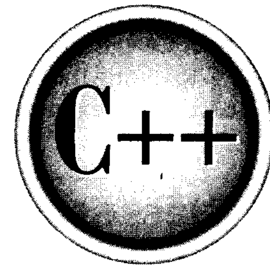
The Complete Reference



Chapter 12

Classes and Objects

In C++, the class forms the basis for object-oriented programming. The class is used to define the nature of an object, and it is C++'s basic unit of encapsulation. This chapter examines classes and objects in detail.

Classes

Classes are created using the keyword **class**. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an *instance* of a class.

A class declaration is similar syntactically to a structure. In Chapter 11, a simplified general form of a class declaration was shown. Here is the entire general form of a **class** declaration that does not inherit any other class.

```
class class-name {
  private data and functions
access-specifier:
  data and functions
access-specifier:
  data and functions
// ...
access-specifier:
  data and functions
} object-list;
```

The *object-list* is optional. If present, it declares objects of the class. Here, *access-specifier* is one of these three C++ keywords:

public

private

protected

By default, functions and data declared within a class are private to that class and may be accessed only by other members of the class. The **public** access specifier allows functions or data to be accessible to other parts of your program. The **protected** access specifier is needed only when inheritance is involved (see Chapter 15). Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.

You may change access specifications as often as you like within a **class** declaration. For example, you may switch to **public** for some declarations and then switch back to **private** again. The class declaration in the following example illustrates this feature:

```
#include <iostream>
#include <cstring>
using namespace std;

class employee {
  char name[80]; // private by default
public:
  void putname(char *n); // these are public
  void getname(char *n);
private:
  double wage; // now, private again
public:
  void putwage(double w); // back to public
  double getwage();
};

void employee::putname(char *n)
{
  strcpy(name, n);
}

void employee::getname(char *n)
{
  strcpy(n, name);
}

void employee::putwage(double w)
{
  wage = w;
}

double employee::getwage()
{
  return wage;
}

int main()
{
  employee ted;
  char name[80];

  ted.putname("Ted Jones");
  ted.putwage(75000);
```

```
    ted.getname(name);
    cout << name << " makes $";
    cout << ted.getwage() << " per year.";

    return 0;
  }
```

Here, **employee** is a simple class that is used to store an employee's name and wage. Notice that the **public** access specifier is used twice.

Although you may use the access specifiers as often as you like within a class declaration, the only advantage of doing so is that by visually grouping various parts of a class, you may make it easier for someone else reading the program to understand it. However, to the compiler, using multiple access specifiers makes no difference. Actually, most programmers find it easier to have only one **private**, **protected**, and **public** section within each class. For example, most programmers would code the **employee** class as shown here, with all private elements grouped together and all public elements grouped together:

```
class employee {
  char name[80];
  double wage;
public:
  void putname(char *n);
  void getname(char *n);
  void putwage(double w);
  double getwage();
};
```

Functions that are declared within a class are called *member functions*. Member functions may access any element of the class of which they are a part. This includes all **private** elements. Variables that are elements of a class are called *member variables* or *data members*. (The term *instance variable* is also used.) Collectively, any element of a class can be referred to as a member of that class.

There are a few restrictions that apply to class members. A non-**static** member variable cannot have an initializer. No member can be an object of the class that is being declared. (Although a member can be a pointer to the class that is being declared.) No member can be declared as **auto**, **extern**, or **register**.

In general, you should make all data members of a class private to that class. This is part of the way that encapsulation is achieved. However, there may be situations in which you will need to make one or more variables public. (For example, a heavily

used variable may need to be accessible globally in order to achieve faster run times.) When a variable is public, it may be accessed directly by any other part of your program. The syntax for accessing a public data member is the same as for calling a member function: Specify the object's name, the dot operator, and the variable name. This simple program illustrates the use of a public variable:

```
#include <iostream>
using namespace std;

class myclass {
public:
  int i, j, k; // accessible to entire program
};

int main()
{
  myclass a, b;

  a.i = 100; // access to i, j, and k is OK
  a.j = 4;
  a.k = a.i * a.j;

  b.k = 12; // remember, a.k and b.k are different
  cout << a.k << " " << b.k;

  return 0;
}
```

Structures and Classes Are Related

Structures are part of the C subset and were inherited from the C language. As you have seen, a **class** is syntactically similar to a **struct**. But the relationship between a **class** and a **struct** is closer than you may at first think. In C++, the role of the structure was expanded, making it an alternative way to specify a class. In fact, the only difference between a **class** and a **struct** is that by default all members are public in a **struct** and private in a **class**. In all other respects, structures and classes are equivalent. That is, in C++, a *structure defines a class type*. For example, consider this short program, which uses a structure to declare a class that controls access to a string:

```
// Using a structure to define a class.
#include <iostream>
#include <cstring>
```

```
using namespace std;

struct mystr {
  void buildstr(char *s); // public
  void showstr();
private: // now go private
  char str[255];
} ;

void mystr::buildstr(char *s)
{
  if(!*s) *str = '\0'; // initialize string
  else strcat(str, s);
}

void mystr::showstr()
{
  cout << str << "\n";
}

int main()
{
  mystr s;

  s.buildstr(""); // init
  s.buildstr("Hello ");
  s.buildstr("there!");

  s.showstr();

  return 0;
}
```

This program displays the string **Hello there!**.

The class **mystr** could be rewritten by using **class** as shown here:

```
class mystr {
  char str[255];
public:
  void buildstr(char *s); // public
  void showstr();
} ;
```

You might wonder why C++ contains the two virtually equivalent keywords **struct** and **class**. This seeming redundancy is justified for several reasons. First, there is no fundamental reason not to increase the capabilities of a structure. In C, structures already provide a means of grouping data. Therefore, it is a small step to allow them to include member functions. Second, because structures and classes are related, it may be easier to port existing C programs to C++. Finally, although **struct** and **class** are virtually equivalent today, providing two different keywords allows the definition of a **class** to be free to evolve. In order for C++ to remain compatible with C, the definition of **struct** must always be tied to its C definition.

Although you can use a **struct** where you use a **class**, most programmers don't. Usually it is best to use a **class** when you want a class, and a **struct** when you want a C-like structure. This is the style that this book will follow. Sometimes the acronym *POD* is used to describe a C-style structure—one that does not contain member functions, constructors, or destructors. It stands for Plain Old Data.

Remember *In C++, a structure declaration defines a class type.*

Unions and Classes Are Related

Like a structure, a **union** may also be used to define a class. In C++, **unions** may contain both member functions and variables. They may also include constructors and destructors. A **union** in C++ retains all of its C-like features, the most important being that all data elements share the same location in memory. Like the structure, **union** members are public by default and are fully compatible with C. In the next example, a **union** is used to swap the bytes that make up an **unsigned short** integer. (This example assumes that short integers are 2 bytes long.)

```
#include <iostream>
using namespace std;

union swap_byte {
  void swap();
  void set_byte(unsigned short i);
  void show_word();

  unsigned short u;
  unsigned char c[2];
};

void swap_byte::swap()
{
```

```
  unsigned char t;

  t = c[0];
  c[0] = c[1];
  c[1] = t;
}

void swap_byte::show_word()
{
  cout << u;
}

void swap_byte::set_byte(unsigned short i)
{
  u = i;
}

int main()
{
  swap_byte b;

  b.set_byte(49034);
  b.swap();
  b.show_word();

  return 0;
}
```

Like a structure, a **union** declaration in C++ defines a special type of class. This means that the principle of encapsulation is preserved.

There are several restrictions that must be observed when you use C++ unions. First, a **union** cannot inherit any other classes of any type. Further, a **union** cannot be a base class. A **union** cannot have virtual member functions. (Virtual functions are discussed in Chapter 17.) No **static** variables can be members of a **union**. A reference member cannot be used. A **union** cannot have as a member any object that overloads the = operator. Finally, no object can be a member of a **union** if the object has an explicit constructor or destructor function.

As with **struct**, the term POD is also commonly applied to unions that do not contain member functions, constructors, or destructors.

Anonymous Unions

There is a special type of **union** in C++ called an *anonymous union*. An anonymous union does not include a type name, and no objects of the union can be declared.

Instead, an anonymous union tells the compiler that its member variables are to share the same location. However, the variables themselves are referred to directly, without the normal dot operator syntax. For example, consider this program:

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
  // define anonymous union
  union {
    long l;
    double d;
    char s[4];
  } ;

  // now, reference union elements directly
  l = 100000;
  cout << l << " ";
  d = 123.2342;
  cout << d << " ";
  strcpy(s, "hi");
  cout << s;

  return 0;
}
```

As you can see, the elements of the union are referenced as if they had been declared as normal local variables. In fact, relative to your program, that is exactly how you will use them. Further, even though they are defined within a **union** declaration, they are at the same scope level as any other local variable within the same block. This implies that the names of the members of an anonymous union must not conflict with other identifiers known within the same scope.

All restrictions involving **union**s apply to anonymous ones, with these additions. First, the only elements contained within an anonymous union must be data. No member functions are allowed. Anonymous unions cannot contain **private** or **protected** elements. Finally, global anonymous unions must be specified as **static**.

Friend Functions

It is possible to grant a nonmember function access to the private members of a class by using a **friend**. A **friend** function has access to all **private** and **protected** members

of the class for which it is a **friend**. To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**. Consider this program:

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  friend int sum(myclass x);
  void set_ab(int i, int j);
};

void myclass::set_ab(int i, int j)
{
  a = i;
  b = j;
}

// Note: sum() is not a member function of any class.
int sum(myclass x)
{
  /* Because sum() is a friend of myclass, it can
     directly access a and b. */

  return x.a + x.b;
}

int main()
{
  myclass n;

  n.set_ab(3, 4);

  cout << sum(n);

  return 0;
}
```

In this example, the **sum( )** function is not a member of **myclass**. However, it still has full access to its private members. Also, notice that **sum( )** is called without the use of the dot operator. Because it is not a member function, it does not need to be (indeed, it may not be) qualified with an object's name.

Although there is nothing gained by making **sum( )** a **friend** rather than a member function of **myclass**, there are some circumstances in which **friend** functions are quite valuable. First, friends can be useful when you are overloading certain types of operators (see Chapter 14). Second, **friend** functions make the creation of some types of I/O functions easier (see Chapter 18). The third reason that **friend** functions may be desirable is that in some cases, two or more classes may contain members that are interrelated relative to other parts of your program. Let's examine this third usage now.

To begin, imagine two different classes, each of which displays a pop-up message on the screen when error conditions occur. Other parts of your program may wish to know if a pop-up message is currently being displayed before writing to the screen so that no message is accidentally overwritten. Although you can create member functions in each class that return a value indicating whether a message is active, this means additional overhead when the condition is checked (that is, two function calls, not just one). If the condition needs to be checked frequently, this additional overhead may not be acceptable. However, using a function that is a **friend** of each class, it is possible to check the status of each object by calling only this one function. Thus, in situations like this, a **friend** function allows you to generate more efficient code. The following program illustrates this concept:

```
#include <iostream>
using namespace std;

const int IDLE = 0;
const int INUSE = 1;

class C2;  // forward declaration

class C1 {
  int status;  // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};

class C2 {
  int status; // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};
```

```
void C1::set_status(int state)
{
  status = state;
}

void C2::set_status(int state)
{
  status = state;
}

int idle(C1 a, C2 b)
{
  if(a.status || b.status) return 0;
  else return 1;
}

int main()
{
  C1 x;
  C2 y;

  x.set_status(IDLE);
  y.set_status(IDLE);

  if(idle(x, y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";

  x.set_status(INUSE);

  if(idle(x, y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";

  return 0;
}
```

Notice that this program uses a *forward declaration* (also called a *forward reference*) for the class **C2**. This is necessary because the declaration of **idle( )** inside **C1** refers to **C2** before it is declared. To create a forward declaration to a class, simply use the form shown in this program.

A **friend** of one class may be a member of another. For example, here is the preceding program rewritten so that **idle( )** is a member of **C1**:

```
#include <iostream>
using namespace std;

const int IDLE = 0;
const int INUSE = 1;

class C2;  // forward declaration

class C1 {
  int status;  // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  int idle(C2 b);  // now a member of C1
};

class C2 {
  int status;  // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  friend int C1::idle(C2 b);
};

void C1::set_status(int state)
{
  status = state;
}

void C2::set_status(int state)
{
  status = state;
}

// idle() is member of C1, but friend of C2
int C1::idle(C2 b)
{
  if(status || b.status) return 0;
  else return 1;
}

int main()
{
```

```
  C1 x;
  C2 y;

  x.set_status(IDLE);
  y.set_status(IDLE);

  if(x.idle(y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";
  x.set_status(INUSE);

  if(x.idle(y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";

  return 0;
}
```

Because **idle( )** is a member of **C1**, it can access the **status** variable of objects of type **C1** directly. Thus, only objects of type **C2** need be passed to **idle( )**.

There are two important restrictions that apply to **friend** functions. First, a derived class does not inherit **friend** functions. Second, **friend** functions may not have a storage-class specifier. That is, they may not be declared as **static** or **extern**.

Friend Classes

It is possible for one class to be a **friend** of another class. When this is the case, the **friend** class and all of its member functions have access to the private members defined within the other class. For example,

```
// Using a friend class.
#include <iostream>
using namespace std;

class TwoValues {
  int a;
  int b;
public:
  TwoValues(int i, int j) { a = i; b = j; }
  friend class Min;
};

class Min {
```

```
public:
  int min(TwoValues x);
};

int Min::min(TwoValues x)
{
  return x.a < x.b ? x.a : x.b;
}

int main()
{
  TwoValues ob(10, 20);
  Min m;

  cout << m.min(ob);

  return 0;
}
```

In this example, class **Min** has access to the private variables **a** and **b** declared within the **TwoValues** class.

It is critical to understand that when one class is a **friend** of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become members of the **friend** class.

Friend classes are seldom used. They are supported to allow certain special case situations to be handled.

Inline Functions

There is an important feature in C++, called an *inline function*, that is commonly used with classes. Since the rest of this chapter (and the rest of the book) will make heavy use of it, inline functions are examined here.

In C++, you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro. To cause a function to be expanded in line rather than called, precede its definition with the **inline** keyword. For example, in this program, the function **max( )** is expanded in line instead of called:

```
#include <iostream>
using namespace std;
```

```
inline int max(int a, int b)
{
  return a>b ? a : b;
}

int main()
{
  cout << max(10, 20);
  cout << " " << max(99, 88);

  return 0;
}
```

As far as the compiler is concerned, the preceding program is equivalent to this one:

```
#include <iostream>
using namespace std;

int main()
{

  cout << (10>20 ? 10 : 20);
  cout << " " << (99>88 ? 99 : 88);

  return 0;
}
```

The reason that **inline** functions are an important addition to C++ is that they allow you to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. As you probably know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time. However, when a function is expanded in line, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to **inline** only very small functions. Further, it is also a good idea to **inline** only those functions that will have significant impact on the performance of your program.

Like the **register** specifier, **inline** is actually just a *request*, not a command, to the compiler. The compiler can choose to ignore it. Also, some compilers may not inline

all types of functions. For example, it is common for a compiler not to inline a recursive function. You will need to check your compiler's documentation for any restrictions to **inline.** Remember, if a function cannot be inlined, it will simply be called as a normal function.

Inline functions may be class member functions. For example, this is a perfectly valid C++ program:

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  void init(int i, int j);
  void show();
};

// Create an inline function.
inline void myclass::init(int i, int j)
{
  a = i;
  b = j;
}

// Create another inline function.
inline void myclass::show()
{
  cout << a << " " << b << "\n";
}

int main()
{
  myclass x;

  x.init(10, 20);
  x.show();

  return 0;
}
```

Note *The **inline** keyword is not part of the C subset of C++. Thus, it is not defined by C89. However, it has been added by C99.*

Defining Inline Functions Within a Class

It is possible to define short functions completely within a class declaration. When a function is defined inside a class declaration, it is automatically made into an **inline** function (if possible). It is not necessary (but not an error) to precede its declaration with the **inline** keyword. For example, the preceding program is rewritten here with the definitions of **init( )** and **show( )** contained within the declaration of **myclass**:

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  // automatic inline
  void init(int i, int j) { a=i; b=j; }
  void show() { cout << a << " " << b << "\n"; }
};

int main()
{
  myclass x;

  x.init(10, 20);
  x.show();

  return 0;
}
```

Notice the format of the function code within **myclass**. Because inline functions are often short, this style of coding within a class is fairly typical. However, you are free to use any format you like. For example, this is a perfectly valid way to rewrite the **myclass** declaration:

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  // automatic inline
  void init(int i, int j)
  {
    a = i;
```

```
    b = j;
  }

  void show()
  {
    cout << a << " " << b << "\n";
  }
};
```

Technically, the inlining of the **show( )** function is of limited value because (in general) the amount of time the I/O statement will take far exceeds the overhead of a function call. However, it is extremely common to see all short member functions defined inside their class in C++ programs. (In fact, it is rare to see short member functions defined outside their class declarations in professionally written C++ code.)

Constructor and destructor functions may also be inlined, either by default, if defined within their class, or explicitly.

Parameterized Constructors

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object. For example, here is a simple class that includes a parameterized constructor:

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  myclass(int i, int j) {a=i; b=j;}
  void show() {cout << a << " " << b;}
};

int main()
{
  myclass ob(3, 5);

  ob.show();

  return 0;
}
```

Notice that in the definition of **myclass( )**, the parameters **i** and **j** are used to give initial values to **a** and **b**.

The program illustrates the most common way to specify arguments when you declare an object that uses a parameterized constructor. Specifically, this statement

```
myclass ob(3, 4);
```

causes an object called **ob** to be created and passes the arguments **3** and **4** to the **i** and **j** parameters of **myclass( )**. You may also pass arguments using this type of declaration statement:

```
myclass ob = myclass(3, 4);
```

However, the first method is the one generally used, and this is the approach taken by most of the examples in this book. Actually, there is a small technical difference between the two types of declarations that relates to copy constructors. (Copy constructors are discussed in Chapter 14.)

Here is another example that uses a parameterized constructor. It creates a class that stores information about library books.

```
#include <iostream>
#include <cstring>
using namespace std;

const int IN = 1;
const int CHECKED_OUT = 0;

class book {
  char author[40];
  char title[40];
  int status;
public:
  book(char *n, char *t, int s);
  int get_status() {return status;}
  void set_status(int s) {status = s;}
  void show();
};

book::book(char *n, char *t, int s)
{
  strcpy(author, n);
```

```
  strcpy(title, t);
  status = s;
}

void book::show()
{
  cout << title << " by " << author;
  cout << " is ";
  if(status==IN) cout << "in.\n";
  else cout << "out.\n";
}

int main()
{
  book b1("Twain", "Tom Sawyer", IN);
  book b2("Melville", "Moby Dick", CHECKED_OUT);

  b1.show();
  b2.show();

  return 0;
}
```

Parameterized constructors are very useful because they allow you to avoid having to make an additional function call simply to initialize one or more variables in an object. Each function call you can avoid makes your program more efficient. Also, notice that the short **get_status( )** and **set_status( )** functions are defined in line, within the **book** class. This is a common practice when writing C++ programs.

Constructors with One Parameter: A Special Case

If a constructor only has one parameter, there is a third way to pass an initial value to that constructor. For example, consider the following short program.

```
#include <iostream>
using namespace std;

class X {
  int a;
public:
```

```
  X(int j) { a = j; }
  int geta() { return a; }
};

int main()
{
  X ob = 99; // passes 99 to j

  cout << ob.geta(); // outputs 99

  return 0;
}
```

Here, the constructor for **X** takes one parameter. Pay special attention to how **ob** is declared in **main( )**. In this form of initialization, 99 is automatically passed to the **j** parameter in the **X( )** constructor. That is, the declaration statement is handled by the compiler as if it were written like this:

```
X ob = X(99);
```

In general, any time you have a constructor that requires only one argument, you can use either *ob(i)* or *ob = i* to initialize an object. The reason for this is that whenever you create a constructor that takes one argument, you are also implicitly creating a conversion from the type of that argument to the type of the class.

Remember that the alternative shown here applies only to constructors that have exactly one parameter.

Static Class Members

Both function and data members of a class can be made **static**. This section explains the consequences of each.

Static Data Members

When you precede a member variable's declaration with **static**, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a **static** member variable are not made for each object. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable. All **static** variables are initialized to zero before the first object is created.

When you declare a **static** data member within a class, you are *not* defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the **static** variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated. (Remember, a class declaration is simply a logical construct that does not have physical reality.)

To understand the usage and effect of a **static** data member, consider this program:

```
#include <iostream>
using namespace std;

class shared {
  static int a;
  int b;
public:
  void set(int i, int j) {a=i; b=j;}
  void show();
} ;

int shared::a; // define a

void shared::show()
{
  cout << "This is static a: " << a;
  cout << "\nThis is non-static b: " << b;
  cout << "\n";
}

int main()
{
  shared x, y;

  x.set(1, 1); // set a to 1
  x.show();

  y.set(2, 2); // change a to 2
  y.show();

  x.show(); /* Here, a has been changed for both x and y
               because a is shared by both objects. */

  return 0;
}
```

This program displays the following output when run.

```
This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1
```

Notice that the integer **a** is declared both inside **shared** and outside of it. As mentioned earlier, this is necessary because the declaration of **a** inside **shared** does not allocate storage.

Note *As a convenience, older versions of C++ did not require the second declaration of a **static** member variable. However, this convenience gave rise to serious inconsistencies and it was eliminated several years ago. However, you may still find older C++ code that does not redeclare **static** member variables. In these cases, you will need to add the required definitions.*

A **static** member variable exists *before* any object of its class is created. For example, in the following short program, **a** is both **public** and **static**. Thus it may be directly accessed in **main( )**. Further, since **a** exists before an object of **shared** is created, **a** can be given a value at any time. As this program illustrates, the value of **a** is unchanged by the creation of object **x**. For this reason, both output statements display the same value: 99.

```
#include <iostream>
using namespace std;

class shared {
public:
  static int a;
} ;

int shared::a; // define a

int main()
{
  // initialize a before creating any objects
  shared::a = 99;

  cout << "This is initial value of a: " << shared::a;
```

```
  cout << "\n";

  shared x;

  cout << "This is x.a: " << x.a;

  return 0;
}
```

Notice how **a** is referred to through the use of the class name and the scope resolution operator. In general, to refer to a **static** member independently of an object, you must qualify it by using the name of the class of which it is a member.

One use of a **static** member variable is to provide access control to some shared resource used by all objects of a class. For example, you might create several objects, each of which needs to write to a specific disk file. Clearly, however, only one object can be allowed to write to the file at a time. In this case, you will want to declare a **static** variable that indicates when the file is in use and when it is free. Each object then interrogates this variable before writing to the file. The following program shows how you might use a **static** variable of this type to control access to a scarce resource:

```
#include <iostream>
using namespace std;

class cl {
  static int resource;
public:
  int get_resource();
  void free_resource() {resource = 0;}
};

int cl::resource; // define resource

int cl::get_resource()
{
  if(resource) return 0; // resource already in use
  else {
    resource = 1;
    return 1;  // resource allocated to this object
  }
}
```

```
int main()
{
  cl ob1, ob2;

  if(ob1.get_resource()) cout << "ob1 has resource\n";

  if(!ob2.get_resource()) cout << "ob2 denied resource\n";

  ob1.free_resource(); // let someone else use it

  if(ob2.get_resource())
    cout << "ob2 can now use resource\n";

  return 0;
}
```

Another interesting use of a **static** member variable is to keep track of the number of objects of a particular class type that are in existence. For example,

```
#include <iostream>
using namespace std;

class Counter {
public:
  static int count;
  Counter() { count++; }
  ~Counter() { count--; }
};
int Counter::count;

void f();

int main(void)
{
  Counter o1;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";

  Counter o2;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";
```

```
  f();
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";

  return 0;
}

void f()
{
  Counter temp;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";
  // temp is destroyed when f() returns
}
```

This program produces the following output.

```
Objects in existence: 1
Objects in existence: 2
Objects in existence: 3
Objects in existence: 2
```

As you can see, the **static** member variable **count** is incremented whenever an object is created and decremented when an object is destroyed. This way, it keeps track of how many objects of type **Counter** are currently in existence.

By using **static** member variables, you should be able to virtually eliminate any need for global variables. The trouble with global variables relative to OOP is that they almost always violate the principle of encapsulation.

Static Member Functions

Member functions may also be declared as **static**. There are several restrictions placed on **static** member functions. They may only directly refer to other **static** members of the class. (Of course, global functions and data may be accessed by **static** member functions.) A **static** member function does not have a **this** pointer. (See Chapter 13 for information on **this**.) There cannot be a **static** and a non-**static** version of the same function. A **static** member function may not be virtual. Finally, they cannot be declared as **const** or **volatile**.

Following is a slightly reworked version of the shared-resource program from the previous section. Notice that **get_resource( )** is now declared as **static**. As the program illustrates, **get_resource( )** may be called either by itself, independently of any object, by using the class name and the scope resolution operator, or in connection with an object.

```
#include <iostream>
using namespace std;

class cl {
  static int resource;
public:
  static int get_resource();
  void free_resource() { resource = 0; }
};

int cl::resource; // define resource

int cl::get_resource()
{
  if(resource) return 0; // resource already in use
  else {
    resource = 1;
    return 1;  // resource allocated to this object
  }
}

int main()
{
  cl ob1, ob2;

  /* get_resource() is static so may be called independent
     of any object. */
  if(cl::get_resource()) cout << 'ob1 has resource\n";

  if(!cl::get_resource()) cout << "ob2 denied resource\n";

  ob1.free_resource();

  if(ob2.get_resource()) // can still call using object syntax
    cout << "ob2 can now use resource\n";

  return 0;
}
```

Actually, **static** member functions have limited applications, but one good use for them is to "preinitialize" private **static** data before any object is actually created. For example, this is a perfectly valid C++ program:

```
#include <iostream>
using namespace std;

class static_type {
  static int i;
public:
  static void init(int x) {i = x;}
  void show() {cout << i;}
};

int static_type::i; // define i

int main()
{
  // init static data before object creation
  static_type::init(100);

  static_type x;
  x.show(); // displays 100

  return 0;
}
```

When Constructors and Destructors Are Executed

As a general rule, an object's constructor is called when the object comes into existence, and an object's destructor is called when the object is destroyed. Precisely when these events occur is discussed here.

A local object's constructor is executed when the object's declaration statement is encountered. The destructors for local objects are executed in the reverse order of the constructor functions.

Global objects have their constructors execute *before* **main( )** begins execution. Global constructors are executed in order of their declaration, within the same file. You cannot know the order of execution of global constructors spread among several files. Global destructors execute in reverse order *after* **main( )** has terminated.

This program illustrates when constructors and destructors are executed:

```
#include <iostream>
using namespace std;
```

```
class myclass {
public:
  int who;
  myclass(int id);
  ~myclass();
} glob_ob1(1), glob_ob2(2);

myclass::myclass(int id)
{
  cout << "Initializing " << id << "\n";
  who = id;
}

myclass::~myclass()
{
  cout << "Destructing " << who << "\n";
}

int main()
{
  myclass local_ob1(3);

  cout << "This will not be first line displayed.\n";

  myclass local_ob2(4);

  return 0;
}
```

It displays this output:

```
Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed.
Initializing 4
Destructing 4
Destructing 3
Destructing 2
Destructing 1
```

One thing: Because of differences between compilers and execution environments, you may or may not see the last two lines of output.

The Scope Resolution Operator

As you know, the :: operator links a class name with a member name in order to tell the compiler what class the member belongs to. However, the scope resolution operator has another related use: it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name. For example, consider this fragment:

```
int i;  // global i

void f()
{
  int i; // local i

  i = 10; // uses local i
  .
  .
  .
}
```

As the comment suggests, the assignment **i = 10** refers to the local **i**. But what if function **f( )** needs to access the global version of **i**? It may do so by preceding the **i** with the :: operator, as shown here.

```
int i;  // global i

void f()
{
  int i; // local i

  ::i = 10; // now refers to global i
  .
  .
  .
}
```

Nested Classes

It is possible to define one class within another. Doing so creates a *nested* class. Since a **class** declaration does, in fact, define a scope, a nested class is valid only within the scope of the enclosing class. Frankly, nested classes are seldom used. Because of C++'s flexible and powerful inheritance mechanism, the need for nested classes is virtually nonexistent.

Local Classes

A class may be defined within a function. For example, this is a valid C++ program:

```
#include <iostream>
using namespace std;

void f();

int main()
{
  f();
  // myclass not known here
  return 0;
}

void f()
{
  class myclass {
    int i;
  public:
    void put_i(int n) { i=n; }
    int get_i() { return i; }
  } ob;

  ob.put_i(10);
  cout << ob.get_i();
}
```

When a class is declared within a function, it is known only to that function and unknown outside of it.

Several restrictions apply to local classes. First, all member functions must be defined within the class declaration. The local class may not use or access local variables of the function in which it is declared (except that a local class has access to **static** local variables declared within the function or those declared as **extern**). It may access type names and enumerators defined by the enclosing function, however. No **static** variables may be declared inside a local class. Because of these restrictions, local classes are not common in C++ programming.

Passing Objects to Functions

Objects may be passed to functions in just the same way that any other type of variable can. Objects are passed to functions through the use of the standard call-by-value mechanism. Although the passing of objects is straightforward, some rather

unexpected events occur that relate to constructors and destructors. To understand why, consider this short program.

```
// Passing an object to a function.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  myclass(int n);
  ~myclass();
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};

myclass::myclass(int n)
{
  i = n;
  cout << "Constructing " << i << "\n";
}

myclass::~myclass()
{
  cout << "Destroying " << i << "\n";
}

void f(myclass ob);

int main()
{
  myclass o(1);

  f(o);
  cout << "This is i in main: ";
  cout << o.get_i() << "\n";

  return 0;
}

void f(myclass ob)
{
  ob.set_i(2);
```

```
  cout << "This is local i: " << ob.get_i();
  cout << "\n";
}
```

This program produces this output:

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

As the output shows, there is one call to the constructor, which occurs when **o** is created in **main( )**, but there are *two* calls to the destructor. Let's see why this is the case.

When an object is passed to a function, a copy of that object is made (and this copy becomes the parameter in the function). This means that a new object comes into existence. When the function terminates, the copy of the argument (i.e., the parameter) is destroyed. This raises two fundamental questions: First, is the object's constructor called when the copy is made? Second, is the object's destructor called when the copy is destroyed? The answers may, at first, surprise you.

When a copy of an argument is made during a function call, the normal constructor is *not* called. Instead, the object's *copy constructor* is called. A copy constructor defines how a copy of an object is made. As explained in Chapter 14, you can explicitly define a copy constructor for a class that you create . However, if a class does not explicitly define a copy constructor, as is the case here, then C++ provides one by default. The default copy constructor creates a bitwise (that is, identical) copy of the object. The reason a bitwise copy is made is easy to understand if you think about it. Since a normal constructor is used to initialize some aspect of an object, it must not be called to make a copy of an already existing object. Such a call would alter the contents of the object. When passing an object to a function, you want to use the current state of the object, not its initial state.

However, when the function terminates and the copy of the object used as an argument is destroyed, the destructor *is* called. This is necessary because the object has gone out of scope. This is why the preceding program had two calls to the destructor. The first was when the parameter to **f( )** went out-of-scope. The second is when **o** inside **main( )** was destroyed when the program ended.

To summarize: When a copy of an object is created to be used as an argument to a function, the normal constructor is not called. Instead, the default copy constructor makes a bit-by-bit identical copy. However, when the copy is destroyed (usually by going out of scope when the function returns), the destructor is called.

Because the default copy constructor creates an exact duplicate of the original, it can, at times, be a source of trouble. Even though objects are passed to functions by means of the normal call-by-value parameter passing mechanism which, in theory,

protects and insulates the calling argument, it is still possible for a side effect to occur that may affect, or even damage, the object used as an argument. For example, if an object used as an argument allocates memory and frees that memory when it is destroyed, then its local copy inside the function will free the same memory when its destructor is called. This will leave the original object damaged and effectively useless. To prevent this type of problem you will need to define the copy operation by creating a copy constructor for the class, as explained in Chapter 14.

Returning Objects

A function may return an object to the caller. For example, this is a valid C++ program:

```
// Returning objects from a function.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};

myclass f();  // return object of type myclass

int main()
{
  myclass o;

  o = f();

  cout << o.get_i() << "\n";

  return 0;
}

myclass f()
{
  myclass x;

  x.set_i(1);
  return x;
}
```

When an object is returned by a function, a temporary object is automatically created that holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed. The destruction of this temporary object may cause unexpected side effects in some situations. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is receiving the return value is still using it. There are ways to overcome this problem that involve overloading the assignment operator (see Chapter 15) and defining a copy constructor (see Chapter 14).

Object Assignment

Assuming that both objects are of the same type, you can assign one object to another. This causes the data of the object on the right side to be copied into the data of the object on the left. For example, this program displays **99**:

```
// Assigning objects.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};

int main()
{
  myclass ob1, ob2;

  ob1.set_i(99);
  ob2 = ob1; // assign data from ob1 to ob2

  cout << "This is ob2's i: " << ob2.get_i();

  return 0;
}
```

By default, all data from one object is assigned to the other by use of a bit-by-bit copy. However, it is possible to overload the assignment operator and define some other assignment procedure (see Chapter 15).