23

Other Topics

Objectives

In this chapter you'll:

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

Outline

23.1 Introduction

We now consider additional C++ features, including:

- Storage classes and storage duration, which determine an object's lifetime in a program.
- The `const_cast` operator, which allows you to add or remove the `const` qualification of a variable.
- Storage classes and storage duration, which determine the period during which that identifier exists in memory.
- `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.
- *Operator keywords* that are useful for programmers who have keyboards that do not support certain characters used in operator symbols, such as `!`, `&`, `^`, `~` and `|`.
- Special operators that you can use with pointers to class members to access a data member or member function *without knowing its name* in advance.
- *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 them.

23.2 `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 errors.

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. 23.1.

```
1  // Fig. 23.1: fig23_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 strings
9  const char* maximum(const char* first, const char* second) {
10    return (strcmp(first, second) >= 0 ? first : second);
11 }
12
13 int main() {
14    char s1[]{"hello"}; // modifiable array of characters
15    char s2[]{"goodbye"}; // modifiable array of characters
16
17    // const_cast required to allow the const char* returned by maximum
18    // to be assigned to the char* variable maxPtr
19    char* maxPtr{const_cast<char*>(maximum(s1, s2))};
20
21    cout << "The larger string is: " << maxPtr << endl;
22
23    for (size_t i{0}; i < strlen(maxPtr); ++i) {
24       maxPtr[i] = toupper(maxPtr[i]);
25    }
26
27    cout << "The larger string capitalized is: " << maxPtr << endl;
28 }
```

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

Fig. 23.1 | Demonstrating operator `const_cast`.

In this program, function `maximum` (lines 9–11) receives two C 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 strings as non-`const` `char` arrays (lines 14–15); thus, these arrays are modifiable. In `main`, we wish to output the larger of the two C strings, then modify that C 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 19) 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 strings. We can then use the pointer as the name of a character array in the `for` statement (lines 23–25) to convert the contents of the larger string to uppercase letters. Without the `const_cast` in line 19, 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 23.1

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.

23.3 Storage Classes and Storage Duration

The programs you've seen so far use identifiers for variable names and functions. The attributes of variables include *name*, *type*, *size* and *value*. Each identifier in a program also has other attributes, including scope, **linkage** and **storage duration**.

As we discussed in Section 6.11, an identifier's *scope* is *where the identifier can be referenced* in a program. Some identifiers can be referenced throughout a program; others can be referenced from only limited portions of a program. An identifier's *linkage* determines whether it's known only in the *source file where it's declared* or *across multiple files that are compiled, then linked together*. An identifier's *storage-class specifier* helps determine its storage duration and linkage.

Storage Class Specifiers

C++ provides several **storage-class specifiers** that determine a variable's storage duration: **`extern`**, **`mutable`**, **`static`** and `thread_local`. Storage-class specifier `mutable` is used exclusively with classes and `thread_local` is used in multithreaded applications.

23.3.1 Storage Duration

An identifier's *storage duration* determines the period during which that identifier's storage *exists in memory*. Some exist briefly, some are repeatedly created and destroyed, and others exist for a program's entire execution.

The storage-class specifiers can be split into four storage durations: *automatic*, *static*, *dynamic* and *thread*. In Chapter 10, you learned that you can request additional memory in your program during the program's execution—so-called *dynamic memory allocation*. Variables allocated dynamically have *dynamic storage duration*. Chapter 24 discusses *thread storage duration*. The rest of this section focuses on automatic and static storage duration.

23.3.2 Local Variables and Automatic Storage Duration

Variables with **automatic storage duration** include:

- local variables declared in functions
- function parameters

Such variables—often called *automatic variables*—are created when program execution enters the block in which they're defined, they exist while the block is active and they're destroyed when the program exits the block. An automatic variable exists only from where it's defined to the *closing brace* of the block in which the definition appears, or for the entire function body in the case of a function parameter. Local variables are of automatic storage duration by *default*.

Performance Tip 23.1

Automatic storage is a means of conserving memory, because automatic storage duration variables exist in memory only when the block in which they're defined is executing.

Good Programming Practice 23.1

Declare variables as close to where they're first used as possible.

23.3.3 Static Storage Duration

Keywords `extern` and `static` declare identifiers for variables with **static storage duration** and functions. Variables with static storage duration exist in memory from the point at which the program begins execution and until the program terminates. Such a variable is *initialized once when its declaration is encountered*. For functions, the name of the function exists when the program begins execution. Even though function names and static-storage-duration variables exist from the start of program execution, their scope determines where they can be used in the program.

Identifiers with Static Storage Duration

There are two types of identifiers with *static storage duration*—external identifiers (such as global variables) and local variables declared with the storage-class specifier `static`. **Global variables** are created by placing variable declarations *outside* any class or function definition. Global variables retain their values throughout a program's execution. Global variables and global functions can be referenced by any function that follows their declarations or definitions in the source file.

Software Engineering Observation 23.1

Declaring a variable as global rather than local allows unintended side effects *to occur when a function that does not need access to the variable accidentally or maliciously modifies it. This is another example of the principle of least privilege—in general, except for truly global resources such as* `cin` *and* `cout`*, the use of global variables should be avoided unless there are unique performance requirements.*

Software Engineering Observation 23.2

Variables used only in a particular function should be declared as local variables in that function rather than as global variables.

`static` Local Variables

Local variables declared `static` are still known only in the function in which they're declared, but, unlike automatic variables, *`static` local variables retain their values when the function returns to its caller*. The next time the function is called, the `static` local variables contain the values they had when the function last completed execution. The following statement declares local variable `count` to be `static` and to be initialized to 1:

```
static unsigned int count{1};
```

All numeric variables of `static` storage duration are *initialized to zero by default*, but it's nevertheless a good practice to explicitly initialize all variables.

Storage-class specifiers `extern` and `static` have special meaning when they're applied explicitly to external identifiers such as global variables and global function names. In Appendix F, C Legacy Code Topics, we discuss using `extern` and `static` with external identifiers and multiple-source-file programs.

23.3.4 `mutable` Class Members

In Section 23.2, 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. Though `mutable` is a storage-class specifier, it does not affect a variable's storage duration or linkage.

Portability Tip 23.1

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 23.3

`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.

23.3.5 Mechanical Demonstration of a `mutable` Data Member

Figure 23.2 demonstrates using a `mutable` member. The program defines class `TestMutable` (lines 7–16), which contains a constructor, function `getValue` and a `private` data member `value` that's declared `mutable`. Lines 11–13 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* to a non-`const` type via `const_cast`. Because `value` is `mutable` (line 15), this `const` function *can* modify the data.

```
1  // Fig. 23.2: fig23_02.cpp
2  // Demonstrating storage-class specifier mutable.
3  #include <iostream>
4  using namespace std;
5
6  // class TestMutable definition
7  class TestMutable {
8  public:
9     TestMutable(int v = 0) : value{v} { }
10
11    int getValue() const {
12       return ++value; // increments value
13    }
14 private:
15    mutable int value; // mutable member
16 };
17
18 int main() {
19    const TestMutable test{99};
20
21    cout << "Initial value: " << test.getValue();
22    cout << "\nModified value: " << test.getValue() << endl;
23 }
```

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

Fig. 23.2 | Demonstrating a `mutable` data member.

Line 19 declares `const TestMutable` object `test` and initializes it to `99`. Line 21 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 22 invokes `getValue` again, the new `value` (`100`) is output to prove that the `mutable` data member was indeed *modified*.

23.4 namespaces

A program may include many identifiers defined in different scopes. Sometimes a variable of one scope will 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 compilation errors.

C++ 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 of the form

```
using namespace std;
```

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

Error-Prevention Tip 23.2

Precede a member with its `namespace` *name and the scope resolution operator* (`::`) *to prevent naming conflicts.*

Not all `namespaces` *are guaranteed to be unique.* Two third-party vendors might inadvertently use the same identifiers for their `namespace` names. Figure 23.3 demonstrates the use of `namespaces`.

```
1  // Fig. 23.3: fig23_03.cpp
2  // Demonstrating namespaces.
3  #include <iostream>
4  using namespace std;
```

Fig. 23.3 | Demonstrating the use of `namespaces`. (Part 1 of 3.)

```
5
6  int integer1 = 98; // global variable
7
8  // create namespace Example
9  namespace Example {
10    // declare two constants and one variable
11    const double PI = 3.14159;
12    const double E = 2.71828;
13    int integer1 = 8;
14
15    void printValues(); // prototype
16
17    // nested namespace
18    namespace Inner {
19       // define enumeration
20       enum Years {FISCAL1 = 2017, FISCAL2, FISCAL3};
21    }
22 }
23
24 // create unnamed namespace
25 namespace {
26    double doubleInUnnamed = 88.22; // declare variable
27 }
28
29 int main() {
30    // output value doubleInUnnamed of unnamed namespace
31    cout << "doubleInUnnamed = " << doubleInUnnamed;
32
33    // output global variable
34    cout << "\n(global) integer1 = " << integer1;
35
36    // output values of Example namespace
37    cout << "\nPI = " << Example::PI << "\nE = " << Example::E
38       << "\ninteger1 = " << Example::integer1 << "\nFISCAL3 = "
39       << Example::Inner::FISCAL3 << endl;
40
41    Example::printValues(); // invoke printValues function
42 }
43
44 // display variable and constant values
45 void Example::printValues() {
46    cout << "\nIn printValues:\ninteger1 = " << integer1 << "\nPI = "
47       << PI << "\nE = " << E << "\ndoubleInUnnamed = "
48       << doubleInUnnamed << "\n(global) integer1 = " << ::integer1
49       << "\nFISCAL3 = " << Inner::FISCAL3 << endl;
50 }
```

```
doubleInUnnamed = 88.22
(global) integer1 = 98
PI = 3.14159
E = 2.71828
integer1 = 8
FISCAL3 = 2019
```

Fig. 23.3 | 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 = 2019
```

Fig. 23.3 | Demonstrating the use of `namespaces`. (Part 3 of 3.)

23.4.1 Defining namespaces

Lines 9–22 use the keyword `namespace` to define `namespace Example`. The body of a `namespace` is delimited by braces (`{}`). The `namespace Example`'s members consist of two constants (`PI` and `E` in lines 11–12), an `int` (`integer1` in line 13), a function (`printValues` in line 15) and a **nested namespace** (`Inner` in lines 18–21). 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 25–27 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 23.4

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

23.4.2 Accessing namespace Members with Qualified Names

Line 31 outputs the value of variable `doubleInUnnamed`, which is directly accessible as part of the *unnamed* `namespace`. Line 34 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 37–39 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 45–50) is a member of `Example`, so it can access other members of the `Example` `namespace` directly *without using a* `namespace` *qualifier*. Line 46–49 output `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 23.1

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

23.4.3 `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. For such projects, there should *never* be a `using` directive in a header—this 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`).

23.4.4 Aliases for `namespace` Names

`namespaces` can be *aliased*. This might be useful when dealing with long namespace identifiers or nested namespaces. For example, assuming we have the `namespace` identifier `CPlusPlusHowToProgram`, the statement

```
namespace CPPHTP = CPlusPlusHowToProgram;
```

creates the shorter **namespace alias** `CPPHTP` for `CPlusPlusHowToProgram`. Similarly, you could define the an alias for the nested `namespace` `Inner` in Fig. 23.3 as follows:

```
namespace Innermost = Example::Inner;
```

23.5 Operator Keywords

The C++ standard provides **operator keywords** (Fig. 23.4) 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

Fig. 23.4 | Operator keyword alternatives to operator symbols. (Part 1 of 2.)

Operator	Operator keyword	Description
Inequality operator keyword		
!=	not_eq	inequality
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. 23.4 | Operator keyword alternatives to operator symbols. (Part 2 of 2.)

Figure 23.5 demonstrates the operator keywords. Microsoft Visual C++ requires the header <ciso646> (line 4) to use the operator keywords. In GNU C++ and LLVM, the operator keywords are always defined and this header is not required.

```
1  // Fig. 23.5: fig23_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     bool a{true};
9     bool b{false};
10    int c{2};
11    int d{3};
12
13    // sticky setting that causes bool values to display as true or false
14    cout << boolalpha;
15
16    cout << "a = " << a << "; b = " << b
17       << "; c = " << c << "; d = " << d;
18
19    cout << "\n\nLogical operator keywords:";
20    cout << "\n   a and a: " << (a and a)  ;
21    cout << "\n   a and b: " << (a and b)  ;
22    cout << "\n    a or a: " << (a or a)  ;
23    cout << "\n    a or b: " << (a or b)  ;
24    cout << "\n     not a: " << (not a)  ;
25    cout << "\n     not b: " << (not b)  ;
26    cout << "\na not_eq b: " << (a not_eq b)  ;
```

Fig. 23.5 | Demonstrating operator keywords. (Part 1 of 2.)

```
27
28      cout << "\n\nBitwise operator keywords:";
29      cout << "\nc bitand d: " << (c bitand d)  ;
30      cout << "\n c bitor d: " << (c bitor d)  ;
31      cout << "\n   c xor d: " << (c xor d)  ;
32      cout << "\n   compl c: " << (compl c)  ;
33      cout << "\nc and_eq d: " << (c and_eq d)  ;
34      cout << "\n c or_eq d: " << (c or_eq d)  ;
35      cout << "\nc xor_eq d: " << (c xor_eq d)  << endl;
36   }
```

```
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 bitor 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. 23.5 | Demonstrating operator keywords. (Part 2 of 2.)

The program declares and initializes two `bool` variables and two integer variables (lines 8–11). Logical operations (lines 20–26) are performed with `bool` variables `a` and `b` using the various logical operator keywords. Bitwise operations (lines 29–35) are performed with the `int` variables `c` and `d` using the various bitwise operator keywords. The result of each operation is output.

23.6 Pointers to Class Members (.* and ->*)

C++ provides the `.*` and `->*` operators for accessing class members via pointers. This is a rarely used capability, primarily for advanced C++ programmers. We provide only a mechanical example of using pointers to class members here. Figure 23.6 demonstrates the pointer-to-class-member operators.

```
1  // Fig. 23.6: fig23_06.cpp
2  // Demonstrating operators .* and ->*.
3  #include <iostream>
4  using namespace std;
5
```

Fig. 23.6 | Demonstrating operatprs `.*` and `->*`. (Part 1 of 2.)

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

   int value; // public data member
};

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
}

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

// 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
}
```

```
In test function
8
```

Fig. 23.6 | Demonstrating operatprs `.*` and `->*`. (Part 2 of 2.)

The program declares class `Test` (lines 7–14), which provides `public` member function `test` and `public` data member `value`. Lines 16–17 provide prototypes for the functions `arrowStar` (defined in lines 27–30) and `dotStar` (defined in lines 33–36), which demonstrate the `->*` and `.*` operators, respectively. Line 30 creates object `test`, and line 21 assigns `8` to its data member `value`. Lines 22–23 call functions `arrowStar` and `dotStar` with the address of the object `test`.

Line 28 in function `arrowStar` declares and initializes variable `memberPtr` as a *pointer to a member function*. In this declaration, `Test::*` indicates that the variable `memberPtr` 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::*memberPtr)`. 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 `memberPtr` 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 29 invokes the member function stored in `memberPtr` (i.e., `test`), using the `->*` operator. Because `memberPtr` is a pointer to a member of a class, the `->*` operator must be used rather than the `->` operator to invoke the function.

Line 34 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 35 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 23.2

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

Common Programming Error 23.3

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 23.4

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

23.7 Multiple Inheritance

In Chapters 11 and 12, 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 23.5

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 (Figs. 23.7–23.11). Class `Base1` (Fig. 23.7) contains one `protected int` data

member—`value` (line 20), a constructor (line 9) that sets `value` and `public` member function `getData` (line 11) that returns `value`.

```
1  // Fig. 23.7: Base1.h
2  // Definition of class Base1
3  #ifndef BASE1_H
4  #define BASE1_H
5
6  // class Base1 definition
7  class Base1 {
8  public:
9     Base1(int parameterValue) : value{parameterValue} {}
10
11    int getData() const {return value;}
12 protected: // accessible to derived classes
13    int value; // inherited by derived class
14 };
15
16 #endif // BASE1_H
```

Fig. 23.7 | Demonstrating multiple inheritance—`Base1.h`.

Class `Base2` (Fig. 23.8) is similar to class `Base1`, except that its `protected` data is a `char` named `letter` (line 13). Like class `Base1`, `Base2` has a `public` member function `getData`, but this function returns the value of `char` data member `letter`.

```
1  // Fig. 23.8: Base2.h
2  // Definition of class Base2
3  #ifndef BASE2_H
4  #define BASE2_H
5
6  // class Base2 definition
7  class Base2 {
8  public:
9     Base2(char characterData) : letter{characterData} {}
10
11    char getData() const {return letter;}
12 protected: // accessible to derived classes
13    char letter; // inherited by derived class
14 };
15
16 #endif // BASE2_H
```

Fig. 23.8 | Demonstrating multiple inheritance—`Base2.h`.

Class `Derived` (Figs. 23.9–23.10) inherits from both class `Base1` and class `Base2` through *multiple inheritance*. Class `Derived` has a `private` data member of type `double` named `real` (Fig. 23.9, line 19), 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. 23.9: 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    friend ostream &operator<<(ostream &, const Derived &);
15 public:
16    Derived(int, char, double);
17    double getReal() const;
18 private:
19    double real; // derived class's private data
20 };
21
22 #endif // DERIVED_H
```

Fig. 23.9 | Demonstrating multiple inheritance—`Derived.h`.

```
1  // Fig. 23.10: 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.
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 {return real;}
13
14 // display all data members of Derived
15 ostream &operator<<(ostream &output, const Derived &derived) {
16    output << "    Integer: " << derived.value << "\n  Character: "
17           << derived.letter << "\nReal number: " << derived.real;
18    return output; // enables cascaded calls
19 } <<
```

Fig. 23.10 | Demonstrating multiple inheritance—`Derived.cpp`.

To indicate *multiple inheritance* (in Fig. 23.9) we follow the colon (:) after `class Derived` with a comma-separated list of base classes (line 13). In Fig. 23.10, 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 construc-*

tors 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. 23.10, lines 15–19) 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. 23.11) that tests the classes in Figs. 23.7–23.10. Line 10 creates `Base1` object `base1` and initializes it to the `int` value `10`. Line 11 creates `Base2` object `base2` and initializes it to the `char` value `'Z'`. Line 12 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. 23.11: fig23_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;
8
9  int main() {
10    Base1 base1{10}; // create Base1 object
11    Base2 base2{'Z'}; // create Base2 object
12    Derived derived{7, 'A', 3.5}; // create Derived object
13
14    // print data members of base-class objects
15    cout << "Object base1 contains integer " << base1.getData()
16       << "\nObject base2 contains character " << base2.getData()
17       << "\nObject derived contains:\n" << derived << "\n\n";
18
19    // print data members of derived-class object
20    // scope resolution operator resolves getData ambiguity
21    cout << "Data members of Derived can be accessed individually:"
22       << "\n    Integer: " << derived.Base1::getData()
23       << "\n  Character: " << derived.Base2::getData()
24       << "\nReal number: " << derived.getReal() << "\n\n";
25    cout << "Derived can be treated as an object of either base class:\n";
26
27    // treat Derived as a Base1 object
28    Base1* base1Ptr = &derived;
29    cout << "base1Ptr->getData() yields " << base1Ptr->getData() << '\n';
30
31    // treat Derived as a Base2 object
32    Base2* base2Ptr = &derived;
33    cout << "base2Ptr->getData() yields " << base2Ptr->getData() << endl;
34 }
```

Fig. 23.11 | Demonstrating multiple inheritance. (Part 1 of 2.)

```
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. 23.11 | Demonstrating multiple inheritance. (Part 2 of 2.)

Lines 15–17 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 15, the compiler knows that `base1` is an object of class `Base1`, so class `Base1`'s `getData` is called. In line 16, the compiler knows that `base2` is an object of class `Base2`, so class `Base2`'s `getData` is called. Line 17 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 21–24 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 28 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 29 invokes `Base1` member function `getData` via `base1Ptr` to obtain the value of only the `Base1` part of the object `derived`. Line 32 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 32 invokes `Base2` member function `getData` via `base2Ptr` to obtain the value of only the `Base2` part of the object `derived`.

23.8 Multiple Inheritance and `virtual` Base Classes

In Section 23.7, 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. 23.12).

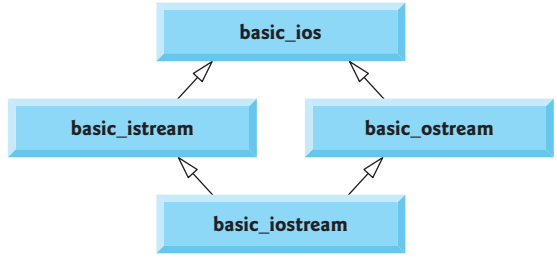

Fig. 23.12 | Multiple inheritance to form class `basic_iostream`.

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 inheritance described in Fig. 23.12 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. 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 23.13 demonstrates the *ambiguity* that can occur in *diamond inheritance*. Class `Base` (lines 8–11) contains pure `virtual` function `print` (line 10). Classes `DerivedOne` (lines 14–18) and `DerivedTwo` (lines 21–25) each publicly inherit from `Base` and override function `print`. Class `DerivedOne` and class `DerivedTwo` each contain a **base-class subobject**—i.e., the members of class `Base` in this example.

```
1  // Fig. 23.13: fig23_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 {
```

Fig. 23.13 | Attempting to call a multiply inherited function polymorphically. (Part 1 of 2.)

```
 9  public:
10     virtual void print() const = 0; // pure virtual
11  };
12
13  // class DerivedOne definition
14  class DerivedOne : public Base {
15  public:
16     // override print function
17     void print() const {cout << "DerivedOne\n";}
18  };
19
20  // class DerivedTwo definition
21  class DerivedTwo : public Base {
22  public:
23     // override print function
24     void print() const {cout << "DerivedTwo\n";}
25  };
26
27  // class Multiple definition
28  class Multiple : public DerivedOne, public DerivedTwo {
29  public:
30     // qualify which version of function print
31     void print() const {DerivedTwo::print();}
32  };
33
34  int main() {
35     Multiple both; // instantiate Multiple object
36     DerivedOne one; // instantiate DerivedOne object
37     DerivedTwo two; // instantiate DerivedTwo object
38     Base* array[3]; // create array of base-class pointers
39
40     array[0] = &both; // ERROR--ambiguous
41     array[1] = &one;
42     array[2] = &two;
43
44     // polymorphically invoke print
45     for (int i{0}; i < 3; ++i) {
46        array[i] -> print();
47     }
48  }
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp10_examples\ch23\fig23_13\fig23_13.cpp(54) : error C2594: '=' :
   ambiguous conversions from 'Multiple *' to 'Base *'
```

Fig. 23.13 | Attempting to call a multiply inherited function polymorphically. (Part 2 of 2.)

Class `Multiple` (lines 28–32) inherits from *both* class `DerivedOne` and class `DerivedTwo`. In class `Multiple`, function `print` is overridden to call `DerivedTwo`'s `print` (line 31). Notice that we must *qualify* the `print` call with the class name `DerivedTwo` to specify which version of `print` to call.

Function `main` (lines 34–48) declares objects of classes `Multiple` (line 35), `DerivedOne` (line 36) and `DerivedTwo` (line 37). Line 38 declares an array of `Base*`

pointers. Each array element is initialized with the address of an object (lines 40–42). 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 23.14 revises the program of Fig. 23.13 to use a `virtual` base class.

```
1  // Fig. 23.14: fig23_14.cpp
2  // Using virtual base classes.
3  #include <iostream>
4  using namespace std;
5
6  // class Base definition
7  class Base {
8  public:
9     virtual void print() const = 0; // pure virtual
10 };
11
12 // class DerivedOne definition
13 class DerivedOne : virtual public Base {
14 public:
15    // override print function
16    void print() const {cout << "DerivedOne\n";}
17 };
18
19 // class DerivedTwo definition
20 class DerivedTwo : virtual public Base {
21 public:
22    // override print function
23    void print() const {cout << "DerivedTwo\n";}
24 };
25
26 // class Multiple definition
27 class Multiple : public DerivedOne, public DerivedTwo {
28 public:
29    // qualify which version of function print
30    void print() const {DerivedTwo::print();}
31 };
32
33 int main() {
34    Multiple both; // instantiate Multiple object
35    DerivedOne one; // instantiate DerivedOne object
36    DerivedTwo two; // instantiate DerivedTwo object
37    Base* array[3];
38
39    array[0] = &both;
```

Fig. 23.14 | Using `virtual` base classes. (Part 1 of 2.)

```
40      array[1] = &one;
41      array[2] = &two;
42
43      // polymorphically invoke function print
44      for (int i = 0; i < 3; ++i) {
45         array[i]->print();
46      }
47   }
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

Fig. 23.14 | Using `virtual` base classes. (Part 2 of 2.)

The key change is that classes `DerivedOne` (line 13) and `DerivedTwo` (line 20) 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 27). 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. 23.13. The compiler now allows the implicit conversion of the derived-class pointer (`&both`) to the base-class pointer `array[0]` in line 39 in `main`. The `for` statement in lines 44–46 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 23.13 and 23.14 use compiler-generated *default constructors*. 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 23.6

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

23.9 Wrap-Up

In this chapter, you learned how to use the `const_cast` operator to remove the `const` qualification of a variable. We showed how to use `namespaces` to ensure that every identifier in a program has a unique name and explained how `namespaces` 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 `|`. 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.

Self-Review Exercises

23.1 Fill in the blanks for each of the following:

a) The _____ operator qualifies a member with its namespace.
b) The _____ operator allows an object's "`const`-ness" to be cast away.
c) Because an unnamed `namespace` has an implicit `using` directive, its members appear to occupy the _____, are accessible directly and do not have to be qualified with a `namespace` name.
d) Operator _____ is the operator keyword for inequality.
e) _____ allows a class to be derived from more than one base class.
f) When a base class is inherited as _____, only one subobject of the base class will appear in the derived class.

23.2 State which of the following are *true* and which are *false*. If a statement is *false*, explain why.

a) When passing a non-`const` argument to a `const` function, the `const_cast` operator should be used to cast away the "`const`-ness" of the function.
b) A `mutable` data member cannot be modified in a `const` member function.
c) `namespaces` are guaranteed to be unique.
d) Like class bodies, `namespace` bodies also end in semicolons.
e) `namespaces` cannot have `namespaces` as members.

Answers to Self-Review Exercises

23.1 a) binary scope resolution (`::`). b) `const_cast`. c) global `namespace`. d) `not_eq`. e) multiple inheritance. f) `virtual`.

23.2 a) False. It's legal to pass a non-`const` argument to a `const` function. However, when passing a `const` reference or pointer to a non-`const` function, the `const_cast` operator should be used to cast away the "`const`-ness" of the reference or pointer
b) False. A `mutable` data member is always modifiable, even in a `const` member function.
c) False. Programmers might inadvertently choose the `namespace` already in use.
d) False. `namespace` bodies do not end in semicolons.
e) False. `namespaces` can be nested.

Exercises

23.3 ***(Fill in the Blanks)*** Fill in the blanks for each of the following:

a) Keyword _____ specifies that a `namespace` or `namespace` member is being used.
b) Operator _____ is the operator keyword for logical OR.
c) Storage specifier _____ allows a member of a `const` object to be modified.
d) The _____ qualifier specifies that an object can be modified by other programs.
e) Precede a member with its _____ name and the scope resolution operator if the possibility exists of a scoping conflict.
f) The body of a namespace is delimited by _____.
g) For a `const` object with no _____ data members, operator _____ must be used every time a member is to be modified.

23.4 ***(`Currency namespace`)*** Write a `namespace`, `Currency`, that defines constant members `ONE`, `TWO`, `FIVE`, `TEN`, `TWENTY`, `FIFTY` and `HUNDRED`. Write two short programs that use `Currency`. One program should make all constants available and the other should make only `FIVE` available.

23.5 ***(Namespaces)*** Given the `namespaces` in Fig. 23.15, determine whether each statement is *true* or *false*. Explain any *false* answers.

a) Variable `kilometers` is visible within `namespace Data`.

b) Object `string1` is visible within `namespace Data`.
c) Constant `POLAND` is not visible within `namespace Data`.
d) Constant `GERMANY` is visible within `namespace Data`.
e) Function `function` is visible to `namespace Data`.
f) Namespace `Data` is visible to `namespace CountryInformation`.
g) Object `map` is visible to `namespace CountryInformation`.
h) Object `string1` is visible within `namespace RegionalInformation`.

```
1  namespace CountryInformation {
2     using namespace std;
3     enum Countries {POLAND, SWITZERLAND, GERMANY, AUSTRIA, CZECH_REPUBLIC };
4
5     int kilometers;
6     string string1;
7
8     namespace RegionalInformation {
9        short getPopulation(); // assume definition exists
10       MapData map; // assume definition exists
11    }
12 }
13
14 namespace Data {
15    using namespace CountryInformation::RegionalInformation;
16    void* function(void*, int);
17 }
```

Fig. 23.15 | `namespaces` for Exercise 23.5.

23.6 *(`mutable` vs. `const_cast`)* Compare and contrast `mutable` and `const_cast`. Give at least one example of when one might be preferred over the other. [*Note:* This exercise does not require any code.]

23.7 *(Modifying a `const` Variable)* Write a program that uses `const_cast` to modify a `const` variable. [*Hint:* Use a pointer in your solution to point to the `const` identifier.]

23.8 *(`virtual` Base Classes)* What problem do `virtual` base classes solve?

23.9 *(`virtual` Base Classes)* Write a program that uses `virtual` base classes. The class at the top of the hierarchy should provide a constructor that takes at least one argument (i.e., do not provide a default constructor). What challenges does this present for the inheritance hierarchy?