option **-Hnonexcept_wrap**=*xxxx* when you compile.

For example, given the following command line, any include file with the string "`nonexc`" in its name is wrapped with **#pragma** `Off(`*`Exception_aware_class`*`)` at the beginning and **#pragma** `Pop(`*`Exception_aware_class`*`)` at the end:

driver_command `fnc.cpp -Hexcept -Hnonexcept_wrap=nonexc`

Where *driver_command* is the driver command for your target processor.

Option **-Hnonexcept_wrap** behaves just like option **-Hc_wrap**, except that option **-Hnonexcept_wrap** uses wrapper files called `exc_off.h` and `exc_pop.h`.

Chapter 14 — Special cast notations

In This Chapter

Introduction

The following four special forms of cast notation were added to the C++ language to express the safe casting of one type to another:

```
dynamic_cast<Type>
static_cast<Type>
reinterpret_cast<Type>
const_cast<Type>
```

In the MetaWare C++ implementation, these are four predefined templates:

```
template <class Type> class dynamic_cast { ... };
template <class Type> class static_cast { ... };
template <class Type> class reinterpret_cast { ... };
template <class Type> class const_cast { ... };
```

Types that are instantiations of any of these four templates behave in a special manner when they are used in a cast. **Q<***Type***>(***expr***)**, where **Q** is one of the four special MetaWare C++ templates, actually casts *expr* to type *Type*, subject to conditions described in the following sections. For example:

```
static_cast<Type>(expr)
```

casts *expr* to *Type*. So does this:

```
typedef static_cast<Type> sc_Type;
... = sc_Type(expr);
```

Each of the last three templates expresses approximately a subset of the full potential of a C++ cast. **dynamic_cast<***Type***>** gives the ability to safely cast from, say, a pointer-to-base to a pointer-to-derived, even when the base is virtual. This ability has not previously been present in C++.

static_cast

This is a reasonably safe subset of **(***Type***)(***expr***)**. The result is of type *Type*.

If *expr* is of type *Te*, the cast of *expr* to type *Type* is valid if it falls into any one of the following cases:

- *Te* => *Type* is a standard conversion.
- *Type* => *Te* is a standard pointer, reference, or pointer-to-member conversion (this covers contravariant conversions); *Te* is not a virtual base of *Type*, and *Te* => *Type* does not cast away **const** (see const_cast and casting away const on page 179).

> **NOTE** A *contravariant conversion* is a conversion from pointer-to-base to pointer-to-derived, or from reference-to-base to reference-to-derived.

- *Type* is **void** or a class type, and **(***Type***)(***expr***)** is valid.

Semantics of the conversion are the same as for **(***Type***)(***expr***)**.

Thus, conversions from pointer-to-non-virtual-base to pointer-to-derived are permitted, but dangerous conversions from type **int *** to **char *** or from pointers to two unrelated class lattices are not.

To safely convert a virtual base to a derived class, use **dynamic_cast**; to convert dangerously, use **reinterpret_cast**.

const_cast and casting away const

Casting away **const** means converting a pointer-to-**const** to a pointer-to-non-**const**. To cast away **const**, use **const_cast**.

Casting away **const** also incorporates casting away **volatile**, and handles multi-level pointer conversion. It is defined as follows: The conversion

```
Type cv * => Type' cv' *
```

where `cv` is one of the four sets:

```
{const}, {volatile}, {const, volatile}, {}
```

is said to cast away **const** if one of the following is true:

- `cv` is not a subset of `cv'`.
- *Type* and *Type*' are pointer types, and *Type* => *Type*' casts away **const**.

For example:

```
const int * => int *
```

and

```
const int * * * * => int * * * *
```

both cast away **const**. However:

```
const int * * => int *
```

does not cast away **const**, because there is an implicit conversion of **const int *** to **int**, which is not subject to the rules for casting away **const**. This last case is irrelevant to **static_cast** because you can never have an unequal number of *s; however, it is relevant to **reinterpret_cast**, which also cannot cast away **const**.

dynamic_cast

dynamic_cast<*Type***>(***expr***)** allows safe conversion from, for example, a pointer-to-base to a pointer-to-derived. Run-time checks are used to be sure the conversion is completely valid. To do the run-time checks, run-time type information must be available for both the source and destination types, or the cast might fail. A failure of the cast either returns **0** (zero) or throws an exception, depending on the nature of the failure.

The result of the cast is type *Type*. *Type* must be one of the following:

- a pointer to a fully defined class
- a reference to a fully defined class
- **void** *

Let *Te* be the type of *expr*. Then, if the following are *all* true:

- *Type* is a pointer to a base class `B`.
- *Te* is a pointer to a derived class `D`.
- `B` is an accessible base class of `D`.

the result is the same as `(D*)B`. The same is true if *Type* and *Te* are reference types that are similarly related.

Otherwise, *Te* must be a polymorphic type. That is, it must have virtual functions, so that *expr*'s true run-time type can be accessed from the virtual function table.

If *Type* = **void** `*`, the result is the pointer to the complete object pointed to by *expr*.

Otherwise, a run-time check is applied to see if *expr* can be converted to type *Type*. Basically, *expr* is first converted into a pointer to the complete object; then this pointer is converted to *Type*. The first conversion fails if *expr* does not point to an unambiguous base class of its complete type. If either conversion cannot be done, the result is **0** (zero). If *Type* is a reference type, and the cast fails, exception **Bad_cast** is thrown. (See Exceptions in MetaWare C++ on page 165 for general information on exception handling.)

reinterpret_cast

A **reinterpret_cast** allows "raw" conversion from:

- any pointer type to any other pointer type that is big enough
- any pointer type to any integer type that is big enough
- any integer type to any pointer type
- any object `E` to any reference type *Type*, provided that **reinterpret_cast** can convert `&E` to *Type*`*` by a conversion in one of the preceding three bullets.

No class hierarchy navigation is done; basically, when the type sizes are the same, the cast value is unchanged. When the sizes of the types are different, the compiler warns. For conversion to a larger type, the semantics are equivalent to ***((***Type***)&***expr***)**. This implies that the result contains some indeterminate value.

Migrating to the special cast notations

The proposal of the X3J16/96-0018 ANSI C++ Draft Standard for the special, newer cast-notation syntax suggests the following:

> When moving from the old-style casts to the new, replace all old-style casts with the new notation **static_cast**, and see what diagnostics you get when you recompile.

To facilitate this replacement process, the MetaWare C/C++ compiler provides the toggle `Try_static_cast` (default off). The effect of this toggle is to treat all C++ casts of the form **(***Type***)***expr* as **static_cast**, and warn at each line containing a cast (so you can find them all). An error might also be generated if the **static_cast** is not valid.

Chapter 15 — C++ type information

In This Chapter

Introduction

C++ allows implicit conversions from a derived class to a base class. That is, an object of a derived class contains a subobject of the type of one of the bases. For example:

```
struct D : B { ... };
void f(B* bp) { ... }
...
D *dp = new D;
f(dp);  // Automatic conversion to pointer-to-base
```

The conversion in the opposite direction is not automatic, because a pointer to a base object (say of class `B`) is not necessarily a pointer to a larger object of derived class `D`. You might just have a `B`, not a `B` within a `D`. Therefore C++ requires casting to convert in the opposite direction. With a cast, you assert to the compiler that a pointer is in fact a pointer to a `D` object:

```
B *bp = dp;       // Points to base
D *dp2 = (D *)bp; // Converts back; points to D object
```

This process works only if the base is not virtual. If the base is virtual, the compiler cannot generate code to make the conversion to `D*`, and the cast elicits an error message.

These problems are solved with run-time type information (RTTI).

Run-time type information (RTTI) and type identification

For subobjects of polymorphic classes (those that have virtual functions), with RTTI you can find out the complete type of a subobject, the ultimate derived class. For example, you can write the following code:

```
B *bp = new D;
```

Keyword typeid

Although the static type of `bp` is pointer to `B`, the complete type of `bp` is pointer to `D`, because `bp` points to a subportion of a `D` object. You can use the keyword **typeid** to obtain a representation of type `D`:

```
typeid(*bp);
```

The representation takes the form of another object corresponding to type `D`.

Other benefits of RTTI include being able to do the following:

- print out the name of the complete type of an object
- find out the size of an instance of the complete type
- inquire as to the base classes

For example:

```
printf("bp really points to a %s",typeid(*bp).name());
```

prints:

```
bp really points to a D
```

The type object

typeid(*expression***)** or **typeid(***type***)** returns a reference to an object of type **const** `Type_info`, or of some class derived from `Type_info` (**class** `Type_info` is defined in `typeinfo.h`). This object is called a *type object*.

For **typeid(***type***)** (similar to **sizeof(***type***)**), the type object describes exactly the type given.

Table 8 Objects returned by typeid expressions

Expression	Returns a reference to a Type_info object
typeid(int)	Describing type **int**
typeid(int *)	Describing pointer to **int**
typeid(void (*)(int))	Describing pointer to a function taking **int** and returning **void**
typeid(`D`**)**	Describing class `D`

For **typeid(***type***)**, the type object is statically known by the compiler; you specify precisely the type. The same holds for an expression of anything other than a polymorphic class: the type is what is statically known by the compiler, as in the following examples.

```
int i;
typeid(i);      // Same as typeid(int)
int *p;
typeid(p);      // Same as typeid(int *)
B *bp;
typeid(bp);     // Same as typeid(B *)
struct s {};    // Not polymorphic
struct d:s{};   // Derived from s
s *sp = new d;  // Cast from derived to base
typeid(*sp);    // Same as typeid(s), because s
                //  isn't polymorphic
```

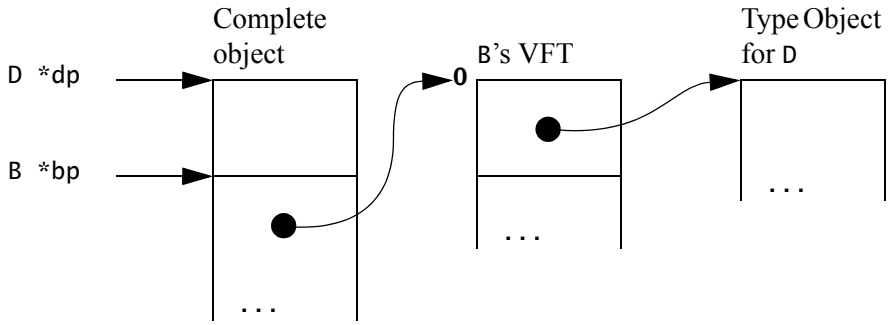
Polymorphic type identification

However, when the argument is an object of a polymorphic class, the compiler generates code to fetch the true complete type:

```
typeid(*bp);    // Produces typeid(D)
```

We call this a *polymorphic* **typeid**. The code fetches a pointer to the type object describing the complete type from position `0` of the polymorphic class's virtual function table. That is why the class must be polymorphic, as is shown in Figure 3 A Polymorphic typeid.

Figure 3 A Polymorphic typeid



MetaWare C++ reserves location 0 (zero) of the virtual function table for this purpose.

The single-copy heuristic

Type objects take up space, and there might be duplicates of them. For example, each compilation unit containing **typeid(int)** has a distinct type object describing the type **int**. Duplicates can be avoided for classes. A class has just one copy of its type object if it has a non-pure-virtual, non-inline member function. The type object is placed in the module containing the definition of this function (we say the class satisfies the *single-copy heuristic*).

The single-copy heuristic is also used by the compiler to determine the location of the unique copy of a class's virtual function table; but the heuristic is used for type objects whether or not the class has virtual functions. For example:

```
struct s {
   int f();       // Where s::f is, so is typeid(s)
   int g();
   };
struct t { };     // Every occurrence of typeid(t) produces a distinct copy.
```

Using the type object

You can use the type object for other things than printing out the name of a type. For example, suppose you have a pointer to a base object, and you want to call a function that is defined only on the complete object:

```
((D*)bp)->dfunc();
```

You want to make this cast secure, in case the base is not part of a D. One way you can do this is by checking for **typeid** equality:

```
if (typeid(*bp) == typeid(D))
   ((D*)bp)->dfunc();
```

To use ==, include typeinfo.h, the header file in which Type_info is defined.

However, checking **typeid** equality is not the best way to make sure your base object is part of D, because in fact *bp might be part of an object that is even larger than a D; that is, it might be of a type derived from D. The previous example works if the complete type is exactly a D.

dynamic_cast

To determine if *bp belongs to D or to a class derived from D, you use **dynamic_cast** (see dynamic_cast on page 179):

```
D *try_dp = dynamic_cast<D *>(bp);
```

dynamic_cast fetches the type object from *bp's virtual-function table, then does the following:

- If *bp's complete type is D, **dynamic_cast** simply makes sure that B is an unambiguous base of D, and then does the conversion from the base to D.
- If *bp's complete type is E, derived from D, **dynamic_cast** converts B to E unambiguously, then converts E back to base D.

In both cases, the pointer returned is 0 (zero) if *bp is not part of a D, or the pointer to the D object if *bp is part of a D or class derived from D.

This means you can write the following:

```
D *try_dp = dynamic_cast<D *>(bp);
if (try_dp != 0)
   try_dp->dfunc();
```

```
else
   printf("Sorry, bp doesn't point to a "
          "piece of a D.");
```

dynamic_cast is described in detail in dynamic_cast on page 179.

Explicit initialization in conditions

With the introduction of RTTI, C++ was also extended to allow declarations with explicit "=" initialization in conditions. For example:

```
if (D *try_dp = dynamic_cast<D *>(bp))
   try_dp->dfunc();
else
   printf("Sorry, bp doesn't point to a "
          "piece of a D.");
```

Unfortunately, this added syntax conflicts with previously correct programs. However, conflicts are expected to be rare:

```
if (Type(X) = 3) ...
```

*Type***(X) = 3** could be a declaration of **X** of type *Type*, initialized to **3**. Or it could be the type cast of **X** to *Type* and a subsequent assignment to the result of the cast.

Because of the ambiguity, MetaWare C++ also provides syntax where the declaration can be followed by a "**;**" to specify a declaration unambiguously:

```
if (Type(X) = 3;) // Declaration, no question about that!
```

To sum up:

- Use **dynamic_cast** when you want to know if an object is part of a bigger type `D`, but you do not care whether the complete type is derived from `D`.
- Use equality of **typeid**s when you want to know if an object is exactly of a certain type.

Functions on type objects

The following functions, available on all type objects, are declared in `typeinfo.h`:

```
const char* name() const;   // Simple name
const char* full_name() const;
                            // Full name: w/mangling
int Type_info::operator ==(const Type_info&) const;
int Type_info::operator !=(const Type_info&) const;
int before(const Type_info&) const;
long size() const; // Size of an object of this type
```

These functions on type objects do the following:

name()	Returns the name of the type, suitable for printing.
full_name()	*MetaWare C/C++ extension*. Returns the fully mangled name of the type. Whereas **name()** might be the same for two distinct types (say, a global class *X* and a class *X* nested within another class *Y*), **full_name()** is guaranteed to be distinct for distinct types.
== and !=	Compare for equality, by comparing the two **full_name()** functions; the comparison is not cheap, although a hash code makes the string compare unnecessary for most non-equal types.

before()	Sorts type objects; you can use it for implementing an efficient associative array indexed by type objects. **before()** lexicographically compares the **full_name()** functions of the two types to determine the order.
size()	*MetaWare C/C++ extension*. Returns the total number of bytes required to store an instance of the type.

Suppose you write a routine that takes an arbitrary `Type_info` object and prints its name:

```
void f(const Type_info& tid) {
   printf("type:%s ",tid.name());
   }
...
int a, *b;
f(typeid(b));
f(typeid(a));
```

Suppose you want to print more information if the type object turns out to be of a particular kind of type. For example, you might want to print the pointed-to type of a pointer type. `typeinfo.h` contains the definition of `Type_info` and of several derived classes. The derived classes are a MetaWare C++ extension, and they classify the kind of types a type object can represent. The derived classes are as follows:

```
class Modified_type_info: public Type_info
class Pointer_type_info: public Type_info
class Member_pointer_type_info: public Type_info
class Array_type_info: public Type_info
class Func_type_info: public Type_info
class Class_type_info: public Type_info
```

NOTE These classes answer the need of the exception-handling implementation for dynamic type information.

If a `Type_info` is actually a `Pointer_type_info`, the type is a pointer type, and you can discover the type it points to. You first determine that a `Type_info` is a pointer type by using **dynamic_cast** to determine the complete type of a particular `Type_info` (because `Type_info` is a polymorphic class, you can inquire as to the true type of the `Type_info` object):

```
void f(const Type_info& tid) {
   printf("type:%s ",tid.name());
   if (Pointer_type_info *PTI =
      dynamic_cast<Pointer_type_info*>(&tid))
      printf("pointed-to type is %s",
             PTI->base_type->name());
   }
```

NOTE Be sure to use **dynamic_cast** with pointer types, not reference types, because **dynamic_cast** throws an exception if a reference cast fails.

You can also find out whether the **typeid** of the `Type_info` is that of `Pointer_type_info`, and use an explicit cast:

```
void f(const Type_info& tid) {
   printf("type:%s ",tid.name());
   if (typeid(tid) == typeid(Pointer_type_info)) {
```

```
    Pointer_type_info *PTI =
       (Pointer_type_info*)(&tid);
    printf("pointed-to type is %s",
           PTI->base_type->name());
    }
}
```

This is generally much more efficient than using **dynamic_cast**.

Generating RTTI

For a non-class type, use **typeid** to generate run-time type information (RTTI).

For a class type, turn on toggle `Run_time_type_info` to ensure that RTTI is generated and stored in the class's virtual-function tables. (The default for toggle `Run_time_type_info` is on).

For such a class *D*, polymorphic **typeid** on a subobject of *D* return the type object for *D*, and **dynamic_cast** works for such subobjects. Also, classes satisfying the single-copy heuristic have just one copy of the RTTI. (For a discussion, see The single-copy problem on page 151.)

When toggle `Run_time_type_info` is off, RTTI is generated only as needed by **typeid**, so you cannot guarantee the presence of a pointer to the RTTI in the virtual-function tables. Thus, a polymorphic **typeid** can return a **NULL** reference.

In particular, a compilation unit that generates virtual-function tables for the class, but lacks a reference to **typeid**, also lacks a pointer to the RTTI in the virtual-function table.

When not to use toggle Run_time_type_info

It is best to leave toggle `Run_time_type_info` turned on for all your classes. Be aware, however, of two things:

- You should not turn on toggle `Run_time_type_info` globally for header files from third parties.

  Because RTTI is a new feature, third-party libraries might not contain RTTI. If you turn the toggle on around such headers, the compiler might generate a reference to the RTTI for a class satisfying the single-copy heuristic when it generates the RTTI for one of your classes, and your program link will fail.

- You might want to turn off toggle `Rubu_time_type_info` around classes with virtual functions that do not satisfy the single-copy heuristic.

  For such classes, the RTTI and virtual-function tables are generated for every compilation unit containing the definition of the class, even if the class is not used. For these classes, the compiler warns that each compilation unit will have a copy of the virtual-function tables.

Appendix A — Template instantiation issues

In This Appendix

NOTE If you are not using the ARC® International ELF linker, you must manually instantiate templates as described in this appendix, in order to avoid the single-copy problem. You must specify option **-Hnocomdat** when you use this manual solution.

The single-copy problem

There is a problem with using templates: you must ensure that, for each distinct set of parameters to a template, there is only a single copy of an instantiated function or static class data member. We call this the “single-copy problem.”

If you are using templates defined locally within a single source module, there is no single-copy problem and no special care is needed. Each source module has its own templates and instances, as shown here:

File `1.cpp`:

```
template <class T> T min(T x, T y) {...}
...
int i = min(1,2) + min(3,4);
```

File `2.cpp`:

```
template <class T> T max(T x, T y) {...}
...
int j = max(1,2) + max(3,4);
```

In this example, each module contains an instance of a template function: **min** for `1.cpp` and **max** for `2.cpp`.

The source of the single-copy problem

The single-copy problem arises only when you are using the same template in separate source modules, as shown here:

File `hdr.h`:

```
template < class T > T min(T x, T y) {...}
template < class T > T max(T x, T y) {...}
```

File `1.cpp`:

```
#include "hdr.h"
int i = min(1,2) + max(3,4);
```

File `2.cpp`:

```
#include "hdr.h"
int j = min(1,2) + max(3,4);
```

In this example, the object modules for `1.cpp` and `2.cpp` both contain definitions of **min(int,int)** and **max(int,int)**, and the linker will generate an error message about a duplicate definition.

Preventing duplicate function definitions

One way to prevent duplicate definitions is to turn off automatic template instantiations when compiling. You turn them off for template functions by turning off toggle `Auto_func_instantiation` as shown here:

File `1.cpp`:

```
#pragma Off(Auto_func_instantiation)
#include "hdr.h"
int i = min(1,2) + max(3,4);
```

File 2.cpp:

```
#pragma Off(Auto_func_instantiation)
#include "hdr.h"
int j = min(1,2) + max(3,4);
```

The Auto_func_instantiation toggle must be turned off in the definition of the function template — not at the reference to the function template. Essentially, the compiler records the setting of the toggle when it encounters the function template. For a non-inline function template encountered with the toggle off, there are no instantiations of the function template bodies in the source code.

However, you now have two modules, both of which make external references to **min(int,int)** and **max(int,int)**, and the linker will generate an error message about missing functions.

You can provide the necessary functions as follows. In a single new source module, include calls (with the appropriate arguments) to the function templates, and keep the Auto_func_instantiation toggle on. The object code for this new module contains the required instantiated functions.

We recommend you place these calls to the function templates in the module in such a way that the functions are never actually called at run time, and the code for calling them is never actually generated by the compiler. The following code shows a convenient way to do this with macros:

```
#include "hdr.h"
#define DONTCALL(f) (0 && f)
static void dummy() {
   DONTCALL(min(1,1));
   DONTCALL(max(1,1));
   };
```

Because of the conditional expression in the macro, the body for **dummy()** is empty, yet the object module for this source module contains the definitions of **min(int,int)** and **max(int,int)**.

Class templates and duplicate definitions

A class template must always be instantiated when it is used — because the resultant type is required for further processing — but the single-copy problem applies to the non-inline class member functions and static data members.

If you use a class template only locally to a source module there is no problem; the problem arises when you share an instantiation of a member function or data member.

As with function templates, the compiler by default always generates instances of class members when the class is used in a source module, and also when the definitions of those members are presented to the compiler. You can omit the definitions if you want.

You can turn off this automatic instantiation by turning off toggle Auto_class_member_instantiation. Then you must provide another source module (with the toggle turned on), in which you define the class members, as shown in Example 35 Turning off automatic instantiation.

Example 35 Turning off automatic instantiation

File stack.h:

```
template < class T > class stack {
   T *v, *p; int sz;
public:
   stack(int);
   ~stack() { delete[] v; }
```

```
   void push(T);
   T pop() { return *--p; }
   };
template<class T> void stack<T>::push(T a) {
   *p++ = a;
   }
template<class T>      stack<T>::stack(int s) {
   v = p = new T[sz=s];
   }
```

File `3.cpp`:

```
#pragma Off(Auto_class_member_instantiation)
#include "stack.h"
f() {
   stack<int> I(10);
   stack<float> F(20);
   I.push(1);    F.push(1.0);
   I.pop();      F.pop();
   }
```

File `4.cpp`:

```
#pragma Off(Auto_class_member_instantiation)
#include "stack.h"
f() {
   stack<int>    I(10);
   stack<float>  F(20);
   I.push(2);
   F.push(2.0);
   I.pop();
   F.pop();
   }
```

In Example 35 Turning off automatic instantiation, both modules contain external references to the following:

```
stack<int>::push(int)
stack<int>::stack<int>(int)
stack<float>::push(float)
stack<float>::stack<float>(int)
```

Note that there are no references to the **inline** members (such as **pop**) of **stack<int>** and **stack<float>**. These **inline** members are in fact generated by the compiler and inlined. The external references are made only to the non-**inline** functions and static data members.

You can provide the **push()** and constructor functions by compiling module `cti.cpp`, which simply refers to the two stack types:

File `cti.cpp`:

```
#include "stack.h"
typedef stack<int>   dummy1;
typedef stack<float> dummy2;
```

Selective template instantiation

Regardless of how toggle `Auto_class_member_instantiation` is set, you can ensure or suppress the instantiation of selected template types with pragmas `Ensure_instantiation(`*T1*`, `*T2*`,...`*Tn*`)` and `Exclude_instantiation(`*T1*`, `*T2*`,...`*Tn*`)`, where *T1*, *T2*,...*Tn* are names of specific template types.

How to ensure instantiation

When you use pragma `Ensure_instantiation`, the class members for each template type in its argument list are instantiated. Here is an example:

```
#pragma Off(Auto_class_member_instantiation)
template <class T> struct Q {
   int f();
   };
template <class T> Q<T>::f() {
   return sizeof(T);
   }
typedef Q<int> x;
typedef Q<double> y;
typedef Q<Q<int> > z;
// Make sure Q<double> and Q<Q<int> > are instantiated
#pragma Ensure_instantiation(Q<double>, Q<Q<int> >)
// Q<int> will NOT be instantiated, because it is not listed
```

How to exclude instantiations

When you use pragma `Exclude_instantiation`, you suppress instantiation of the class members for each template type in its argument list.

You can use pragma `Exclude_instantiation` to resolve the single-copy problem in programs compiled from two or more source modules. If you find, after linking the program, that a template has been instantiated in two or more object files, add pragma `Exclude_instantiation` to all but one of the source files and recompile.

In the following example, file `1.cc` and file `2.cc` both contain an instantiation of **Q<int>**:

File `x.h`:

```
template <class T> struct Q {
   int f();
   }
template <class T> Q<T>::f() {
   ...
   }
```

File `1.cc`:

```
#include "x.h"
Q<int> x;
```

File `2.cc`:

```
#include "x.h"
Q<int> y;
```

To resolve the problem, modify file `2.cc` as follows, and recompile:

Modified file `2.cc`:

```
#include "x.h"
```

```
Q<int> y;
#pragma Exclude_instantiation(Q<int>)
```

How to force instantiation of all template types

You can use toggle `Ensure_instantiation` to force instantiation of template types. Any template used within the scope of this toggle gets instantiated. For example:

```
#pragma Off(Auto_class_member_instantiation)
template <class T> struct Q {
   int f();
   }
template <class T> Q<T>::f() {
   return sizeof(T);
   }
typedef Q<int> x;
typedef Q<double> y;
#pragma On(Ensure_instantiation)
// Q<int> and Q<Q<int> > will both be instantiated
typedef Q<Q<int> > z;
#pragma Off(Ensure_instantiation)
```

With toggle `Ensure_instantiation`, it is harder to predict exactly what will be instantiated than with pragma `Ensure_instantiation`, because one template usage can indirectly cause other templates to be used. In the last example, the use of `Q<int>` causes the expansion of `Q`'s class body, in which any other template types used are then instantiated.

Checking template usage

To get an idea of which templates were used in a compilation, turn on toggle `Print_template_usage`. This toggle prints both function templates and class templates. It also prints the members of a class template, indented below the class template. A single character in the left-most column indicates the following:

- `I` means the compiler instantiated this item
- `e` means undefined (the item is external) and you must supply it
- `U` means the user defined a special version of a function, class member, or class template

The following output shows the result of compiling `3.cpp` (see Example 35 Turning off automatic instantiation):

```
Template usage:
I    stack<int>            (at "3.cpp",L5/C7)
e       stack(int)         (at "stack.h",L4/C11)
e       void push(int)     (at "stack.h",L6/C16)
I    stack<float>          (at "3.cpp",L5/C25)
e       stack(int)         (at "stack.h",L4/C11)
e       void push(float)   (at "stack.h",L6/C16)
```

This output shows that the two **stack** classes are instantiated (`I`) but the two member functions are external (`e`).

When you compile `cti.cpp` (see the discussion following Example 35 Turning off automatic instantiation), you see the two **stack** classes instantiated:

```
Template usage:
```

```
I    stack<int>            (at "cti.cpp",L2/C11)
I       stack(int s)       (at "stack.h",L10/C26)
I       void push(int a)   (at "stack.h",L9/C26)
I    stack<float>          (at "cti.cpp",L3/C11)
I       stack(int s)       (at "stack.h",L10/C26)
I       void push(float a) (at "stack.h",L9/C26)
```