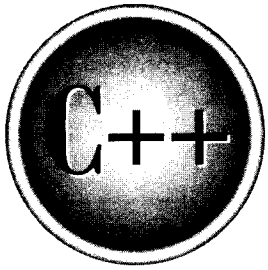
The Complete Reference



Chapter 23

Namespaces, Conversion Functions, and Other Advanced Topics

This chapter describes namespaces and several other advanced features, including conversion functions, explicit constructors, **const** and **volatile** member functions, the **asm** keyword, and linkage specifications. It ends with a discussion of C++'s array-based I/O and a summary of the differences between C and C++.

Namespaces

Namespaces were briefly introduced earlier in this book. They are a relatively recent addition to C++. Their purpose is to localize the names of identifiers to avoid name collisions. The C++ programming environment has seen an explosion of variable, function, and class names. Prior to the invention of namespaces, all of these names competed for slots in the global namespace and many conflicts arose. For example, if your program defined a function called **abs( )**, it could (depending upon its parameter list) override the standard library function **abs( )** because both names would be stored in the global namespace. Name collisions were compounded when two or more third-party libraries were used by the same program. In this case, it was possible—even likely—that a name defined by one library would conflict with the same name defined by the other library. The situation can be particularly troublesome for class names. For example, if your program defines a class call **ThreeDCircle** and a library used by your program defines a class by the same name, a conflict will arise.

The creation of the **namespace** keyword was a response to these problems. Because it localizes the visibility of names declared within it, a namespace allows the same name to be used in different contexts without conflicts arising. Perhaps the most noticeable beneficiary of **namespace** is the C++ standard library. Prior to **namespace**, the entire C++ library was defined within the global namespace (which was, of course, the only namespace). Since the addition of **namespace**, the C++ library is now defined within its own namespace, called **std**, which reduces the chance of name collisions. You can also create your own namespaces within your program to localize the visibility of any names that you think may cause conflicts. This is especially important if you are creating class or function libraries.

Namespace Fundamentals

The **namespace** keyword allows you to partition the global namespace by creating a declarative region. In essence, a **namespace** defines a scope. The general form of **namespace** is shown here:

```
namespace name {
  // declarations
}
```

Anything defined within a **namespace** statement is within the scope of that namespace.

Here is an example of a **namespace**. It localizes the names used to implement a simple countdown counter class. In the namespace are defined the **counter** class, which implements the counter, and the variables **upperbound** and **lowerbound**, which contain the upper and lower bounds that apply to all counters.

```
namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
     int count;
   public:
     counter(int n) {
       if(n <= upperbound) count = n;
       else count = upperbound;
     }

     void reset(int n) {
       if(n <= upperbound) count = n;
     }

     int run() {
       if(count > lowerbound) return count--;
       else return lowerbound;
     }
  };
}
```

Here, **upperbound**, **lowerbound**, and the class **counter** are part of the scope defined by the **CounterNameSpace** namespace.

Inside a namespace, identifiers declared within that namespace can be referred to directly, without any namespace qualification. For example, within **CounterNameSpace**, the **run( )** function can refer directly to **lowerbound** in the statement

```
if(count > lowerbound) return count--;
```

However, since **namespace** defines a scope, you need to use the scope resolution operator to refer to objects declared within a namespace from outside that namespace.

For example, to assign the value 10 to **upperbound** from code outside **CounterNameSpace**, you must use this statement:

```
CounterNameSpace::upperbound = 10;
```

Or to declare an object of type **counter** from outside **CounterNameSpace**, you will use a statement like this:

```
CounterNameSpace::counter ob;
```

In general, to access a member of a namespace from outside its namespace, precede the member's name with the name of the namespace followed by the scope resolution operator.

Here is a program that demonstrates the use of **CounterNameSpace**.

```
// Demonstrate a namespace.
#include <iostream>
using namespace std;

namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
     int count;
   public:
     counter(int n) {
       if(n <= upperbound) count = n;
       else count = upperbound;
     }

     void reset(int n) {
       if(n <= upperbound) count = n;
     }

     int run() {
       if(count > lowerbound) return count--;
       else return lowerbound;
     }
```

```
  };
}

int main()
{
  CounterNameSpace::upperbound = 100;
  CounterNameSpace::lowerbound = 0;

  CounterNameSpace::counter ob1(10);
  int i;

  do {
    i = ob1.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;

  CounterNameSpace::counter ob2(20);

  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;

  ob2.reset(100);
  CounterNameSpace::lowerbound = 90;
  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);

  return 0;
}
```

Notice that the declaration of a **counter** object and the references to **upperbound** and **lowerbound** are qualified by **CounterNameSpace**. However, once an object of type **counter** has been declared, it is not necessary to further qualify it or any of its members. Thus, **ob1.run( )** can be called directly; the namespace has already been resolved.

using

As you can imagine, if your program includes frequent references to the members of a namespace, having to specify the namespace and the scope resolution operator each time you need to refer to one quickly becomes a tedious chore. The **using** statement was invented to alleviate this problem. The **using** statement has these two general forms:

using namespace *name;*
using *name::member;*

In the first form, *name* specifies the name of the namespace you want to access. All of the members defined within the specified namespace are brought into view (i.e., they become part of the current namespace) and may be used without qualification. In the second form, only a specific member of the namespace is made visible. For example, assuming **CounterNameSpace** as shown above, the following **using** statements and assignments are valid.

```
using CounterNameSpace::lowerbound; // only lowerbound is visible
lowerbound = 10; // OK because lowerbound is visible

using namespace CounterNameSpace; // all members are visible
upperbound = 100; // OK because all members are now visible
```

The following program illustrates **using** by reworking the counter example from the previous section.

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

namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
     int count;
   public:
     counter(int n) {
       if(n <= upperbound) count = n;
       else count = upperbound;
     }
```

```
    void reset(int n) {
      if(n <= upperbound) count = n;
    }

    int run() {
      if(count > lowerbound) return count--;
      else return lowerbound;
    }
  };
}

int main()
{
  // use only upperbound from CounterNameSpace
  using CounterNameSpace::upperbound;

  // now, no qualification needed to set upperbound
  upperbound = 100;

  // qualification still needed for lowerbound, etc.
  CounterNameSpace::lowerbound = 0;

  CounterNameSpace::counter ob1(10);
  int i;

  do {
    i = ob1.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;

  // now, use entire CounterNameSpace
  using namespace CounterNameSpace;

  counter ob2(20);

  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > lowerbound);
  cout << endl;

  ob2.reset(100);
```

```
  lowerbound = 90;
  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > lowerbound);

  return 0;
}
```

The program illustrates one other important point: using one namespace does not override another. When you bring a namespace into view, it simply adds its names to whatever other namespaces are currently in effect. Thus, by the end of the program, both **std** and **CounterNameSpace** have been added to the global namespace.

Unnamed Namespaces

There is a special type of namespace, called an *unnamed namespace*, that allows you to create identifiers that are unique within a file. Unnamed namespaces are also called *anonymous namespaces*. They have this general form:

```
namespace {
  // declarations
}
```

Unnamed namespaces allow you to establish unique identifiers that are known only within the scope of a single file. That is, within the file that contains the unnamed namespace, the members of that namespace may be used directly, without qualification. But outside the file, the identifiers are unknown.

Unnamed namespaces eliminate the need for certain uses of the **static** storage class modifier. As explained in Chapter 2, one way to restrict the scope of a global name to the file in which it is declared is to use **static**. For example, consider the following two files that are part of the same program.

File One

```
static int k;
void f1() {
  k = 99; // OK
}
```

File Two

```
extern int k;
void f2() {
  k = 10; // error
}
```

Because **k** is defined in File One, it may be used in File One. In File Two, **k** is specified as **extern**, which means that its name and type are known but that **k** itself is not actually

defined. When these two files are linked, the attempt to use **k** within File Two results in an error because there is no definition for **k**. By preceding **k** with **static** in File One, its scope is restricted to that file and it is not available to File Two.

While the use of **static** global declarations is still allowed in C++, a better way to accomplish the same effect is to use an unnamed namespace. For example:

File One

```
namespace {
  int k;
}
void f1() {
  k = 99; // OK
}
```

File Two

```
extern int k;
void f2() {
  k = 10; // error
}
```

Here, **k** is also restricted to File One. The use of the unnamed namespace rather than **static** is recommended for new code.

Some Namespace Options

There may be more than one namespace declaration of the same name. This allows a namespace to be split over several files or even separated within the same file. For example:

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

namespace NS {
  int i;
}

// ...

namespace NS {
  int j;
}

int main()
{
  NS::i = NS::j = 10;

  // refer to NS specifically
```

```
  cout << NS::i * NS::j << "\n";

  // use NS namespace
  using namespace NS;

  cout << i * j;

  return 0;
}
```

This program produces the following output:

```
100
100
```

Here, **NS** is split into two pieces. However, the contents of each piece are still within the same namespace, that is, **NS**.

A namespace must be declared outside of all other scopes. This means that you cannot declare namespaces that are localized to a function, for example. There is, however, one exception: a namespace can be nested within another. Consider this program:

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

namespace NS1 {
  int i;
  namespace NS2 { // a nested namespace
    int j;
  }
}

int main()
{
  NS1::i = 19;
  // NS2::j = 10; Error, NS2 is not in view
  NS1::NS2::j = 10; // this is right

  cout << NS1::i << " "<<  NS1::NS2::j << "\n";

  // use NS1
```

```
  using namespace NS1;

  /* Now that NS1 is in view, NS2 can be used to
     refer to j. */
  cout << i * NS2::j;

  return 0;
}
```

This program produces the following output:

```
19 10
190
```

Here, the namespace **NS2** is nested within **NS1**. Thus, when the program begins, to refer to **j**, you must qualify it with both the **NS1** and **NS2** namespaces. **NS2** by itself is insufficient. After the statement

```
using namespace NS1;
```

executes, you can refer directly to **NS2** since the **using** statement brings **NS1** into view.

Typically, you will not need to create namespaces for most small to medium-sized programs. However, if you will be creating libraries of reusable code or it you want to ensure the widest portability, then consider wrapping your code within a namespace.

The std Namespace

Standard C++ defines its entire library in its own namespace called **std**. This is the reason that most of the programs in this book include the following statement:

```
using namespace std;
```

This causes the **std** namespace to be brought into the current namespace, which gives you direct access to the names of the functions and classes defined within the library without having to qualify each one with **std::**.

Of course, you can explicitly qualify each name with **std::** if you like. For example, the following program does not bring the library into the global namespace.

```
// Use explicit std:: qualification.
```

```
#include <iostream>

int main()
{
  int val;

  std::cout << "Enter a number: ";

  std::cin >> val;

  std::cout << "This is your number: ";
  std::cout << std::hex << val;

  return 0;
}
```

Here, **cout**, **cin**, and the manipulator **hex** are explicitly qualified by their namespace. That is, to write to standard output, you must specify **std::cout**; to read from standard input, you must use **std::cin**; and the hex manipulator must be referred to as **std::hex**.

You may not want to bring the standard C++ library into the global namespace if your program will be making only limited use of it. However, if your program contains hundreds of references to library names, then including **std** in the current namespace is far easier than qualifying each name individually.

If you are using only a few names from the standard library, it may make more sense to specify a **using** statement for each individually. The advantage to this approach is that you can still use those names without an **std::** qualification, but you will not be bringing the entire standard library into the global namespace. For example:

```
// Bring only a few names into the global namespace.
#include <iostream>

// gain access to cout, cin, and hex
using std::cout;
using std::cin;
using std::hex;

int main()
{
  int val;
```

```
  cout << "Enter a number: ";

  cin >> val;
  cout << "This is your number: ";
  cout << hex << val;
  return 0;
}
```

Here, **cin**, **cout**, and **hex** may be used directly, but the rest of the **std** namespace has not been brought into view.

As explained, the original C++ library was defined in the global namespace. If you will be converting older C++ programs, then you will need to either include a **using namespace std** statement or qualify each reference to a library member with **std::**. This is especially important if you are replacing old **.H** header files with the new-style headers. Remember, the old **.H** headers put their contents into the global namespace; the new-style headers put their contents into the **std** namespace.

Creating Conversion Functions

In some situations, you will want to use an object of a class in an expression involving other types of data. Sometimes, overloaded operator functions can provide the means of doing this. However, in other cases, what you want is a simple type conversion from the class type to the target type. To handle these cases, C++ allows you to create custom *conversion functions*. A conversion function converts your class into a type compatible with that of the rest of the expression. The general format of a type conversion function is

operator *type*( ) { return *value*; }

Here, *type* is the target type that you are converting your class to, and *value* is the value of the class after conversion. Conversion functions return data of type *type*, and no other return type specifier is allowed. Also, no parameters may be included. A conversion function must be a member of the class for which it is defined. Conversion functions are inherited and they may be virtual.

The following illustration of a conversion function uses the **stack** class first developed in Chapter 11. Suppose that you want to be able to use objects of type **stack** within an integer expression. Further, suppose that the value of a **stack** object used in an integer expression is the number of values currently on the stack. (You might want

to do something like this if, for example, you are using **stack** objects in a simulation and are monitoring how quickly the stacks fill up.) One way to approach this is to convert an object of type **stack** into an integer that represents the number of items on the stack. To accomplish this, you use a conversion function that looks like this:

```
operator int() { return tos; }
```

Here is a program that illustrates how the conversion function works:

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

const int SIZE=100;

// this creates the class stack
class stack {
  int stck[SIZE];
  int tos;
public:
  stack() { tos=0; }
  void push(int i);
  int pop(void);
  operator int() { return tos; } // conversion of stack to int
};

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Stack underflow.\n";
    return 0;
  }
```

```
  tos--;
  return stck[tos];
}

int main()
{
  stack stck;
  int i, j;

  for(i=0; i<20; i++)  stck.push(i);

  j = stck; // convert to integer

  cout << j << " items on stack.\n";

  cout << SIZE - stck << " spaces open.\n";
  return 0;
}
```

This program displays this output:

```
20 items on stack.
80 spaces open.
```

As the program illustrates, when a **stack** object is used in an integer expression, such as **j = stck**, the conversion function is applied to the object. In this specific case, the conversion function returns the value 20. Also, when **stck** is subtracted from **SIZE**, the conversion function is also called.

Here is another example of a conversion function. This program creates a class called **pwr( )** that stores and computes the outcome of some number raised to some power. It stores the result as a **double**. By supplying a conversion function to type **double** and returning the result, you can use objects of type **pwr** in expressions involving other **double** values.

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

class pwr {
  double b;
  int e;
  double val;
```

```
public:
  pwr(double base, int exp);
  pwr operator+(pwr o) {
    double base;
    int exp;
    base = b + o.b;
    exp = e + o.e;

    pwr temp(base, exp);
    return temp;
  }
  operator double() { return val; } // convert to double
};

pwr::pwr(double base, int exp)
{
  b = base;
  e = exp;
  val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--) val = val * b;
}

int main()
{
  pwr x(4.0, 2);
  double a;

  a = x; // convert to double
  cout << x + 100.2; // convert x to double and add 100.2
  cout << "\n";

  pwr y(3.3, 3), z(0, 0);

  z = x + y;  // no conversion
  a = z;  // convert to double
  cout << a;

  return 0;
}
```

The output from the program is shown here.

```
116.2
20730.7
```

As you can see, when **x** is used in the expression **x + 100.2**, the conversion function is used to produce the **double** value. Notice also that in the expression **x + y**, no conversion is applied because the expression involves only objects of type **pwr**.

As you can infer from the foregoing examples, there are many situations in which it is beneficial to create a conversion function for a class. Often, conversion functions provide a more natural syntax to be used when class objects are mixed with the built-in types. Specifically, in the case of the **pwr** class, the availability of the conversion to **double** makes using objects of that class in "normal" mathematical expressions both easier to program and easier to understand.

You can create different conversion functions to meet different needs. You could define another that converts to **long**, for example. Each will be applied automatically as determined by the type of each expression.

const Member Functions and mutable

Class member functions may be declared as **const**, which causes **this** to be treated as a **const** pointer. Thus, that function cannot modify the object that invokes it. Also, a **const** object may not invoke a non-**const** member function. However, a **const** member function can be called by either **const** or non-**const** objects.

To specify a member function as **const**, use the form shown in the following example.

```
class X {
  int some_var;
public:
  int f1() const; // const member function
};
```

As you can see, the **const** follows the function's parameter declaration.

The purpose of declaring a member function as **const** is to prevent it from modifying the object that invokes it. For example, consider the following program.

```
/*
   Demonstrate const member functions.
```

```
   This program won't compile.
*/
#include <iostream>
using namespace std;

class Demo {
  int i;
public:
  int geti() const {
    return i; // ok
  }

  void seti(int x) const {
    i = x; // error!
  }
};

int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();

  return 0;
}
```

This program will not compile because **seti( )** is declared as **const**. This means that it is not allowed to modify the invoking object. Since it attempts to change **i**, the program is in error. In contrast, since **geti( )** does not attempt to modify **i**, it is perfectly acceptable.

Sometimes there will be one or more members of a class that you want a **const** function to be able to modify even though you don't want the function to be able to modify any of its other members. You can accomplish this through the use of **mutable**. It overrides **const**ness. That is, a **mutable** member can be modified by a **const** member function. For example:

```
// Demonstrate mutable.
#include <iostream>
using namespace std;

class Demo {
```

```
  mutable int i;
  int j;
public:
  int geti() const {
    return i; // ok
  }

  void seti(int x) const {
    i = x; // now, OK.
  }

/* The following function won't compile.
  void setj(int x) const {
    j = x; // Still Wrong!
  }
*/
};

int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();

  return 0;
}
```

Here, **i** is specified as **mutable**, so it may be changed by the **seti( )** function. However, **j** is not **mutable** and **setj( )** is unable to modify its value.

Volatile Member Functions

Class member functions may be declared as **volatile**, which causes **this** to be treated as a **volatile** pointer. To specify a member function as **volatile**, use the form shown in the following example:

```
class X {
public:
  void f2(int a) volatile; // volatile member function
};
```

Explicit Constructors

As explained in Chapter 12, any time you have a constructor that requires only one argument, you can use either *ob*(*x*) or *ob* = *x* 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. But there may be times when you do not want this automatic conversion to take place. For this purpose, C++ defines the keyword **explicit**. To understand its effects, consider the following program.

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

class myclass {
  int a;
public:
  myclass(int x) { a = x; }
  int geta() { return a; }
};

int main()
{
  myclass ob = 4; // automatically converted into myclass(4)

  cout << ob.geta();

  return 0;
}
```

Here, the constructor for **myclass** takes one parameter. Pay special attention to how **ob** is declared in **main( )**. The statement

```
myclass ob = 4; // automatically converted into myclass(4)
```

is automatically converted into a call to the **myclass** constructor with 4 being the argument. That is, the preceding statement is handled by the compiler as if it were written like this:

```
myclass ob(4);
```

If you do not want this implicit conversion to be made, you can prevent it by using **explicit**. The **explicit** specifier applies only to constructors. A constructor specified as **explicit** will only be used when an initialization uses the normal constructor syntax. It will not perform any automatic conversion. For example, by declaring the **myclass** constructor as **explicit**, the automatic conversion will not be supplied. Here is **myclass( )** declared as **explicit**.

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

class myclass {
  int a;
public:
  explicit myclass(int x) { a = x; }
  int geta() { return a; }
};
```

Now, only constructors of the form

```
myclass ob(4);
```

will be allowed and a statement like

```
myclass ob = 4; // now in error
```

will be invalid.

The Member Initialization Syntax

Example code throughout the preceding chapters has initialized member variables inside the constructor for their class. For example, the following program contains the **MyClass** class, which has two integer data members called **numA** and **numB**. These member variables are initialized inside **MyClass**' constructor.

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

```
class MyClass {
  int numA;
  int numB;
public:
  /* Initialize numA and numB inside the MyClass constructor
     using normal syntax. */
  MyClass(int x, int y) {
    numA = x;
    numB = y;
  }

  int getNumA() { return numA; }
  int getNumB() { return numB; }
};

int main()
{
  MyClass ob1(7, 9), ob2(5, 2);

  cout << "Values in ob1 are " << ob1.getNumB() <<
          " and " << ob1.getNumA() << endl;

  cout << "Values in ob2 are " << ob2.getNumB() <<
          " and " << ob2.getNumA() << endl;

  return 0;
}
```

Assigning initial values to member variables **numA** and **numB** inside the constructor, as **MyClass( )** does, is the usual approach, and is the way that member initialization is accomplished for many, many classes. However, this approach won't work in all cases. For example, if **numA** and **numB** were specified as **const**, like this

```
class MyClass {
  const int numA; // const member
  const int numB; // const member
```

then they could not be given values by the **MyClass** constructor because **const** variables must be initialized and cannot be assigned values after the fact. Similar problems arise when using reference members, which must be initialized, and when using class members that don't have default constructors. To solve these types of

problems, C++ supports an alternative member initialization syntax, which is used to give a class member an initial value when an object of the class is created.

The member initialization syntax is similar to that used to call a base class constructor. Here is the general form:

```
constructor(arg-list) : member1(initializer),
                        member2(initializer),
                        // ...
                        memberN(initializer)
{
  // body of constructor
}
```

The members that you want to initialize are specified before the body of the constructor, separated from the constructor's name and argument list by a colon. You can mix calls to base class constructors with member initializations in the same list.

Here is **MyClass** rewritten so that **numA** and **numB** are **const** members that are given values using the member initialization syntax.

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

class MyClass {
  const int numA; // const member
  const int numB; // const member
public:
  // Initialize numA and numB using initialization syntax.
  MyClass(int x, int y) : numA(x), numB(y) { }

  int getNumA() { return numA; }
  int getNumB() { return numB; }
};

int main()
{
  MyClass ob1(7, 9), ob2(5, 2);

  cout << "Values in ob1 are " << ob1.getNumB() <<
          " and " << ob1.getNumA() << endl;

  cout << "Values in ob2 are " << ob2.getNumB() <<
          " and " << ob2.getNumA() << endl;

  return 0;
}
```

Notice how **numA** and **numB** are initialized by this statement:

```
MyClass(int x, int y) : numA(x), numB(y) { }
```

Here, **numA** is initialized with the value passed in **x**, and **numB** is initialized with the value passed in **y**. Even though **numA** and **numB** are now **const**, they can be given initial values when a **MyClass** object is created because the member initialization syntax is used.

The member initialization syntax is especially useful when you have a member that is of a class type for which there is no default constructor. To understand why, consider this slightly different version of **MyClass** that attempts to store the two integer values in an object of type **IntPair**. Because **IntPair** has no default constructor, this program is in error and won't compile.

```
// This program is in error and won't compile.
#include <iostream>
using namespace std;

class IntPair {
public:
  int a;
  int b;

  IntPair(int i, int j) : a(i), b(j) { }
};

class MyClass {
  IntPair nums; // Error: no default constructor for IntPair!
public:
  // This won't work!
  MyClass(int x, int y) {
    nums.a = x;
    nums.b = y;
  }

  int getNumA() { return nums.a; }
  int getNumB() { return nums.b; }
};

int main()
{
  MyClass ob1(7, 9), ob2(5, 2);
```

```
  cout << "Values in ob1 are " << ob1.getNumB() <<
          " and " << ob1.getNumA() << endl;

  cout << "Values in ob2 are " << ob2.getNumB() <<
          " and " << ob2.getNumA() << endl;

  return 0;
}
```

The reason that the program won't compile is that **IntPair** has only one constructor and it requires two arguments. However, **nums** is declared inside **MyClass** without any parameters and the values of **a** and **b** are set inside **MyClass**' constructor. This causes an error because it implies that a default (i.e., parameterless) constructor is available to initially create an **IntPair** object, which is not the case.

To fix this problem, you could add a default constructor to **IntPair**. However, this only works if you have access to the source code for the class, which might not always be the case. A better solution is to use the member initialization syntax, as shown in this correct version of the program.

```
// This program is now correct.
#include <iostream>
using namespace std;

class IntPair {
public:
  int a;
  int b;

  IntPair(int i, int j) : a(i), b(j) { }
};

class MyClass {
  IntPair nums; // now OK
public:
  // Initialize nums object using initialization syntax.
  MyClass(int x, int y) : nums(x,y) { }

  int getNumA() { return nums.a; }
  int getNumB() { return nums.b; }
};

int main()
```

```
{
  MyClass ob1(7, 9), ob2(5, 2);

  cout << "Values in ob1 are " << ob1.getNumB() <<
          " and " << ob1.getNumA() << endl;

  cout << "Values in ob2 are " << ob2.getNumB() <<
          " and " << ob2.getNumA() << endl;

  return 0;
}
```

Here, **nums** is given an initial value when a **MyClass** object is created. Thus, no default constructor is required.

One last point: Class members are constructed and initialized in the order in which they are declared in a class, not in the order in which their initializers occur.

Using the asm Keyword

While C++ is a comprehensive and powerful programming language, there are a few highly specialized situations that it cannot handle. (For example, there is no C++ statement that disables interrupts.) To accommodate special situations, C++ provides a "trap door" that allows you to drop into assembly code at any time, bypassing the C++ compiler entirely. This "trap door" is the **asm** statement. Using **asm**, you can embed assembly language directly into your C++ program. This assembly code is compiled without any modification, and it becomes part of your program's code at the point at which the **asm** statement occurs.

The general form of the **asm** keyword is shown here:

asm ("*op-code*");

where *op-code* is the assembly language instruction that will be embedded in your program. However, several compilers also allow the following forms of **asm**:

asm *instruction* ;
asm *instruction newline*
asm {
  instruction sequence
}

Here, *instruction* is any valid assembly language instruction. Because of the implementation- specific nature of **asm**, you must check the documentation that came with your compiler for details.

At the time of this writing, Microsoft's Visual C++ uses **__asm** for embedding assembly code. It is otherwise similar to **asm**.

Here is a simple (and fairly "safe") example that uses the **asm** keyword:

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

int main()
{
  asm int 5; // generate intertupt 5

  return 0;
}
```

When run under DOS, this program generates an INT 5 instruction, which invokes the print-screen function.

Caution *A thorough working knowledge of assembly language programming is required for using the* ***asm*** *statement. If you are not proficient with assembly language, it is best to avoid using* ***asm*** *because very nasty errors may result.*

Linkage Specification

In C++ you can specify how a function is linked into your program. By default, functions are linked as C++ functions. However, by using a *linkage specification*, you can cause a function to be linked for a different type of language. The general form of a linkage specifier is

extern *"language" function-prototype*

where *language* denotes the desired language. All C++ compilers support both C and C++ linkage. Some will also allow linkage specifiers for Fortran, Pascal, or BASIC. (You will need to check the documentation for your compiler.)

This program causes **myCfunc( )** to be linked as a C function.

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

```
extern "C" void myCfunc();

int main()
{
  myCfunc();

  return 0;
}

// This will link as a C function.
void myCfunc()
{
  cout << "This links as a C function.\n";
}
```

Note *The **extern** keyword is a necessary part of the linkage specification. Further, the linkage specification must be global; it cannot be used inside of a function.*

You can specify more than one function at a time using this form of the linkage specification:

```
extern "language" {
  prototypes
}
```

Array-Based I/O

In addition to console and file I/O, C++'s stream-based I/O system allows *array-based I/O*. Array-based I/O uses a character array as either the input device, the output device, or both. Array-based I/O is performed through normal C++ streams. In fact, everything you already know about C++ I/O is applicable to array-based I/O. The only thing that makes array-based I/O unique is that the device linked to the stream is an array of characters. Streams that are linked to character arrays are commonly referred to as **char** * streams. To use array-based I/O in your programs, you must include **<strstream>**.

Note *The character-based stream classes described in this section are deprecated by Standard C++. This means that they are still valid, but not recommended for new code. This brief discussion is included for the benefit of readers working on older code.*

The Array-Based Classes

The array-based I/O classes are **istrstream**, **ostrstream**, and **strstream**. These classes are used to create input, output, and input/output streams, respectively. Further, the **istrstream** class is derived from **istream**, the **ostrstream** class is derived from **ostream**, and **strstream** has **iostream** as a base class. Therefore, all array-based classes are indirectly derived from **ios** and have access to the same member functions that the "normal" I/O classes do.

Creating an Array-Based Output Stream

To perform output to an array, you must link that array to a stream using this **ostrstream** constructor:

ostrstream *ostr*(char **buf*, streamsize *size*, openmode *mode*=ios::out);

Here, *buf* is a pointer to the array that will be used to collect characters written to the stream *ostr*. The size of the array is passed in the *size* parameter. By default, the stream is opened for normal output, but you can OR various other options with it to create the mode that you need. For example, you might include **ios::app** to cause output to be written at the end of any information already contained in the array. For most purposes, *mode* will be allowed to default.

Once you have opened an array-based output stream, all output to that stream is put into the array. However, no output will be written outside the bounds of the array. Attempting to do so will result in an error.

Here is a simple program that demonstrates an array-based output stream.

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

int main()
{
  char str[80];

  ostrstream outs(str, sizeof(str));

  outs << "C++ array-based I/O. ";
  outs << 1024 << hex << " ";
  outs.setf(ios::showbase);
  outs << 100 << ' ' << 99.789 << ends;
```

```
  cout << str;  // display string on console

  return 0;
}
```

This program displays the following:

```
C++ array-based I/O. 1024 0x64 99.789
```

Keep in mind that **outs** is a stream like any other stream; it has the same capabilities as any other type of stream that you have seen earlier. The only difference is that the device that it is linked to is a character array. Because **outs** is a stream, manipulators like **hex** and **ends** are perfectly valid. **ostream** member functions, such as **setf( )**, are also available for use.

This program manually null terminates the array by using the **ends** manipulator. Whether the array will be automatically null terminated or not depends on the implementation, so it is best to perform null termination manually if it is important to your application.

You can determine how many characters are in the output array by calling the **pcount( )** member function. It has this prototype:

```
streamsize pcount( );
```

The number returned by **pcount( )** also includes the null terminator, if it exists.

The following program demonstrates **pcount( )**. It reports that **outs** contains 18 characters: 17 characters plus the null terminator.

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

int main()
{
  char str[80];

  ostrstream outs(str, sizeof(str));

  outs << "abcdefg ";
  outs << 27 << " "  << 890.23;
  outs << ends;  // null terminate
```

```
  cout << outs.pcount(); // display how many chars in outs

  cout << ' " << str;

  return 0;
}
```

Using an Array as Input

To link an input stream to an array, use this **istrstream** constructor:

istrstream *istr*(const char **buf*);

Here, *buf* is a pointer to the array that will be used as a source of characters each time input is performed on the stream *istr*. The contents of the array pointed to by *buf* must be null terminated. However, the null terminator is never read from the array.

Here is a sample program that uses a string as input.

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

int main()
{
  char s[] = "10 Hello 0x75 42.73 OK";

  istrstream ins(s);

  int i;
  char str[80];
  float f;

  // reading: 10 Hello
  ins >> i;
  ins >> str;
  cout << i << " " << str << endl;

  // reading 0x75 42.73 OK
  ins >> hex >> i;
```

```
  ins >> f;
  ins >> str;

  cout << hex << i << " " << f << " " << str;

  return 0;
}
```

If you want only part of a string to be used for input, use this form of the **istrstream** constructor:

istrstream *istr*(const char **buf*, streamsize *size*);

Here, only the first *size* elements of the array pointed to by *buf* will be used. This string need not be null terminated, since it is the value of *size* that determines the size of the string.

Streams linked to memory behave just like those linked to other devices. For example, the following program demonstrates how the contents of any text array can be read. When the end of the array (same as end-of-file) is reached, **ins** will be false.

```
/* This program shows how to read the contents of any
   array that contains text. */
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char s[] = "10.23 this is a test <<>><<?!\n";

  istrstream ins(s);

  char ch;

  /* This will read and display the contents
     of any text array. */

  ins.unsetf(ios::skipws); // don't skip spaces
  while (ins) { // false when end of array is reached
    ins >> ch;
    cout << ch;
```

```
  }

  return 0;
}
```

Input/Output Array-Based Streams

To create an array-based stream that can perform both input and output, use this **strstream** constructor function:

strstream *iostr*(char **buf*, streamsize *size*, openmode *mode* = ios::in | ios::out);

Here, *buf* points to the string that will be used for I/O operations. The value of *size* specifies the size of the array. The value of *mode* determines how the stream *iostr* operates. For normal input/output operations, *mode* will be **ios::in | ios::out**. For input, the array must be null terminated.

Here is a program that uses an array to perform both input and output.

```
// Perform both input and output.
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char iostr[80];

  strstream strio(iostr, sizeof(iostr), ios::in | ios::out);

  int a, b;
  char str[80];

  strio << "10 20 testing ";
  strio >> a >> b >> str;
  cout << a << " " << b << " " << str << endl;

  return 0;
}
```

This program first writes **10 20 testing** to the array and then reads it back in again.

Using Dynamic Arrays

In the preceding examples, when you linked a stream to an output array, the array and its size were passed to the **ostrstream** constructor. This approach is fine as long as you know the maximum number of characters that you will be outputting to the array. However, what if you don't know how large the output array needs to be? The solution to this problem is to use a second form of the **ostrstream** constructor, shown here:

```
ostrstream( );
```

When this constructor is used, **ostrstream** creates and maintains a dynamically allocated array, which automatically grows in length to accommodate the output that it must store.

To access the dynamically allocated array, you must use a second function, called **str( )**, which has this prototype:

```
char *str( );
```

This function "freezes" the array and returns a pointer to it. You use the pointer returned by **str( )** to access the dynamic array as a string. Once a dynamic array is frozen, it cannot be used for output again unless its is unfrozen (see below). Therefore, you will not want to freeze the array until you are through outputting characters to it.

Here is a program that uses a dynamic output array.

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

int main()
{
  char *p;

  ostrstream outs;  // dynamically allocate array

  outs << "C++ array-based I/O ";
  outs << -10 << hex << " ";
  outs.setf(ios::showbase);
  outs << 100 << ends;

  p = outs.str(); // Freeze dynamic buffer and return
                  // pointer to it.

  cout << p;
```

```
  return 0;
}
```

You can also use dynamic I/O arrays with the **strstream** class, which can perform both input and output on an array.

It is possible to freeze or unfreeze a dynamic array by calling the **freeze( )** function. Its prototype is shown here:

void freeze(bool *action* = true);

If *action* is true, the array is frozen. If *action* is false, the array is unfrozen.

Using Binary I/O with Array-Based Streams

Remember that array-based I/O has all of the functionality and capability of "normal" I/O. Therefore, arrays linked to array-based streams can also contain binary information. When reading binary information, you may need to use the **eof( )** function to determine when the end of the array has been reached. For example, the following program shows how to read the contents of any array—binary or text—using the function **get( )**.

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

int main()
{
  char *p = "this is a test\1\2\3\4\5\6\7";

  istrstream ins(p);

  char ch;

  // read and display binary info
  while (!ins.eof()) {
    ins.get(ch);
    cout << hex << (int) ch << ' ';

  }
  return 0;
}
```

In this example, the values formed by \1\2\3, and so on are nonprinting values.

To output binary characters, use the **put( )** function. If you need to read buffers of binary data, you can use the **read( )** member function. To write buffers of binary data, use the **write( )** function.

Summarizing the Differences Between C and C++

For the most part, Standard C++ is a superset of Standard C, and virtually all C programs are also C++ programs. However, a few differences do exist, and these have been discussed throughout Parts One and Two of this book. The most important are summarized here.

In C++, local variables can be declared anywhere within a block. In C, they must be declared at the start of a block, before any "action" statements occur. (C99 has removed this restriction.)

In C, a function declared like

```
int f();
```

says *nothing* about any parameters to that function. That is, when there is nothing specified between the parentheses following the function's name, in C this means that nothing is being stated, one way or the other, about any parameters to that function. It might have parameters, or it might not. However, in C++, a function declaration like this means that the function does *not* have parameters. That is, in C++, these two declarations are equivalent:

```
int f();

int f(void);
```

In C++, **void** in a parameter list is optional. Many C++ programmers include **void** as a means of making it completely clear to anyone reading the program that a function does not have any parameters, but this is technically unnecessary.

In C++, all functions must be prototyped. This is an option in C (although good programming practice suggests full prototyping be used in a C program).

A small but potentially important difference between C and C++ is that in C, a character constant is automatically elevated to an integer. In C++, it is not.

In C, it is not an error to declare a global variable several times, even though this is bad programming practice. In C++, it is an error.

In C, an identifier will have at least 31 significant characters. In C++, all characters are significant. However, from a practical point of view, extremely long identifiers are unwieldy and seldom needed.

In C, although it is unusual, you can call **main( )** from within your program. This is not allowed by C++.

In C, you cannot take the address of a **register** variable. In C++, this is allowed.

In C, if no type specifier is present in some types of declaration statements, the type **int** is assumed. This "default-to-int" rule no longer applies to C++. (C99 also drops the "default-to-int" rule.)