24 Other Topics

What's in a name? that which we call a rose
By any other name would smell as sweet.
—William Shakespeare

O Diamond! Diamond! thou little knowest the mischief done!
—Sir Isaac Newton

Objectives

In this chapter you'll learn:

- To use `const_cast` to temporarily treat a `const` object as a non-`const` object.
- To use `namespace`s.
- To use operator keywords.
- To use `mutable` members in `const` objects.
- To use class-member pointer operators `.*` and `->*`.
- To use multiple inheritance.
- The role of `virtual` base classes in multiple inheritance.

24.1 Introduction

We now consider additional C++ features. First, we discuss the `const_cast` operator, which allows you to add or remove the `const` qualification of a variable. Next, we discuss `namespaces`, which can be used to ensure that every identifier in a program has a unique name and can help resolve naming conflicts caused by using libraries that have the same variable, function or class names. We then present several operator keywords that are useful for programmers who have keyboards that do not support certain characters used in operator symbols, such as `!`, `&`, `^`, `~` and `|`. We continue our discussion with the `mutable` storage-class specifier, which enables you to indicate that a data member should always be modifiable, even when it appears in an object that's currently being treated as a `const` object by the program. Next we introduce two special operators that we can use with pointers to class members to access a data member or member function without knowing its name in advance. Finally, we introduce multiple inheritance, which enables a derived class to inherit the members of several base classes. As part of this introduction, we discuss potential problems with multiple inheritance and how `virtual` inheritance can be used to solve those problems.

24.2 `const_cast` Operator

C++ provides the **`const_cast`** operator for casting away `const` or `volatile` qualification. You declare a variable with the **`volatile`** qualifier when you expect the variable to be modified by hardware or other programs not known to the compiler. Declaring a variable `volatile` indicates that the compiler should not optimize the use of that variable because doing so could affect the ability of those other programs to access and modify the `volatile` variable.

In general, it's dangerous to use the `const_cast` operator, because it allows a program to modify a variable that was declared `const`. There are cases in which it's desirable, or even necessary, to cast away `const`-ness. For example, older C and C++ libraries might provide functions that have non-`const` parameters and that do not modify their parameters—if you wish to pass `const` data to such a function, you'd need to cast away the data's `const`-ness; otherwise, the compiler would report error messages.

Similarly, you could pass non-`const` data to a function that treats the data as if it were constant, then returns that data as a constant. In such cases, you might need to cast away the `const`-ness of the returned data, as we demonstrate in Fig. 24.1.

```
1  // Fig. 24.1: fig24_01.cpp
2  // Demonstrating const_cast.
3  #include <iostream>
4  #include <cstring> // contains prototypes for functions strcmp and strlen
5  #include <cctype> // contains prototype for function toupper
6  using namespace std;
7
8  // returns the larger of two C-style strings
9  const char *maximum( const char *first, const char *second )
10 {
11    return ( strcmp( first, second ) >= 0 ? first : second );
12 } // end function maximum
13
14 int main()
15 {
16    char s1[] = "hello"; // modifiable array of characters
17    char s2[] = "goodbye"; // modifiable array of characters
18
19    // const_cast required to allow the const char * returned by maximum
20    // to be assigned to the char * variable maxPtr
21    char *maxPtr = const_cast< char * >( maximum( s1, s2 ) );
22
23    cout << "The larger string is: " << maxPtr << endl;
24
25    for ( size_t i = 0; i < strlen( maxPtr ); ++i )
26       maxPtr[ i ] = toupper( maxPtr[ i ] );
27
28    cout << "The larger string capitalized is: " << maxPtr << endl;
29 } // end main
```

```
The larger string is: hello
The larger string capitalized is: HELLO
```

Fig. 24.1 | Demonstrating operator `const_cast`.

In this program, function `maximum` (lines 9–12) receives two C-style strings as `const char *` parameters and returns a `const char *` that points to the larger of the two strings. Function `main` declares the two C-style strings as non-`const char` arrays (lines 16–17); thus, these arrays are modifiable. In `main`, we wish to output the larger of the two C-style strings, then modify that C-style string by converting it to uppercase letters.

Function `maximum`'s two parameters are of type `const char *`, so the function's return type also must be declared as `const char *`. If the return type is specified as only `char *`, the compiler issues an error message indicating that the value being returned cannot be converted from `const char *` to `char *`—a dangerous conversion, because it attempts to treat data that the function believes to be `const` as if it were non-`const` data.

Even though function `maximum` *believes* the data to be constant, we know that the original arrays in `main` do *not* contain constant data. Therefore, `main` should be able to modify the contents of those arrays as necessary. Since we know these arrays are modifiable, we use `const_cast` (line 21) to *cast away the const-ness* of the pointer returned by `maximum`, so we can then modify the data in the array representing the larger of the two C-style

strings. We can then use the pointer as the name of a character array in the `for` statement (lines 25–26) to convert the contents of the larger string to uppercase letters. Without the `const_cast` in line 21, this program will not compile, because you are not allowed to assign a pointer of type `const char *` to a pointer of type `char *`.

Error-Prevention Tip 24.1

In general, a `const_cast` should be used only when it is known in advance that the original data is not constant. Otherwise, unexpected results may occur.

24.3 mutable Class Members

In Section 24.2, we introduced the `const_cast` operator, which allowed us to remove the "`const`-ness" of a type. A `const_cast` operation can also be applied to a data member of a `const` object from the body of a `const` member function of that object's class. This enables the `const` member function to modify the data member, even though the object is considered to be `const` in the body of that function. Such an operation might be performed when most of an object's data members should be considered `const`, but a particular data member still needs to be modified.

As an example, consider a linked list that maintains its contents in sorted order. Searching through the linked list does not require modifications to the data of the linked list, so the search function could be a `const` member function of the linked-list class. However, it's conceivable that a linked-list object, in an effort to make future searches more efficient, might keep track of the location of the last successful match. If the next search operation attempts to locate an item that appears later in the list, the search could begin from the location of the last successful match, rather than from the beginning of the list. To do this, the `const` member function that performs the search must be able to modify the data member that keeps track of the last successful search.

If a data member such as the one described above should *always* be modifiable, C++ provides the storage-class specifier **`mutable`** as an alternative to `const_cast`. A `mutable` data member is always modifiable, even in a `const` member function or `const` object.

Portability Tip 24.1

The effect of attempting to modify an object that was defined as constant, regardless of whether that modification was made possible by a `const_cast` or C-style cast, varies among compilers.

`mutable` and `const_cast` are used in different contexts. For a `const` object with no `mutable` data members, operator `const_cast` *must* be used every time a member is to be modified. This greatly reduces the chance of a member being accidentally modified because the member is not permanently modifiable. Operations involving `const_cast` are typically *hidden* in a member function's implementation. The user of a class might not be aware that a member is being modified.

Software Engineering Observation 24.1

`mutable` members are useful in classes that have "secret" implementation details that do not contribute to a client's use of an object of the class.

Mechanical Demonstration of a `mutable` *Data Member*

Figure 24.2 demonstrates using a `mutable` member. The program defines class `TestMutable` (lines 7–21), which contains a constructor, function `getValue` and a `private` data member `value` that's declared `mutable`. Lines 15–18 define function `getValue` as a `const` member function that returns a copy of `value`. Notice that the function increments `mutable` data member `value` in the `return` statement. Normally, a `const` member function cannot modify data members unless the object on which the function operates—i.e., the one to which `this` points—is cast (using `const_cast`) to a non-`const` type. Because `value` is `mutable`, this `const` function can modify the data.

```
1  // Fig. 24.2: fig24_02.cpp
2  // Demonstrating storage-class specifier mutable.
3  #include <iostream>
4  using namespace std;
5
6  // class TestMutable definition
7  class TestMutable
8  {
9  public:
10    TestMutable( int v = 0 )
11    {
12       value = v;
13    } // end TestMutable constructor
14
15    int getValue() const
16    {
17       return ++value; // increments value
18    } // end function getValue
19 private:
20    mutable int value; // mutable member
21 }; // end class TestMutable
22
23 int main()
24 {
25    const TestMutable test( 99 );
26
27    cout << "Initial value: " << test.getValue();
28    cout << "\nModified value: " << test.getValue() << endl;
29 } // end main
```

```
Initial value: 99
Modified value: 100
```

Fig. 24.2 | Demonstrating a `mutable` data member.

Line 25 declares `const TestMutable` object `test` and initializes it to `99`. Line 27 calls the `const` member function `getValue`, which adds one to `value` and returns its previous contents. Notice that the compiler *allows* the call to member function `getValue` on the object `test` because it's a `const` object and `getValue` is a `const` member function. However, `getValue` modifies variable `value`. Thus, when line 28 invokes `getValue` again, the new `value` (100) is output to prove that the `mutable` data member was indeed modified.

24.4 namespaces

A program may include many identifiers defined in different scopes. Sometimes a variable of one scope will "overlap" (i.e., collide) with a variable of the *same* name in a *different* scope, possibly creating a naming conflict. Such overlapping can occur at many levels. Identifier overlapping occurs frequently in third-party libraries that happen to use the same names for global identifiers (such as functions). This can cause compiler errors.

The C++ standard solves this problem with **namespaces**. Each `namespace` defines a scope in which identifiers and variables are placed. To use a **`namespace` member**, either the member's name must be qualified with the `namespace` name and the scope resolution operator (`::`), as in

```
MyNameSpace::member
```

or a `using` directive must appear *before* the name is used in the program. Typically, such `using` statements are placed at the beginning of the file in which members of the `namespace` are used. For example, placing the following `using` directive at the beginning of a source-code file

```
using namespace MyNameSpace;
```

specifies that members of `namespace` *MyNameSpace* can be used in the file without preceding each member with *MyNameSpace* and the scope resolution operator (`::`).

A `using` directive of the form

```
using std::cout;
```

brings *one* name into the scope where the directive appears. A `using` directive f the form

```
using namespace std;
```

brings *all* the names from the specified namespace (`std`) into the scope where the directive appears.

Error-Prevention Tip 24.2

Precede a member with its `namespace` *name and the scope resolution operator (*`::`*) if the possibility exists of a naming conflict.*

Not all `namespace`*s are guaranteed to be unique.* Two third-party vendors might inadvertently use the same identifiers for their `namespace` names. Figure 24.3 demonstrates the use of `namespace`s.

```
1  // Fig. 24.3: fig24_03.cpp
2  // Demonstrating namespaces.
3  #include <iostream>
4  using namespace std;
5
6  int integer1 = 98; // global variable
7
8  // create namespace Example
9  namespace Example
10 {
```

Fig. 24.3 | Demonstrating the use of `namespace`s. (Part 1 of 3.)

```
11    // declare two constants and one variable
12    const double PI = 3.14159;
13    const double E = 2.71828;
14    int integer1 = 8;
15
16    void printValues(); // prototype
17
18    // nested namespace
19    namespace Inner
20    {
21       // define enumeration
22       enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
23    } // end Inner namespace
24 } // end Example namespace
25
26 // create unnamed namespace
27 namespace
28 {
29    double doubleInUnnamed = 88.22; // declare variable
30 } // end unnamed namespace
31
32 int main()
33 {
34    // output value doubleInUnnamed of unnamed namespace
35    cout << "doubleInUnnamed = " << doubleInUnnamed;
36
37    // output global variable
38    cout << "\n(global) integer1 = " << integer1;
39
40    // output values of Example namespace
41    cout << "\nPI = " << Example::PI << "\nE = " << Example::E
42       << "\ninteger1 = " << Example::integer1 << "\nFISCAL3 = "
43       << Example::Inner::FISCAL3 << endl;
44
45    Example::printValues(); // invoke printValues function
46 } // end main
47
48 // display variable and constant values
49 void Example::printValues()
50 {
51    cout << "\nIn printValues:\ninteger1 = " << integer1 << "\nPI = "
52       << PI << "\nE = " << E << "\ndoubleInUnnamed = "
53       << doubleInUnnamed << "\n(global) integer1 = " << ::integer1
54       << "\nFISCAL3 = " << Inner::FISCAL3 << endl;
55 } // end printValues
```

```
doubleInUnnamed = 88.22
(global) integer1 = 98
PI = 3.14159
E = 2.71828
integer1 = 8
FISCAL3 = 1992
```

Fig. 24.3 | Demonstrating the use of namespaces. (Part 2 of 3.)

```
In printValues:
integer1 = 8
PI = 3.14159
E = 2.71828
doubleInUnnamed = 88.22
(global) integer1 = 98
FISCAL3 = 1992
```

Fig. 24.3 | Demonstrating the use of `namespaces`. (Part 3 of 3.)

Defining Namespaces

Lines 9–24 use the keyword `namespace` to define namespace `Example`. The body of a namespace is delimited by braces (`{}`). Namespace `Example`'s members consist of two constants (`PI` and `E` in lines 12–13), an `int` (`integer1` in line 14), a function (`printValues` in line 16) and a **nested namespace** (`Inner` in lines 19–23). Notice that member `integer1` has the same name as global variable `integer1` (line 6). *Variables that have the same name must have different scopes*—otherwise compilation errors occur. A namespace can contain constants, data, classes, nested namespaces, functions, etc. Definitions of namespaces must occupy the *global scope* or be *nested* within other namespaces. Unlike classes, different namespace members can be defined in separate `namespace` blocks—each standard library header has a `namespace` block placing its contents in `namespace std`.

Lines 27–30 create an **unnamed namespace** containing the member `doubleInUnnamed`. Variables, classes and functions in an unnamed namespace are accessible only in the current **translation unit** (a `.cpp` file and the files it `includes`). However, unlike variables, classes or functions with `static` linkage, those in the unnamed namespace may be used as template arguments. The unnamed namespace has an implicit `using` directive, so its members appear to occupy the **global namespace**, are accessible directly and do not have to be qualified with a namespace name. Global variables are also part of the global namespace and are accessible in all scopes following the declaration in the file.

Software Engineering Observation 24.2

Each separate compilation unit has its own unique unnamed namespace; i.e., the unnamed namespace replaces the `static` *linkage specifier.*

Accessing Namespace Members with Qualified Names

Line 35 outputs the value of variable `doubleInUnnamed`, which is directly accessible as part of the unnamed namespace. Line 38 outputs the value of global variable `integer1`. For both of these variables, the compiler first attempts to locate a local declaration of the variables in `main`. Since there are no local declarations, the compiler assumes those variables are in the global namespace.

Lines 41–43 output the values of `PI`, `E`, `integer1` and `FISCAL3` from namespace `Example`. Notice that each must be qualified with `Example::` because the program does not provide any `using` directive or declarations indicating that it will use members of namespace `Example`. In addition, member `integer1` must be qualified, because a global variable has the same name. Otherwise, the global variable's value is output. `FISCAL3` is a member of nested namespace `Inner`, so it must be qualified with `Example::Inner::`.

Function `printValues` (defined in lines 49–55) is a member of `Example`, so it can access other members of the `Example` namespace directly without using a namespace qualifier. The output statement in lines 51–54 outputs `integer1`, `PI`, `E`, `doubleInUnnamed`, global variable `integer1` and `FISCAL3`. Notice that `PI` and `E` are not qualified with `Example`. Variable `doubleInUnnamed` is still accessible, because it's in the unnamed namespace and the variable name does not conflict with any other members of namespace `Example`. The global version of `integer1` must be qualified with the scope resolution operator (`::`), because its name conflicts with a member of namespace `Example`. Also, `FISCAL3` must be qualified with `Inner::`. When accessing members of a nested namespace, the members must be qualified with the namespace name (unless the member is being used inside the nested namespace).

Common Programming Error 24.1

Placing `main` in a `namespace` is a compilation error.

`using` Directives Should Not *Be Placed in Headers*

Namespaces are particularly useful in large-scale applications that use many class libraries. In such cases, there's a higher likelihood of naming conflicts. When working on such projects, there should *never* be a `using` directive in a header. Having one brings the corresponding names into any file that includes the header. This could result in name collisions and subtle, hard-to-find errors. Instead, use only fully qualified names in headers (for example, `std::cout` or `std::string`).

Aliases for Namespace Names

Namespaces can be *aliased*. For example the statement

```
namespace CPPHTP = CPlusPlusHowToProgram;
```

creates the **namespace alias** `CPPHTP` for `CPlusPlusHowToProgram`.

24.5 Operator Keywords

The C++ standard provides **operator keywords** (Fig. 24.4) that can be used in place of several C++ operators. You can use operator keywords if you have keyboards that do not support certain characters such as `!`, `&`, `^`, `~`, `|`, etc.

Operator	Operator keyword	Description
Logical operator keywords		
`&&`	`and`	logical AND
`||`	`or`	logical OR
`!`	`not`	logical NOT
Inequality operator keyword		
`!=`	`not_eq`	inequality

Fig. 24.4 | Operator keyword alternatives to operator symbols. (Part 1 of 2.)

Operator	Operator keyword	Description
Bitwise operator keywords		
&	`bitand`	bitwise AND
|	`bitor`	bitwise inclusive OR
^	`xor`	bitwise exclusive OR
~	`compl`	bitwise complement
Bitwise assignment operator keywords		
&=	`and_eq`	bitwise AND assignment
|=	`or_eq`	bitwise inclusive OR assignment
^=	`xor_eq`	bitwise exclusive OR assignment

Fig. 24.4 | Operator keyword alternatives to operator symbols. (Part 2 of 2.)

Figure 24.5 demonstrates the operator keywords. Microsoft Visual C++ 2010 requires the header <ciso646> (line 4) to use the operator keywords. In GNU C++, this header is empty because the operator keywords are always defined.

```
1  // Fig. 24.5: fig24_05.cpp
2  // Demonstrating operator keywords.
3  #include <iostream>
4  #include <ciso646> // enables operator keywords in Microsoft Visual C++
5  using namespace std;
6
7  int main()
8  {
9     bool a = true;
10    bool b = false;
11    int c = 2;
12    int d = 3;
13
14    // sticky setting that causes bool values to display as true or false
15    cout << boolalpha;
16
17    cout << "a = " << a << "; b = " << b
18       << "; c = " << c << "; d = " << d;
19
20    cout << "\n\nLogical operator keywords:";
21    cout << "\n   a and a: " << ( a and a );
22    cout << "\n   a and b: " << ( a and b );
23    cout << "\n    a or a: " << ( a or a );
24    cout << "\n    a or b: " << ( a or b );
25    cout << "\n     not a: " << ( not a );
26    cout << "\n     not b: " << ( not b );
27    cout << "\na not_eq b: " << ( a not_eq b );
28
```

Fig. 24.5 | Demonstrating the operator keywords. (Part 1 of 2.)

```
29      cout << "\n\nBitwise operator keywords:";
30      cout << "\nc bitand d: " << ( c bitand d );
31      cout << "\nc bit_or d: " << ( c bitor d );
32      cout << "\n   c xor d: " << ( c xor d );
33      cout << "\n   compl c: " << ( compl c );
34      cout << "\nc and_eq d: " << ( c and_eq d );
35      cout << "\n c or_eq d: " << ( c or_eq d );
36      cout << "\nc xor_eq d: " << ( c xor_eq d ) << endl;
37   } // end main
```

```
a = true; b = false; c = 2; d = 3

Logical operator keywords:
   a and a: true
   a and b: false
    a or a: true
    a or b: true
     not a: false
     not b: true
a not_eq b: true

Bitwise operator keywords:
c bitand d: 2
c bit_or d: 3
   c xor d: 1
   compl c: -3
c and_eq d: 2
 c or_eq d: 3
c xor_eq d: 0
```

Fig. 24.5 | Demonstrating the operator keywords. (Part 2 of 2.)

The program declares and initializes two `bool` variables and two integer variables (lines 9–12). Logical operations (lines 21–27) are performed with `bool` variables `a` and `b` using the various logical operator keywords. Bitwise operations (lines 30–36) are performed with the `int` variables `c` and `d` using the various bitwise operator keywords. The result of each operation is output.

24.6 Pointers to Class Members (`.*` and `->*`)

C++ provides the `.*` and `->*` operators for accessing class members via pointers. This is a rarely used capability that's used primarily by advanced C++ programmers. We provide only a mechanical example of using pointers to class members here. Figure 24.6 demonstrates the pointer-to-class-member operators.

```
1  // Fig. 24.6: fig24_06.cpp
2  // Demonstrating operators .* and ->*.
3  #include <iostream>
4  using namespace std;
5
```

Fig. 24.6 | Demonstrating the `.*` and `->*` operators. (Part 1 of 2.)

```
// class Test definition
class Test
{
public:
   void func()
   {
      cout << "In func\n";
   } // end function func

   int value; // public data member
}; // end class Test

void arrowStar( Test * ); // prototype
void dotStar( Test * ); // prototype

int main()
{
   Test test;
   test.value = 8; // assign value 8
   arrowStar( &test ); // pass address to arrowStar
   dotStar( &test ); // pass address to dotStar
} // end main

// access member function of Test object using ->*
void arrowStar( Test *testPtr )
{
   void ( Test::*memberPtr )() = &Test::func; // declare function pointer
   ( testPtr->*memberPtr )(); // invoke function indirectly
} // end arrowStar

// access members of Test object data member using .*
void dotStar( Test *testPtr2 )
{
   int Test::*vPtr = &Test::value; // declare pointer
   cout << ( *testPtr2 ).*vPtr << endl; // access value
} // end dotStar
```

```
In test function
8
```

Fig. 24.6 | Demonstrating the .* and ->* operators. (Part 2 of 2.)

The program declares class `Test` (lines 7–16), which provides `public` member function `test` and `public` data member `value`. Lines 18–19 provide prototypes for the functions `arrowStar` (defined in lines 30–34) and `dotStar` (defined in lines 37–41), which demonstrate the `->*` and `.*` operators, respectively. Lines 23 creates object `test`, and line 24 assigns `8` to its data member `value`. Lines 25–26 call functions `arrowStar` and `dotStar` with the address of the object `test`.

Line 32 in function `arrowStar` declares and initializes variable `memPtr` as a pointer to a member function. In this declaration, `Test::*` indicates that the variable `memPtr` is a pointer to a member of class `Test`. To declare a pointer to a function, enclose the pointer name preceded by `*` in parentheses, as in `(Test::*memPtr)`. A pointer to a function must

specify, as part of its type, both the return type of the function it points to and the parameter list of that function. The function's return type appears to the left of the left parenthesis and the parameter list appears in a separate set of parentheses to the right of the pointer declaration. In this case, the function has a `void` return type and no parameters. The pointer `memPtr` is initialized with the address of class `Test`'s member function named `test`. The header of the function must match the function pointer's declaration—i.e., function `test` must have a `void` return type and no parameters. Notice that the right side of the assignment uses the address operator (`&`) to get the address of the member function `test`. Also, notice that *neither the left side nor the right side of the assignment in line 32 refers to a specific object of class* `Test`. Only the class name is used with the scope resolution operator (`::`). Line 33 invokes the member function stored in `memPtr` (i.e., `test`), using the `->*` operator. Because `memPtr` is a pointer to a member of a class, the `->*` operator must be used rather than the `->` operator to invoke the function.

Line 39 declares and initializes `vPtr` as a pointer to an `int` data member of class `Test`. The right side of the assignment specifies the address of the data member `value`. Line 40 dereferences the pointer `testPtr2`, then uses the `.*` operator to access the member to which `vPtr` points. *The client code can create pointers to class members for only those class members that are accessible to the client code.* In this example, both member function `test` and data member `value` are publicly accessible.

Common Programming Error 24.2

Declaring a member-function pointer without enclosing the pointer name in parentheses is a syntax error.

Common Programming Error 24.3

Declaring a member-function pointer without preceding the pointer name with a class name followed by the scope resolution operator (`::`) is a syntax error.

Common Programming Error 24.4

Attempting to use the `->` or `*` operator with a pointer to a class member generates syntax errors.

24.7 Multiple Inheritance

In Chapters 12 and 13, we discussed *single inheritance, in which each class is derived from exactly one base class*. In C++, a class may be derived from *more than one* base class—a technique known as **multiple inheritance** in which a derived class inherits the members of two or more base classes. This powerful capability encourages interesting forms of software reuse but can cause a variety of ambiguity problems. *Multiple inheritance is a difficult concept that should be used only by experienced programmers.* In fact, some of the problems associated with multiple inheritance are so subtle that newer programming languages, such as Java and C#, do not enable a class to derive from more than one base class.

Software Engineering Observation 24.3

Great care is required in the design of a system to use multiple inheritance properly; it should not be used when single inheritance and/or composition will do the job.

A common problem with multiple inheritance is that each of the base classes might contain data members or member functions that have the same name. This can lead to ambiguity problems when you attempt to compile. Consider the multiple-inheritance example (Fig. 24.7, Fig. 24.8, Fig. 24.9, Fig. 24.10, Fig. 24.11). Class `Base1` (Fig. 24.7) contains one `protected` `int` data member—`value` (line 20), a constructor (lines 10–13) that sets `value` and `public` member function `getData` (lines 15–18) that returns `value`.

```
1  // Fig. 24.7: Base1.h
2  // Definition of class Base1
3  #ifndef BASE1_H
4  #define BASE1_H
5
6  // class Base1 definition
7  class Base1
8  {
9  public:
10    Base1( int parameterValue )
11    {
12       value = parameterValue;
13    } // end Base1 constructor
14
15    int getData() const
16    {
17       return value;
18    } // end function getData
19 protected: // accessible to derived classes
20    int value; // inherited by derived class
21 }; // end class Base1
22
23 #endif // BASE1_H
```

Fig. 24.7 | Demonstrating multiple inheritance—`Base1.h`.

Class `Base2` (Fig. 24.8) is similar to class `Base1`, except that its `protected` data is a `char` named `letter` (line 20). Like class `Base1`, `Base2` has a `public` member function `getData`, but this function returns the value of `char` data member `letter`.

```
1  // Fig. 24.8: Base2.h
2  // Definition of class Base2
3  #ifndef BASE2_H
4  #define BASE2_H
5
6  // class Base2 definition
7  class Base2
8  {
9  public:
10    Base2( char characterData )
11    {
12       letter = characterData;
13    } // end Base2 constructor
```

Fig. 24.8 | Demonstrating multiple inheritance—`Base2.h`. (Part 1 of 2.)

```
14
15    char getData() const
16    {
17       return letter;
18    } // end function getData
19 protected: // accessible to derived classes
20    char letter; // inherited by derived class
21 }; // end class Base2
22
23 #endif // BASE2_H
```

Fig. 24.8 | Demonstrating multiple inheritance—`Base2.h`. (Part 2 of 2.)

Class `Derived` (Figs. 24.9–24.10) inherits from both class `Base1` and class `Base2` through multiple inheritance. Class `Derived` has a `private` data member of type `double` named `real` (line 20), a constructor to initialize all the data of class `Derived` and a `public` member function `getReal` that returns the value of `double` variable `real`.

```
1  // Fig. 24.9: Derived.h
2  // Definition of class Derived which inherits
3  // multiple base classes (Base1 and Base2).
4  #ifndef DERIVED_H
5  #define DERIVED_H
6
7  #include <iostream>
8  #include "Base1.h"
9  #include "Base2.h"
10 using namespace std;
11
12 // class Derived definition
13 class Derived : public Base1, public Base2
14 {
15    friend ostream &operator<<( ostream &, const Derived & );
16 public:
17    Derived( int, char, double );
18    double getReal() const;
19 private:
20    double real; // derived class's private data
21 }; // end class Derived
22
23 #endif // DERIVED_H
```

Fig. 24.9 | Demonstrating multiple inheritance—`Derived.h`.

```
1  // Fig. 24.10: Derived.cpp
2  // Member-function definitions for class Derived
3  #include "Derived.h"
4
5  // constructor for Derived calls constructors for
6  // class Base1 and class Base2.
```

Fig. 24.10 | Demonstrating multiple inheritance—`Derived.cpp`. (Part 1 of 2.)

```
7  // use member initializers to call base-class constructors
8  Derived::Derived( int integer, char character, double double1 )
9     : Base1( integer ), Base2( character ), real( double1 ) { }
10
11 // return real
12 double Derived::getReal() const
13 {
14    return real;
15 } // end function getReal
16
17 // display all data members of Derived
18 ostream &operator<<( ostream &output, const Derived &derived )
19 {
20    output << "    Integer: " << derived.value << "\n  Character: "
21       << derived.letter << "\nReal number: " << derived.real;
22    return output; // enables cascaded calls
23 } // end operator<<
```

Fig. 24.10 | Demonstrating multiple inheritance—`Derived.cpp`. (Part 2 of 2.)

To indicate multiple inheritance we follow the colon (`:`) after `class Derived` with a comma-separated list of base classes (line 13). In Fig. 24.10, notice that constructor `Derived` explicitly calls base-class constructors for each of its base classes—`Base1` and `Base2`—using the member-initializer syntax (line 9). The *base-class constructors are called in the order that the inheritance is specified, not in the order in which their constructors are mentioned*; also, *if the base-class constructors are not explicitly called in the member-initializer list, their default constructors will be called implicitly.*

The overloaded stream insertion operator (Fig. 24.10, lines 18–23) uses its second parameter—a reference to a `Derived` object—to display a `Derived` object's data. This operator function is a `friend` of `Derived`, so `operator<<` can directly access *all* of class `Derived`'s `protected` and `private` members, including the `protected` data member `value` (inherited from class `Base1`), `protected` data member `letter` (inherited from class `Base2`) and `private` data member `real` (declared in class `Derived`).

Now let's examine the `main` function (Fig. 24.11) that tests the classes in Figs. 24.7–24.10. Line 11 creates `Base1` object `base1` and initializes it to the `int` value `10`, then creates the pointer `base1Ptr` and initializes it to the null pointer (i.e., 0). Line 12 creates `Base2` object `base2` and initializes it to the `char` value `'Z'`, then creates the pointer `base2Ptr` and initializes it to the null pointer. Line 13 creates `Derived` object `derived` and initializes it to contain the `int` value `7`, the `char` value `'A'` and the `double` value `3.5`.

```
1  // Fig. 24.11: fig24_11.cpp
2  // Driver for multiple-inheritance example.
3  #include <iostream>
4  #include "Base1.h"
5  #include "Base2.h"
6  #include "Derived.h"
7  using namespace std;
```

Fig. 24.11 | Demonstrating multiple inheritance. (Part 1 of 2.)

```
 8
 9  int main()
10  {
11     Base1 base1( 10 ), *base1Ptr = 0; // create Base1 object
12     Base2 base2( 'Z' ), *base2Ptr = 0; // create Base2 object
13     Derived derived( 7, 'A', 3.5 ); // create Derived object
14
15     // print data members of base-class objects
16     cout << "Object base1 contains integer " << base1.getData()
17        << "\nObject base2 contains character " << base2.getData()
18        << "\nObject derived contains:\n" << derived << "\n\n";
19
20     // print data members of derived-class object
21     // scope resolution operator resolves getData ambiguity
22     cout << "Data members of Derived can be accessed individually:"
23        << "\n    Integer: " << derived.Base1::getData()
24        << "\n  Character: " << derived.Base2::getData()
25        << "\nReal number: " << derived.getReal() << "\n\n";
26     cout << "Derived can be treated as an object of either base class:\n";
27
28     // treat Derived as a Base1 object
29     base1Ptr = &derived;
30     cout << "base1Ptr->getData() yields " << base1Ptr->getData() << '\n';
31
32     // treat Derived as a Base2 object
33     base2Ptr = &derived;
34     cout << "base2Ptr->getData() yields " << base2Ptr->getData() << endl;
35  } // end main
```

```
Object base1 contains integer 10
Object base2 contains character Z
Object derived contains:
    Integer: 7
  Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
    Integer: 7
  Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Fig. 24.11 | Demonstrating multiple inheritance. (Part 2 of 2.)

Lines 16–18 display each object's data values. For objects `base1` and `base2`, we invoke each object's `getData` member function. Even though there are two `getData` functions in this example, the calls are not ambiguous. In line 16, the compiler knows that `base1` is an object of class `Base1`, so class `Base1`'s `getData` is called. In line 17, the compiler knows that `base2` is an object of class `Base2`, so class `Base2`'s `getData` is called. Line 18 displays the contents of object `derived` using the overloaded stream insertion operator.

Resolving Ambiguity Issues That Arise When a Derived Class Inherits Member Functions of the Same Name from Multiple Base Classes
Lines 22–25 output the contents of object `derived` again by using the *get* member functions of class `Derived`. However, there is an *ambiguity* problem, because this object contains two `getData` functions, one inherited from class `Base1` and one inherited from class `Base2`. This problem is easy to solve by using the scope resolution operator. The expression `derived.Base1::getData()` gets the value of the variable inherited from class `Base1` (i.e., the `int` variable named `value`) and `derived.Base2::getData()` gets the value of the variable inherited from class `Base2` (i.e., the `char` variable named `letter`). The `double` value in `real` is printed without ambiguity with the call `derived.getReal()`—there are no other member functions with that name in the hierarchy.

Demonstrating the Is-A *Relationships in Multiple Inheritance*
The *is-a* relationships of single inheritance also apply in multiple-inheritance relationships. To demonstrate this, line 29 assigns the address of object `derived` to the `Base1` pointer `base1Ptr`. This is allowed because an object of class `Derived` *is an* object of class `Base1`. Line 30 invokes `Base1` member function `getData` via `base1Ptr` to obtain the value of only the `Base1` part of the object `derived`. Line 33 assigns the address of object `derived` to the `Base2` pointer `base2Ptr`. This is allowed because an object of class `Derived` *is an* object of class `Base2`. Line 34 invokes `Base2` member function `getData` via `base2Ptr` to obtain the value of only the `Base2` part of the object `derived`.

24.8 Multiple Inheritance and virtual Base Classes

In Section 24.7, we discussed multiple inheritance, the process by which one class inherits from two or more classes. Multiple inheritance is used, for example, in the C++ standard library to form class `basic_iostream` (Fig. 24.12).

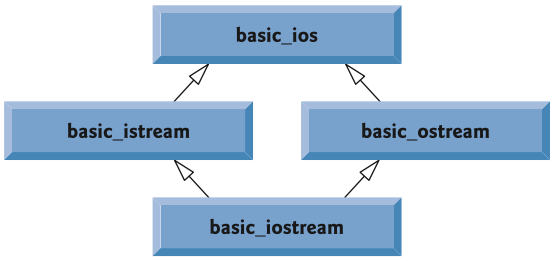

Fig. 24.12 | Multiple inheritance to form class `basic_iostream`.

Class `basic_ios` is the base class for both `basic_istream` and `basic_ostream`, each of which is formed with single inheritance. Class `basic_iostream` inherits from both `basic_istream` and `basic_ostream`. This enables class `basic_iostream` objects to provide the functionality of `basic_istreams` and `basic_ostreams`. In multiple-inheritance hierarchies, the situation described in Fig. 24.12 is referred to as **diamond inheritance**

Because classes `basic_istream` and `basic_ostream` each inherit from `basic_ios`, a potential problem exists for `basic_iostream`. Class `basic_iostream` could contain *two* copies of the members of class `basic_ios`—one inherited via class `basic_istream` and one

inherited via class `basic_ostream`). Such a situation would be ambiguous and would result in a compilation error, because the compiler would not know which version of the members from class `basic_ios` to use. Of course, `basic_iostream` does not really suffer from the problem we mentioned. In this section, you'll see how using `virtual` base classes solves the problem of inheriting duplicate copies of an indirect base class.

Compilation Errors Produced When Ambiguity Arises in Diamond Inheritance

Figure 24.13 demonstrates the ambiguity that can occur in diamond inheritance. Class `Base` (lines 8–12) contains pure `virtual` function `print` (line 11). Classes `DerivedOne` (lines 15–23) and `DerivedTwo` (lines 26–34) each publicly inherit from `Base` and override function `print`. Class `DerivedOne` and class `DerivedTwo` each contain what the C++ standard refers to as a **base-class subobject**—i.e., the members of class `Base` in this example.

```
1  // Fig. 24.13: fig24_13.cpp
2  // Attempting to polymorphically call a function that is
3  // multiply inherited from two base classes.
4  #include <iostream>
5  using namespace std;
6
7  // class Base definition
8  class Base
9  {
10 public:
11    virtual void print() const = 0; // pure virtual
12 }; // end class Base
13
14 // class DerivedOne definition
15 class DerivedOne : public Base
16 {
17 public:
18    // override print function
19    void print() const
20    {
21       cout << "DerivedOne\n";
22    } // end function print
23 }; // end class DerivedOne
24
25 // class DerivedTwo definition
26 class DerivedTwo : public Base
27 {
28 public:
29    // override print function
30    void print() const
31    {
32       cout << "DerivedTwo\n";
33    } // end function print
34 }; // end class DerivedTwo
35
36 // class Multiple definition
37 class Multiple : public DerivedOne, public DerivedTwo
38 {
```

Fig. 24.13 | Attempting to call a multiply inherited function polymorphically. (Part 1 of 2.)

```
39   public:
40      // qualify which version of function print
41      void print() const
42      {
43         DerivedTwo::print();
44      } // end function print
45   }; // end class Multiple
46
47   int main()
48   {
49      Multiple both; // instantiate Multiple object
50      DerivedOne one; // instantiate DerivedOne object
51      DerivedTwo two; // instantiate DerivedTwo object
52      Base *array[ 3 ]; // create array of base-class pointers
53
54      array[ 0 ] = &both; // ERROR--ambiguous
55      array[ 1 ] = &one;
56      array[ 2 ] = &two;
57
58      // polymorphically invoke print
59      for ( int i = 0; i < 3; ++i )
60         array[ i ] -> print();
61   } // end main
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp8_examples\ch25\Fig24_13\fig24_13.cpp(54) : error C2594: '=' :
   ambiguous conversions from 'Multiple *' to 'Base *'
```

GNU C++ compiler error message:

```
fig24_13.cpp: In function 'int main()':
fig24_13.cpp:54: error: 'Base' is an ambiguous base of 'Multiple'
```

Fig. 24.13 | Attempting to call a multiply inherited function polymorphically. (Part 2 of 2.)

Class `Multiple` (lines 37–45) inherits from both classes `DerivedOne` and `DerivedTwo`. In class `Multiple`, function `print` is overridden to call `DerivedTwo`'s `print` (line 43). Notice that we must qualify the `print` call with the class name `DerivedTwo` to specify which version of `print` to call.

Function `main` (lines 47–61) declares objects of classes `Multiple` (line 49), `DerivedOne` (line 50) and `DerivedTwo` (line 51). Line 52 declares an array of `Base *` pointers. Each array element is initialized with the address of an object (lines 54–56). An error occurs when the address of `both`—an object of class `Multiple`—is assigned to `array[ 0 ]`. The object `both` actually contains two subobjects of type `Base`, so the compiler does not know which subobject the pointer `array[ 0 ]` should point to, and it generates a compilation error indicating an ambiguous conversion.

Eliminating Duplicate Subobjects with `virtual` *Base-Class Inheritance*

The problem of duplicate subobjects is resolved with `virtual` inheritance. When a base class is inherited as `virtual`, only one subobject will appear in the derived class—a process

called **virtual base-class inheritance**. Figure 24.14 revises the program of Fig. 24.13 to use a `virtual` base class.

```
// Fig. 24.14: fig24_14.cpp
// Using virtual base classes.
#include <iostream>
using namespace std;

// class Base definition
class Base
{
public:
   virtual void print() const = 0; // pure virtual
}; // end class Base

// class DerivedOne definition
class DerivedOne : virtual public Base
{
public:
   // override print function
   void print() const
   {
      cout << "DerivedOne\n";
   } // end function print
}; // end DerivedOne class

// class DerivedTwo definition
class DerivedTwo : virtual public Base
{
public:
   // override print function
   void print() const
   {
      cout << "DerivedTwo\n";
   } // end function print
}; // end DerivedTwo class

// class Multiple definition
class Multiple : public DerivedOne, public DerivedTwo
{
public:
   // qualify which version of function print
   void print() const
   {
      DerivedTwo::print();
   } // end function print
}; // end Multiple class

int main()
{
   Multiple both; // instantiate Multiple object
   DerivedOne one; // instantiate DerivedOne object
```

Fig. 24.14 | Using `virtual` base classes. (Part 1 of 2.)

```
50      DerivedTwo two; // instantiate DerivedTwo object
51
52      // declare array of base-class pointers and initialize
53      // each element to a derived-class type
54      Base *array[ 3 ];
55      array[ 0 ] = &both;
56      array[ 1 ] = &one;
57      array[ 2 ] = &two;
58
59      // polymorphically invoke function print
60      for ( int i = 0; i < 3; ++i )
61         array[ i ]->print();
62   } // end main
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

Fig. 24.14 | Using virtual base classes. (Part 2 of 2.)

The key change is that classes DerivedOne (line 14) and DerivedTwo (line 25) each inherit from Base by specifying virtual public Base. Since both classes inherit from Base, they each contain a Base subobject. The benefit of virtual inheritance is not clear until class Multiple inherits from DerivedOne and DerivedTwo (line 36). Since each of the base classes used virtual inheritance to inherit class Base's members, the compiler ensures that only one Base subobject is inherited into class Multiple. This eliminates the ambiguity error generated by the compiler in Fig. 24.13. The compiler now allows the implicit conversion of the derived-class pointer (&both) to the base-class pointer array[0] in line 55 in main. The for statement in lines 60–61 polymorphically calls print for each object.

Constructors in Multiple-Inheritance Hierarchies with `virtual` *Base Classes*

Implementing hierarchies with virtual base classes is simpler if default constructors are used for the base classes. Figures 24.13 and 24.14 use compiler-generated ones. If a virtual base class provides a constructor that requires arguments, the derived-class implementations become more complicated, because the **most derived class** must explicitly invoke the virtual base class's constructor.

Software Engineering Observation 24.4

Providing a default constructor for virtual *base classes simplifies hierarchy design.*

Additional Information on Multiple Inheritance

Multiple inheritance is a complex topic typically covered in more advanced C++ texts. For more information on multiple inheritance, please visit our C++ Resource Center at

```
www.deitel.com/cplusplus/
```

In the *C++ Multiple Inheritance* category, you'll find links to several articles and resources, including a multiple inheritance FAQ and tips for using multiple inheritance.

24.9 Wrap-Up

In this chapter, you learned how to use the `const_cast` operator to remove the `const` qualification of a variable. We then showed how to use `namespaces` to ensure that every identifier in a program has a unique name and explained how they can help resolve naming conflicts. You saw several operator keywords to use if your keyboards do not support certain characters used in operator symbols, such as `!`, `&`, `^`, `~` and `|`. Next, we showed how the `mutable` storage-class specifier enables you to indicate that a data member should always be modifiable, even when it appears in an object that's currently being treated as a `const`. We also showed the mechanics of using pointers to class members and the `->*` and `.*` operators. Finally, we introduced multiple inheritance and discussed problems associated with allowing a derived class to inherit the members of several base classes. As part of this discussion, we demonstrated how `virtual` inheritance can be used to solve those problems.

Summary

Section 24.2 `const_cast` Operator

- C++ provides the `const_cast` operator for casting away `const` or `volatile` qualification.
- A program declares a variable with the `volatile` qualifier (p. 975) when that program expects the variable to be modified by other programs. Declaring a variable `volatile` indicates that the compiler should not optimize the use of that variable because doing so could affect the ability of those other programs to access and modify the `volatile` variable.
- In general, it is dangerous to use the `const_cast` operator, because it allows a program to modify a variable that was declared `const`, and thus was not supposed to be modifiable.
- There are cases in which it is desirable, or even necessary, to cast away `const`-ness. For example, older C and C++ libraries might provide functions with non-`const` parameters and that do not modify their parameters. If you wish to pass `const` data to such a function, you'd need to cast away the data's `const`-ness; otherwise, the compiler would report error messages.
- If you pass non-`const` data to a function that treats the data as if it were constant, then returns that data as a constant, you might need to cast away the `const`-ness of the returned data to access and modify that data.

Section 24.3 `mutable` Class Members

- If a data member should always be modifiable, C++ provides the storage-class specifier `mutable` as an alternative to `const_cast`. A `mutable` data member (p. 977) is always modifiable, even in a `const` member function or `const` object. This reduces the need to cast away "`const`-ness."
- `mutable` and `const_cast` are used in different contexts. For a `const` object with no `mutable` data members, operator `const_cast` must be used every time a member is to be modified. This greatly reduces the chance of a member being accidentally modified because the member is not permanently modifiable.
- Operations involving `const_cast` are typically hidden in a member function's implementation. The user of a class might not be aware that a member is being modified.

Section 24.4 `namespaces`

- A program includes many identifiers defined in different scopes. Sometimes a variable of one scope will "overlap" with a variable of the same name in a different scope, possibly creating a naming conflict. The C++ standard solves this problem with `namespaces` (p. 979).