

11

Templates and Exception Handling

chapter objectives

11.1 Generic functions

11.2 Generic classes

11.3 Exception handling

11.4 More about exception handling

11.5 Handling exceptions thrown by **new**

THIS chapter discusses two of C++'s most important high-level features: *templates* and *exception handling*. While neither was part of the original specification for C++, both were added several years ago and are defined by Standard C++. They are supported by all modern C++ compilers. These two features help you achieve two of the most elusive goals in programming: the creation of reusable and resilient code.

Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data that is operated upon is specified as a parameter. This allows you to use one function or class with several different types of data without having to explicitly recode a specific version for each different data type. Thus, templates allow you to create reusable code. Both generic functions and generic classes are discussed here.

Exception handling is the subsystem of C++ that allows you to handle errors that occur at run time in a structured and controlled manner. With C++ exception handling, your program can automatically invoke an error handling routine when an error occurs. The principal advantage of exception handling is that it automates much of the error handling code that previously had to be coded "by hand" in any large program. The proper use of exception handling helps you to create resilient code.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. What is a virtual function?
2. What is a pure virtual function? If a class declaration contains a pure virtual function, what is that class called, and what restrictions apply to its usage?
3. Run-time polymorphism is achieved through the use of _____ functions and _____ class pointers. (Fill in the missing words.)

4. If, in a class hierarchy, a derived class neglects to override a (non-pure) virtual function, what happens when an object of that derived class calls that function?
5. What is the main advantage of run-time polymorphism? What is its potential disadvantage?

11.1 GENERIC FUNCTIONS

A generic function defines a general set of operations that will be applied to various types of data. A generic function has the type of data that it will operate upon passed to it as a parameter. Using this mechanism, the same general procedure can be applied to a wide range of data. As you know, many algorithms are logically the same no matter what type of data is being operated upon. For example, the Quicksort algorithm is the same whether it is applied to an array of integers or an array of **float**s. It is just that the type of the data being sorted is different. By creating a generic function, you can define, independent of any data, the nature of the algorithm. Once this is done, the compiler automatically generates the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function you are creating a function that can automatically overload itself.

A generic function is created using the keyword **template**. The normal meaning of the word *template* accurately reflects the keyword's use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed. The general form of a template function definition is shown here:

```
template <class Ttype> ret-type func-name( parameter list )
{
  // body of function
}
```

Here *Ttype* is a placeholder name for a data type used by the function. This name can be used within the function definition. However, it is only a placeholder; the compiler will automatically replace this placeholder with an actual data type when it creates a specific version of the function.

Although the use of the keyword **class** to specify a generic type in a template declaration is traditional, you can also use the keyword **typename**.

EXAMPLES

1. The following program creates a generic function that swaps the values of the two variables it is called with. Because the general process of exchanging two values is independent of the type of the variables, this process is a good choice to be made into a generic function.

```
// Function template example.
#include <iostream>
using namespace std;

// This is a function template.
template <class X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}

int main()
{
  int i=10, j=20;
  float x=10.1, y=23.3;

  cout << "Original i, j: " << i << ' ' << j << endl;
  cout << "Original x, y: " << x << ' ' << y << endl;

  swapargs(i, j); // swap integers
  swapargs(x, y); // swap floats

  cout << "Swapped i, j: " << i << ' ' << j << endl;
  cout << "Swapped x, y: " << x << ' ' << y << endl;

  return 0;
}
```

The keyword **template** is used to define a generic function. The line

```
template <class X> void swapargs(X &a, X &b)
```

tells the compiler two things: that a template is being created and that a generic definition is beginning. Here **X** is a generic type that is used as a placeholder. After the **template** portion, the function **swapargs( )** is declared, using **X** as the data type of the values that will be swapped. In **main( )**, the **swapargs( )** function is called using two different types of data: integers and **floats**. Because **swapargs( )** is a generic function, the compiler automatically creates two versions of **swapargs( )**—one that will exchange integer values and one that will exchange floating-point values. You should compile and try this program now.

Here are some other terms that are sometimes used when templates are discussed, and that you might encounter in other C++ literature. First, a generic function (that is, a function definition preceded by a **template** statement) is also called a *template function*. When the compiler creates a specific version of this function, it is said to have created a *generated function*. The act of generating a function is referred to as *instantiating* it. Put differently, a generated function is a specific instance of a template function.

2. The **template** portion of a generic function definition does not have to be on the same line as the function's name. For example, the following is another way to format the **swapargs( )** function:

```
template <class X>
void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

If you use this form, it is important to understand that no other statements can occur between the **template** statement

and the start of the generic function definition. For example, the following fragment will not compile:

```
// This will not compile.
template <class X>
int i; // this is an error
void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

As the comments imply, the **template** specification must directly precede the rest of the function definition.

3. As mentioned, instead of using the keyword **class**, you can use the keyword **typename** to specify a generic type in a template definition. For example, here is another way to declare the **swapargs( )** function.

```
// Use typename.
template <typename X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

The **typename** keyword can also be used to specify an unknown type within a template, but this use is beyond the scope of this book.

4. You can define more than one generic data type with the **template** statement, using a comma-separated list. For example, this program creates a generic function that has two generic types:

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

template <class type1, class type2>
```

```
void myfunc(type1 x, type2 y)
{
  cout << x << ' ' << y << endl;
}

int main()
{
  myfunc(10, "hi");

  myfunc(0.23, 10L);

  return 0;
}
```

In this example, the placeholder types **type1** and **type2** are replaced by the compiler with the data types **int** and **char *** and **double** and **long**, respectively, when the compiler generates the specific instances of **myfunc( )**.



When you create a generic function, you are, in essence, allowing the compiler to generate as many different versions of that function as necessary to handle the various ways that your program calls that function.

Generic functions are similar to overloaded functions except that they are more restrictive. When functions are overloaded, you can have different actions performed within the body of each function. But a generic function must perform the same general action for all versions. For example, the following overloaded functions *cannot* be replaced by a generic function because they do not do the same thing:

```
void outdata(int i)
{
  cout << i;
}

void outdata(double d)
{
  cout << setprecision(10) << setfill('#');
  cout << d;
```

```
  cout << setprecision(6) << setfill(' ');
}
```

6. Even though a template function overloads itself as needed, you can explicitly overload one, too. If you overload a generic function, that overloaded function overrides (or "hides") the generic function relative to that specific version. For example, consider this version of Example 1:

```
// Overriding a template function.
#include <iostream>
using namespace std;

template <class X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}

// This overrides the generic version of swapargs().
void swapargs(int a, int b)
{
  cout << "this is inside swapargs(int,int)\n";
}

int main()
{
  int i=10, j=20;
  float x=10.1, y=23.3;


  cout << "Original i, j: " << i << ' ' << j << endl;
  cout << "Original x, y: " << x << ' ' << y << endl;

  swapargs(i, j); // calls overloaded swapargs()
  swapargs(x, y); // swap floats

  cout << "Swapped i, j: " << i << ' ' << j << endl;
  cout << "Swapped x, y: " << x << ' ' << y << endl;
```

```
  return 0;
}
```

As the comments indicate, when **swapargs(i, j)** is called, it invokes the explicitly overloaded version of **swapargs( )** defined in the program. Thus, the compiler does not generate this version of the generic **swapargs( )** function because the generic function is overridden by the explicit overloading.

Manual overloading of a template, as shown in this example, allows you to tailor a version of a generic function to accommodate a special situation. However, in general, if you need to have different versions of a function for different data types, you should use overloaded functions rather than templates.

EXERCISES

1. If you have not done so, try each of the preceding examples.
2. Write a generic function, called **min( )**, that returns the lesser of its two arguments. For example, **min(3, 4)** will return 3 and **min('c', 'a')** will return a. Demonstrate your function in a program.
3. A good candidate for a template function is called **find( )**. This function searches an array for an object. It returns either the index of the matching object (if one is found) or -1 if no match is found. Here is the prototype for a specific version of **find( )**. Convert **find( )** into a generic function and demonstrate your solution within a program. (The **size** parameter specifies the number of elements in the array.)

```
int find(int object, int *list, int size)
{
  // ...
}
```

4. In your own words, explain why generic functions are valuable and may help simplify the source code to programs that you create.

11.2 GENERIC CLASSES

In addition to defining generic functions, you can also define generic classes. When you do this, you create a class that defines all algorithms used by that class, but the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.

Generic classes are useful when a class contains generalizable logic. For example, the same algorithm that maintains a queue of integers will also work for a queue of characters. Also, the same mechanism that maintains a linked list of mailing addresses will also maintain a linked list of auto part information. By using a generic class, you can create a class that will maintain a queue, a linked list, and so on for any type of data. The compiler will automatically generate the correct type of object based upon the type you specify when the object is created.

The general form of a generic class declaration is shown here:

```
template <class Ttype> class class-name {
    .
    .
    .
}
```

Here *Ttype* is the placeholder type name that will be specified when a class is instantiated. If necessary, you can define more than one generic data type by using a comma-separated list.

Once you have created a generic class, you create a specific instance of that class by using the following general form:

```
class-name <type> ob;
```

Here *type* is the type name of the data that the class will be operating upon.

Member functions of a generic class are, themselves, automatically generic. They need not be explicitly specified as such using **template**.

As you will see in Chapter 14, C++ provides a library that is built upon template classes. This library is usually referred to as the Standard Template Library, or STL for short. It provides generic versions of the most commonly used algorithms and data structures. If you want to use the STL effectively, you'll need a solid understanding of template classes and their syntax.

EXAMPLES

1. This program creates a very simple generic singly linked list class. It then demonstrates the class by creating a linked list that stores characters.

```
// A simple generic linked list.
#include <iostream>
using namespace std;

template <class data_t> class list {
  data_t data;
  list *next;
public:
  list(data_t d);
  void add(list *node) {node->next = this; next = 0; }
  list *getnext() { return next; }
  data_t getdata() { return data; }
};

template <class data_t> list<data_t>::list(data_t d)
{
  data = d;
  next = 0;
}

int main()
{
  list<char> start('a');
  list<char> *p, *last;
  int i;

  // build a list
  last = &start;
  for(i=1; i<26; i++) {
    p = new list<char> ('a' + i);
    p->add(last);
    last = p;
  }

  // follow the list
  p = &start;
  while(p) {
    cout << p->getdata();
    p = p->getnext();
```

```
  }

  return 0;
}
```

As you can see, the declaration of a generic class is to that of a generic function. The actual type of data store ie list is generic in the class declaration. It is not until ar t of the list is declared that the actual data type is determi . In this example, objects and pointers are created inside **main( )** that specify that the data type of the list will be **char**. Pay special attention to this declaration:

```
list<char> start('a');
```

Notice how the desired data type is passed inside the angle brackets.

You should enter and execute this program. It builds a linked list that contains the characters of the alphabet and then displays them. However, by simply changing the type of data specified when **list** objects are created, you can change the type of data stored by the list. For example, you could create another object that stores integers by using this declaration:

```
list<int> int_start(1);
```

You can also use **list** to store data types that you create. For example, if you want to store address information, use this structure:

```
struct addr {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[12];
}
```

Then, to use **list** to generate objects that will store objects of type **addr**, use a declaration like this (assuming that **structvar** contains a valid **addr** structure):

```
list<addr> obj(structvar);
```

2. Here is another example of a generic class. It is a reworking of the **stack** class first introduced in Chapter 1. However, in this case, **stack** has been made into a template class. Thus, it can be used to store any type of object. In the example shown here, a character stack and a floating-point stack are created:

```
// This function demonstrates a generic stack.
#include <iostream>
using namespace std;

#define SIZE 10

// Create a generic stack class
template <class StackType> class stack {
  StackType stck[SIZE]; // holds the stack
  int tos; // index of top of stack

public:
  void init() { tos = 0; } // initialize stack
  void push(StackType ch); // push object on stack
  StackType pop(); // pop object from stack
};

// Push an object.
template <class StackType>
void stack<StackType>:: push(StackType ob)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = ob;
  tos++;
}

// Pop an object.
template <class StackType>
StackType stack<StackType>::pop()
{
  if(tos==0) {
    cout << "Stack is empty.\n";
    return 0; // return null on empty stack
```

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

int main()
{
  // Demonstrate character stacks.
  stack<char> s1, s2;  // create two stacks
  int i;

  // initialize the stacks
  s1.init();
  s2.init();

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

  // demonstrate double stacks
  stack<double> ds1, ds2;  // create two stacks

  // initialize the stacks
  ds1.init();
  ds2.init();

  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);

  for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop() << "\n";

  return 0;
```

As the **stack** class (and the preceding **list** class) illustrates, generic functions and classes provide a powerful tool that you can use to maximize your programming time because they allow you to define the general form of an algorithm that can be used with any type of data. You are saved from the tedium of creating separate implementations for each data type that you want the algorithm to work with.

3. A template class can have more than one generic data type. Simply declare all the data types required by the class in a comma-separated list within the **template** specification. For example, the following short example creates a class that uses two generic data types:

```
/* This example uses two generic data types in a
   class definition.
*/
#include <iostream>
using namespace std;

template <class Type1, class Type2> class myclass
{
  Type1 i;
  Type2 j;
public:
  myclass(Type1 a, Type2 b) { i = a; j = b; }
  void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
  myclass<int, double> ob1(10, 0.23);
  myclass<char, char *> ob2('X', "This is a test");

  ob1.show(); // show int, double
  ob2.show(); // show char, char *

  return 0;
}
```

This program produces the following output:

```
10 0.23
X This is a test
```

The program declares two types of objects. **ob1** uses integer and **double** data. **ob2** uses a character and a character pointer. For both cases, the compiler automatically generates the appropriate data and functions to accommodate the way the objects are created.

EXERCISES

1. If you have not yet done so, compile and run the two generic class examples. Try declaring lists and/or stacks of different data types.
2. Create and demonstrate a generic queue class.
3. Create a generic class, called **input**, that does the following when its constructor is called:
   - prompts the user for input,
   - inputs the data entered by the user, and
   - reprompts if the data is not within a predetermined range.

   Objects of type **input** should be declared like this:

   input ob("*prompt message*", *min-value*, *max-value*)

   Here *prompt message* is the message that prompts for input. The minimum and maximum acceptable values are specified by *min-value* and *max-value*, respectively. (Note: the type of data entered by the user will be the same as the type of *min-value* and *max-value*.)

11.3 EXCEPTION HANDLING

C++ provides a built-in error handling mechanism that is called *exception handling*. Using exception handling, you can more easily manage and respond to run-time errors. C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**. In the most general

terms, program statements th- you want to monitor for exceptions are contained in a **try** block. an exception (i.e., an error) occurs within the **try** block, it is thrown (using **throw**). The exception is caught, using **catch**, and processed. The following elaborates upon this general description.

As stated, any statement that throws an exception must have been executed from within a **try** block. (A function called from within a **try** block can also throw an exception.) Any exception must be caught by a **catch** statement that immediately follows the **try** statement that throws the exception. The general form of **try** and **catch** are shown here:

```
try {
  // try block
}
catch (type1 arg) {
  // catch block
}
catch (type2 arg) {
  // catch block
}
catch (type3 arg) {
  // catch block
}
  .
  .
  .
catch (typeN arg) {
  // catch block
}
```

The **try** block must contain the portion of your program that you want to monitor for errors. This can be as specific as monitoring a few statements within one function or as all-encompassing as enclosing the **main( )** function code within a **try** block (which effectively causes the entire program to be monitored).

When an exception is thrown, it is caught by its corresponding **catch** statement, which processes the exception. There can be more than one **catch** statement associated with a **try**. The **catch** statement that is used is determined by the type of the exception. That is, if the data type specified by a **catch** matches that of the exception, that

catch statement is executed (and all others are bypassed). When an exception is caught, *arg* will receive its value. If you don't need access to the exception itself, specify only *type* in the **catch** clause—*arg* is optional. Any type of data can be caught, including classes that you create. In fact, class types are frequently used as exceptions.

The general form of the **throw** statement is shown here:

throw *exception*;

throw must be executed either from within the **try** block proper or from any function that the code within the block calls (directly or indirectly). *exception* is the value thrown.

If you throw an exception for which there is no applicable **catch** statement, an abnormal program termination might occur. If your compiler complies with Standard C++, throwing an unhandled exception causes the standard library function **terminate( )** to be invoked. By default, **terminate( )** calls **abort( )** to stop your program but you can specify your own termination handler, if you like. You will need to refer to your compiler's library reference for details.

EXAMPLES

1. Here is a simple example that shows the way C++ exception handling operates:

```
// A simple exception handling example.
#include <iostream>
using namespace std;

int main()
{
  cout << "start\n";

  try { // start a try block
    cout << "Inside try block\n";
    throw 10; // throw an error
    cout << "This will not execute";
  }
  catch (int i) { // catch an error
    cout << "Caught One! Number is: ";
    cout << i << "\n";
  }
```

```
  cout << "end";

  return 0;
}
```

This program displays the following output:

```
start
Inside try block
Caught One! Number is: 10
end
```

Look carefully at this program. As you can see, there is a **try** block containing three statements and a **catch(int i)** statement that processes an integer exception. Within the **try** block, only two of the three statements will execute: the first **cout** statement and the **throw**. Once an exception has been thrown, control passes to the **catch** expression and the **try** block is terminated. That is, **catch** is *not* called. Rather, program execution is transferred to it. (The stack is automatically reset as needed to accomplish this.) Thus, the **cout** statement following the **throw** will never execute.

After the **catch** statement executes, program control continues with the statements following the **catch**. Often, however, a **catch** block will end with a call to **exit( )**, **abort( )**, or some other function that causes program termination because exception handling is frequently used to handle catastrophic errors.

2. As mentioned, the type of the exception must match the type specified in a **catch** statement. For example, in the preceding example, if you change the type in the **catch** statement to **double**, the exception will not be caught and abnormal termination will occur. This change is shown here:

```
// This example will not work.
#include <iostream>
using namespace std;

int main()
{
  cout << "start\n";

  try { // start a try block
```

```
    cout << "Inside try block\n";
    throw 10; // throw an error
    cout << "This will not execute";
  }
  catch (double i) { // Won't work for an int exception
    cout << "Caught One! Number is: ";
    cout << i << "\n";
  }

  cout << "end";

  return 0;
}
```

This program produces the following output because the integer exception will not be caught by a **double catch** statement.

```
start
Inside try block
Abnormal program termination
```

3. An exception can be thrown from a statement that is outside the **try** block as long as the statement is within a function that is called from within the **try** block. For example, this is a valid program:

```
/* Throwing an exception from a function outside
   the try block.
*/
#include <iostream>
using namespace std;

void Xtest(int test)
{
  cout << "Inside Xtest, test is: " << test << "\n";
  if(test) throw test;
}

int main()
{
  cout << "start\n";

  try { // start a try block
    cout << "Inside try block\n";
```

```
    Xtest(0);
    Xtest(1);
    Xtest(2);
  }
  catch (int i) { // catch an error
    cout << "Caught One! Number is: ";
    cout << i << "\n";
  }

  cout << "end";

  return 0;
}
```

This program produces the following output:

```
start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught One! Number is: 1
end
```

4. A **try** block can be localized to a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset. For example, examine this program:

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

// A try/catch can be inside a function other than main()
void Xhandler(int test)
{
  try{
    if(test) throw test;
  }
  catch(int i) {
    cout << "Caught One!  Ex. #: " << i << '\n';
  }
}

int main()
{
  cout << "start\n";
```

```
  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "end";

  return 0;
}
```

This program displays this output:

```
start
Caught One!  Ex. #: 1
Caught One!  Ex. #: 2
Caught One!  Ex. #: 3
end
```

As you can see, three exceptions are thrown. After each exception, the function returns. When the function is called again, the exception handling is reset.

5. As stated earlier, you can have more than one **catch** associated with a **try**. In fact, it is common to do so. However, each **catch** must catch a different type of exception. For example, the following program catches both integers and strings:

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

// Different types of exceptions can be caught.
void Xhandler(int test)
{
  try{
    if(test) throw test;
    else throw "Value is zero";
  }
  catch(int i) {
    cout << "Caught One!  Ex. #: " << i << '\n';
  }
  catch(char *str) {
    cout << "Caught a string: ";
    cout << str << '\n';
```

```
int main()
{
  cout << "start\n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "end";

  return 0;
}
```

This program produces the following output:

```
start
Caught One!  Ex. #: 1
Caught One!  Ex. #: 2
Caught a string: Value is zero
Caught One!  Ex. #: 3
end
```

As you can see, each **catch** statement responds only to its own type.

In general, **catch** expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other **catch** blocks are ignored.

EXERCISES

1. By far, the best way to understand how C++ exception handling works is to play with it. Enter, compile, and run the preceding example programs. Then experiment with them, altering pieces of them and observing the results.
2. What is wrong with this fragment?

```
int main()
{
  throw 12.23;
```

3. What is wrong with this fragment?

```
try {
  // ...
  throw 'a';
  // ...
}
catch(char *) {
 // ...
}
```

4. What will happen if an exception is thrown for which there is no corresponding **catch** statement?

11.4 MORE ABOUT EXCEPTION HANDLIN

There are several additional features and nuances to C++ exception handling that can make it easier and more convenient to use.

In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. This is easy to accomplish. Simply use this form of **catch**:

```
catch(...) {
  // process all exceptions
}
```

Here the ellipsis matches any type of data.

You can restrict the type of exceptions that a function can throw back to its caller. Put differently, you can control what type of exceptions a function can throw outside of itself. In fact, you can also prevent a function from throwing any exceptions whatsoever. To apply these restrictions, you must add a **throw** clause to the function definition. The general form of this is shown here:

```
ret-type func-name(arg-list) throw(type-list)
{
  // ...
}
```

Here only those data types contained in the comma-separated *type-list* may be thrown by the function. Throwing any other type of expression will cause abnormal program termination. If you don't want a function to be able to throw *any* exceptions, use an empty list.

If your compiler complies with Standard C++, when a function attempts to throw a disallowed exception the standard library function **unexpected( )** is called. By default, this causes the **terminate( )** function to be called, which causes abnormal program termination. However, you can specify your own termination handler, if you like. You will need to refer to your compiler's documentation for directions on how this can be accomplished.

If you wish to rethrow an expression from within an exception handler, you can do so by simply calling **throw**, by itself, with no exception. This causes the current exception to be passed on to an outer **try/catch** sequence.

EXAMPLES

1. The following program illustrates **catch(...)**:

```
// This example catches all exceptions.
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try{
    if(test==0) throw test; // throw int
    if(test==1) throw 'a'; // throw char
    if(test==2) throw 123.23; // throw double
  }
  catch(...) { // catch all exceptions
    cout << "Caught One!\n";
  }
}
```

```
int main()
{
  cout << "start\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "end";

  return 0;
}
```

This program displays the following output:

```
start
Caught One!
Caught One!
Caught One!
end
```

As you can see, all three **throw**s were caught using the one **catch** statement.

2. One very good use for **catch(...)** is as the last **catch** of a cluster of catches. In this capacity it provides a useful default or "catch all" statement. For example, this slightly different version of the preceding program explicitly catches integer exceptions but relies upon **catch(...)** to catch all others:

```
// This example uses catch(...) as a default.
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try{
    if(test==0) throw test; // throw int
    if(test==1) throw 'a'; // throw char
    if(test==2) throw 123.23; // throw double
  }
  catch(int i) { // catch an int exception
    cout << "Caught " << i << '\n';
  }
  catch(...) { // catch all other exceptions.
    cout << "Caught One!\n";
```

```
int main()
{
  cout << "start\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "end";

  return 0;
}
```

As This example

The output produced by this program is shown here:

```
start
Caught 0
Caught One!
Caught One!
end
```

As this example suggests, using **catch(...)** as a default is a good way to catch all exceptions that you don't want to handle explicitly. Also, by catching all exceptions, you prevent an unhandled exception from causing an abnormal program termination.

3. The following program shows how to restrict the types of exceptions that can be thrown from a function:

```
// Restricting function throw types.
#include <iostream>
using namespace std;

// This function can only throw ints, chars, and doubles
void Xhandler(int test) throw(int, char, double)
{
  if(test==0) throw test; // throw int
  if(test==1) throw 'a'; // throw char
  if(test==2) throw 123.23; // throw double
}

int main()
```

```
{
  cout << "start\n";

  try{
    Xhandler(0); // also, try passing 1 and 2 to Xhandler()
  }
  catch(int i) {
    cout << "Caught int\n";
  }
  catch(char c) {
    cout << "Caught char\n";
  }
  catch(double d) {
    cout << "Caught double\n";
  }

  cout << "end";

  return 0;
}
```

In this program, the function **Xhandler( )** can throw only integer, character, and **double** exceptions. If it attempts to throw any other type of exception, an abnormal program termination will occur. (That is, **unexpected( )** will be called.) To see an example of this, remove **int** from the list and retry the program.

It is important to understand that a function can only be restricted in what types of exceptions it throws back to the **try** block that called it. That is, a **try** block *within* a function can throw any type of exception so long as it is caught *within* that function. The restriction applies only when throwing an exception out of the function.

4. The following change to **Xhandler( )** prevents it from throwing any exceptions:

```
// This function can throw NO exceptions!
void Xhandler(int test) throw()
{
  /* The following statements no longer work.  Instead,
     they will cause an abnormal program termination. */
  if(test==0) throw test;
  if(test==1) throw 'a';
```

```
  if(test==2) throw 123.23;
}
```

5. As you have learned, you can rethrow an exception. The most likely reason for doing so is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another. An exception can only be rethrown from within a **catch** block (or from any function called from within that block). When you rethrow an exception, it will not be recaught by the same **catch** statement. It will propagate to an outer **catch** statement. The following program illustrates rethrowing an exception. It rethrows a **char *** exception.

```
// Example of rethrowing an exception.
#include <iostream>
using namespace std;

void Xhandler()
{
  try {
    throw "hello"; // throw a char *
  }
  catch(char *) { // catch a char *
    cout << "Caught char * inside Xhandler\n";
    throw ; // rethrow char * out of function
  }
}

int main()
{
  cout << "start\n";

  try{
    Xhandler();
  }
  catch(char *) {
    cout << "Caught char * inside main\n";
  }

  cout << "end";

  return 0;
```

This program displays the following output:

```
start
Caught char * inside Xhandler
Caught char * inside main
end
```

EXERCISES

1. Before continuing, compile and run all of the examples in this section. Be sure you understand why each program produces the output that it does.
2. What is wrong with this fragment?

```
try {
  // ...
  throw 10;
}
catch(int *p) {
  // ...
}
```

3. Show one way to fix the preceding fragment.
4. What **catch** expression catches all types of exceptions?
5. Here is a skeleton for a function called **divide( )**.

```
double divide(double a, double b)
{
  // add error handling
  return a/b;
}
```

This function returns the result of dividing **a** by **b**. Add error checking to this function using C++ exception handling. Specifically, prevent a divide-by-zero error. Demonstrate your solution in a program.

11.5 HANDLING EXCEPTIONS THROWN BY new

In Chapter 4 you learned that the modern specification for the **new** operator states that it will throw an exception if an allocation request fails. Since in Chapter 4 exceptions had not yet been discussed, a description of how to handle that exception was deferred until later. Now the time has come to examine precisely what occurs when **new** fails.

Before we begin, it is necessary to state that the material in this section describes the behavior of **new** as specified by Standard C++. As you should recall from Chapter 4, the precise action that **new** takes on failure has been changed several times since C++ was invented. Specifically, when C++ was first invented, **new** returned null on failure. Later this was changed such that **new** caused an exception on failure. Also, the name of this exception has changed over time. Finally, it was decided that a **new** failure will generate an exception by default, but that a null pointer could be returned instead, as an option. Thus, **new** has been implemented differently, at different times, by compiler manufacturers. Although all compilers will eventually implement **new** in compliance with Standard C++, not all currently do. If the code examples shown here do not work with your compiler, check your compiler's documentation for details on how it implements **new**.

In Standard C++, when an allocation request cannot be honored, **new** throws a **bad_alloc** exception. If you don't catch this exception, your program will be terminated. Although this behavior is fine for short sample programs, in real applications you must catch this exception and process it in some rational manner. To have access to this exception, you must include the header **<new>** in your program.

*Originally this exception was called **xalloc**, and at the time of this writing many compilers still use the older name. However, **bad_alloc** is the name specified by Standard C++, and it is the name that will be used in the future.*

In Standard C++ it is also possible to have **new** return null instead of throwing an exception when an allocation failure occurs. This form

of **new** is most useful when you are compiling older code with a modern C++ compiler. It is also valuable when you are replacing calls to **malloc( )** with **new**. This form of **new** is shown here.

p_var = new(nothrow) *type*;

Here *p_var* is a pointer variable of *type*. The **nothrow** form of **new** works like the original version of **new** from years ago. Since it returns null on failure, it can be "dropped into" older code and you won't have to add exception handling. However, for new code, exceptions provide a better alternative.

EXAMPLES

1. Here is an example of **new** that uses a **try/catch** block to monitor for an allocation failure.

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

int main()
{
  int *p;

   try {
    p = new int; // allocate memory for int
  } catch (bad_alloc xa) {
    cout << "Allocation failure.\n";
    return 1;
  }

  for(*p = 0; *p < 10; (*p)++)
    cout << *p << " ";

  delete p; // free the memory

  return 0;
}
```

Here if an allocation failure occurs, it is caught by the **catch** statement.

2. Since the previous program is unlikely to fail under any normal circumstance, the following program demonstrates **new**'s exception-throwing capability by forcing an allocation failure. It does this by allocating memory until it is exhausted.

```
// Force an allocation failure.
#include <iostream>
#include <new>
using namespace std;

int main()
{
  double *p;

  // this will eventually run out of memory
  do {
    try {
      p = new double[100000];
    } catch (bad_alloc xa) {
      cout << "Allocation failure.\n";
      return 1;
    }
    cout << "Allocation OK.\n";
  } while(p);

  return 0;
}
```

3. The following program shows how to use the **new(nothrow)** alternative. It reworks the preceding program and forces an allocation failure.

```
// Demonstrate the new(nothrow) alternative.
#include <iostream>
#include <new>
using namespace std;

int main()
{
  double *p;

  // this will eventually run out of memory
  do {
    p = new(nothrow) double[100000];
    if(p) cout << "Allocation OK.\n";
```

```
    else cout << "Allocation Error.\n";
  } while(p);

  return 0;
}
```

As this program demonstrates, when you use the **nothrow** approach, you must check the pointer returned by **new** after each allocation request.

EXERCISES

1. Explain the difference between the behavior of **new** and **new(nothrow)** when an allocation failure occurs.
2. Given the following fragment, show two ways to convert it into modern C++-style code.

```
p = malloc(sizeof(int));

if(!p) {
  cout << "Allocation error.\n";
  exit(1);
}
```

SKILLS CHECK



At this point you should be able to perform the following exercises and answer the questions.

1. Create a generic function that returns the mode of an array of values. (The *mode* of a set is the value that occurs most often.)

2. Create a generic function that returns the summation of an array of values.
3. Create a generic bubble sort (or use any other sorting algorithm you like).
4. Rework the **stack** class so that it can store pairs of different-type objects on the stack. Demonstrate your solution.
5. Show the general forms of **try**, **catch**, and **throw**. In your own words, describe their operation.
6. Again, rework the **stack** class so that stack over- and underflows are handled as exceptions.
7. Check your compiler's documentation. See whether it supports the **terminate( )** and **unexpected( )** functions. Generally, these functions can be configured to call any function you choose. If this is the case with your compiler, try creating your own set of customized termination functions that handle otherwise unhandled exceptions.
8. Thought question: Give a reason why having **new** generate an exception is a better approach than having **new** return null on failure.



This section checks how well you have integrated material in this chapter with that from preceding chapters.

1. In Chapter 6, Section 6.7, Example 3,.a safe array class was shown. On your own, convert it into a generic safe array.
2. In Chapter 1, overloaded versions of the **abs( )** function were created. As a better solution, create a generic **abs( )** function on your own that will return the absolute value of any numeric object.





12

Run-Time Type Identification and the Casting Operators

chapter objectives

12.1 Understanding Run-Time Type Identification (RTTI)

12.2 Using **dynamic_cast**

12.3 Using **const_cast**, **reinterpret_cast**, and **static_cast**

THIS chapter discusses two features that were recently added to the C++ language: run-time type identification (RTTI for short) and the new casting operators. RTTI allows you to identify the type of an object during the execution of your program. The casting operators give you safer, more controlled ways to cast. As you will see, one of the casting operators, **dynamic_cast**, relates directly to RTTI, so it makes sense to discuss these two subjects in the same chapter.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. What is a generic function and what is its general form?
2. What is a generic class and what is its general form?
3. Write a generic function called **gexp( )** that returns the value of one of its arguments raised to the power of the other.
4. In Chapter 9, Section 9.7, Example 1, a **coord** class that holds integer coordinates was created and demonstrated in a program. Create a generic version of the **coord** class that can hold coordinates of any type. Demonstrate your solution in a program.
5. Briefly explain how **try**, **catch**, and **throw** work together to provide C++ exception handling.
6. Can **throw** be used if execution has not passed through a **try** block?
7. What purpose do **terminate( )** and **unexpected( )** serve?
8. What form of **catch** will handle all types of exceptions?

12.1 UNDERSTANDING RUN-TIME TYPE IDENTIFICATION (RTTI)

Run-time type information might be new to you because it is not found in non-polymorphic languages such as C. In non-polymorphic languages, there is no need for run-time type information because the type of each object is known at compile time (i.e., when the program is written). However, in polymorphic languages such as C++, there can be situations in which the type of an object is unknown at compile time because the precise nature of that object is not determined until the program is executed. As you know, C++ implements polymorphism through the use of class hierarchies, virtual functions, and base class pointers. In this approach, a base class pointer can be used to point to objects of the base class or to *any object derived from that base*. Thus, it is not always possible to know in advance what type of object will be pointed to by a base pointer at any given moment in time. This determination must be made at run time, using run-time type identification.

To obtain an object's type, use **typeid**. You must include the header **<typeinfo>** in order to use **typeid**. The most common form of **typeid** is shown here:

```
typeid(object)
```

Here *object* is the object whose type you will be obtaining. **typeid** returns a reference to an object of type **type_info** that describes the type of object defined by *object*. The **type_info** class defines the following public members:

```
bool operator==(const type_info &ob);
bool operator!=(const type_info &ob);
bool before(const type_info &ob);
const char *name();
```

The overloaded == and != provide for the comparison of types. The **before( )** function returns true if the invoking object is before the object used as a parameter in collation order. (This function is mostly for internal use only. Its return value has nothing to do with inheritance or class hierarchies.) The **name( )** function returns a pointer to the name of the type.

While **typeid** will obtain the type of any object, its most important use is its application through a pointer of a polymorphic base class. In this case, it will automatically return the type of the actual object being pointed to, which can be a base class object or an object derived from that base. (Remember, a base class pointer can point to an object of the base class or of any class derived from that base.) Thus, using **typeid** you can determine at run time the type of the object that is being pointed to by a base class pointer. The same applies to references. When **typeid** is applied to a reference to an object of a polymorphic class, it will return the type of the object actually being referred to, which can be of a derived type.When **typeid** is applied to a non-polymorphic class, the base type of the pointer or reference is obtained.

There is a second form of **typeid,** one that takes a type name as its argument. This form is shown here:

typeid(*type-name*)

The main use of this form of **typeid** is to obtain a **type_info** object that describes the specified type so that it can be used in a type comparison statement.

Because **typeid** is commonly applied to a dereferenced pointer (i.e., one to which the * operator has been applied), a special exception has been created to handle the situation in which the pointer being dereferenced is null. In this case, **typeid** throws a **bad_typeid** exception.

Run-time type identification is not something that every program will use. However, when you are working with polymorphic types, it allows you to know what type of object is being operated upon in any given situation.

EXAMPLES

1. The following program demonstrates **typeid.** It first obtains type information about one of C++'s built-in types, **int**. It then displays the types of objects pointed to by **p**, which is a pointer of type **BaseClass**.

```
// An example that uses typeid.
#include <iostream>
#include <typeinfo>
```

```
using namespace std;

class BaseClass {
  virtual void f() {}; // make BaseClass polymorphic
  // ...
};

class Derived1: public BaseClass {
  // ...
};

class Derived2: public BaseClass {
  // ...
};

int main()
{
  int i;
  BaseClass *p, baseob;
  Derived1 ob1;
  Derived2 ob2;

  // First, display type name of a built-in type.
  cout << "Typeid of i is ";
  cout << typeid(i).name() << endl;

  // Demonstrate typeid with polymorphic types.
  p = &baseob;
  cout << "p is pointing to an object of type ";
  cout << typeid(*p).name() << endl;

  p = &ob1;
  cout << "p is pointing to an object of type ";
  cout << typeid(*p).name() << endl;

  p = &ob2;
  cout << "p is pointing to an object of type ";
  cout << typeid(*p).name() << endl;

  return 0;
}
```

The output produced by this program is shown here.

```
Typeid of i is int
p is pointing to an object of type class BaseClass
p is pointing to an object of type class Derived1
p is pointing to an object of type class Derived2
```

As explained, when **typeid** is applied to a base class pointer of a polymorphic type, the type of object pointed to will be determined at run time, as the output produced by the program shows. As an experiment, comment out the virtual function **f( )** in **BaseClass** and observe the results.

2. As explained, when **typeid** is applied to a reference to a polymorphic base class, the type returned is that of the actual object being referred to. The circumstance in which you will most often make use of this feature is when objects are passed to functions by reference. For example, in the following program, the function **WhatType( )** declares a reference parameter to objects of type **BaseClass**. This means that **WhatType( )** can be passed references to objects of type **BaseClass** or any class derived from **BaseClass**. When the **typeid** operator is applied to this parameter, it returns the actual type of the object being passed.

```
// Use a reference with typeid.
#include <iostream>
#include <typeinfo>
using namespace std;

class BaseClass {
  virtual void f() {}; // make BaseClass polymorphic
  // ...
};

class Derived1: public BaseClass {
  // ...
};

class Derived2: public BaseClass {
  // ...
};

// Demonstrate typeid with a reference parameter.
void WhatType(BaseClass &ob)
{
```

```
  cout << "ob is referencing an object of type ";
  cout << typeid(ob).name() << endl;
}

int main()
{
  int i;
  BaseClass baseob;
  Derived1 ob1;
  Derived2 ob2;

  WhatType(baseob);
  WhatType(ob1);
  WhatType(ob2);

  return 0;
}
```

The output produced by this program is shown here.

```
ob is referencing an object of type class BaseClass
ob is referencing an object of type class Derived1
ob is referencing an object of type class Derived2
```

3. Although obtaining the type name of an object is useful in some circumstances, often all you need to know is whether the type of one object matches that of another. Since the **type_info** object returned by **typeid** overloads the == and != operators, this too is easy to accomplish. The following program demonstrates the use of these operators.

```
// Demonstrate == and != relative to typeid.
#include <iostream>
#include <typeinfo>
using namespace std;

class X {
  virtual void f() {}
};

class Y {
  virtual void f() {}
};
```

```
int main()
{
  X x1, x2;
  Y y1;

  if(typeid(x1) == typeid(x2))
    cout << "x1 and x2 are same types\n";
  else
    cout << "x1 and x2 are different types\n";

  if(typeid(x1) != typeid(y1))
    cout << "x1 and y1 are different types\n";
  else
    cout << "x1 and y1 are same types\n";

  return 0;
}
```

The program displays the following output.

```
x1 and x2 are same types
x1 and y1 are different types
```

4. Although the preceding examples demonstrate the mechanics of using **typeid**, they don't show its full potential because the types in the preceding programs are knowable at compile time. In the following program, this is not the case. The program defines a simple class hierarchy that draws shapes on the screen. At the top of the hierarchy is the abstract class **Shape**. Four concrete subclasses are created: **Line**, **Square**, **Rectangle**, and **NullShape.** The function **generator( )** generates an object and returns a pointer to it. The actual object created is determined randomly based upon the outcome of the random number generator **rand( )**. (A function that produces objects is sometimes called an *object factory*.) Inside **main( )**, the shape of each object is displayed for all objects but **NullShape** objects, which have no shape. Since objects are generated randomly, there is no way to know in advance what type of object will be created next. Thus, the use of RTTI is required.

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

```
class Shape {
public:
  virtual void example() = 0;
};

class Rectangle: public Shape {
public:
  void example() {
    cout << "******\n*    *\n*    *\n******\n";
  }
};

class Triangle: public Shape {
public:
  void example() {
    cout << "*\n* *\n*  *\n******\n";
  }
};

class Line: public Shape {
public:
  void example() {
    cout << "******\n";
  }
};

class NullShape: public Shape {
public:
  void example() {
  }
};

// A factory for objects derived from Shape.
Shape *generator()
{
  switch(rand() % 4) {
    case 0:
      return new Line;
    case 1:
      return new Rectangle;
    case 2:
      return new Triangle;
    case 3:
      return new NullShape;
```

```
  return NULL;
}

int main()
{
  int i;
  Shape *p;

  for(i=0; i<10; i++) {
    p = generator(); // get next object

    cout << typeid(*p).name() << endl;

    // draw object only if it is not a NullShape
    if(typeid(*p) != typeid(NullShape))
      p->example();
  }

  return 0;
}
```

Sample output from the program is shown here.

```
class Rectangle
*****
*   *
*   *
*****
class NullShape
class Triangle
*
* *
*  *
*****
class Line
******
class Rectangle
*****
*   *
*   *
*****
class Line
******
```

```
class Triangle
*
* *
*  *
*****
class Triangle
*
* *
*  *
*****
class Triangle
*
* *
*  *
*****
class Line
******
```

5. The **typeid** operator can be applied to template classes. For example, consider the following program. It creates a hierarchy of template classes that store a value. The virtual function **get_val( )** returns a value that is defined by each class. For class **Num**, this is the value of the number itself. For **Square**, it is the square of the number, and for **Sqr_root**, it is the square root of the number. Objects derived from **Num** are generated by the **generator( )** function. The **typeid** operator is used to determine what type of object has been generated.

```
// typeid can be used with templates.
#include <iostream>
#include <typeinfo>
#include <cmath>
#include <cstdlib>
using namespace std;

template <class T> class Num {
public:
  T x;
  Num(T i) { x = i; }
  virtual T get_val() { return x; };
};

template <class T>
class Square : public Num<T> {
```

```
public:
  Square(T i) : Num<T>(i) {}
  T get_val() { return x*x; }
};

template <class T>
class Sqr_root : public Num<T> {
public:
  Sqr_root(T i) : Num<T>(i) {}
  T get_val() { return sqrt((double) x); }
};

// A factory for objects derived from Num.
Num<double> *generator()
{
  switch(rand() % 2 ) {
    case 0: return new Square<double> (rand() % 100);
    case 1: return new Sqr_root<double> (rand() % 100);
  }
  return NULL;
}

int main()
{
  Num<double> ob1(10), *p1;
  Square<double> ob2(100.0);
  Sqr_root<double> ob3(999.2);
  int i;

  cout << typeid(ob1).name() << endl;
  cout << typeid(ob2).name() << endl;
  cout << typeid(ob3).name() << endl;

  if(typeid(ob2) == typeid(Square<double>))
    cout << "is Square<double>\n";

  p1 = &ob2;

  if(typeid(*p1) != typeid(ob1))
    cout << "Value is: " << p1->get_val();
  cout << "\n\n";

  cout << "Now, generate some Objects.\n";
  for(i=0; i<10; i++) {
```

```
  p1 = generator(); // get next object

  if(typeid(*p1) == typeid(Square<double>))
    cout << "Square object: ";
  if(typeid(*p1) == typeid(Sqr_root<double>))
    cout << "Sqr_root object: ";

  cout << "Value is: " << p1->get_val();
  cout << endl;
}

  return 0;
}
```

The output from the program is shown here.

```
class Num<double>
class Square<double>
class Sqr_root<double>
is Square<double>
Value is: 10000

Now, generate some Objects.
Sqr_root object: Value is: 8.18535
Square object: Value is: 0
Sqr_root object: Value is: 4.89898
Square object: Value is: 3364
Square object: Value is: 4096
Sqr_root object: Value is: 6.7082
Sqr_root object: Value is: 5.19615
Sqr_root object: Value is: 9.53939
Sqr_root object: Value is: 6.48074
Sqr_root object: Value is: 6
```

EXERCISES

1. Why is RTTI a necessary feature of C++?
2. Try the experiment described in Example 1. What output do you see?
3. Is the following fragment correct?

```
cout << typeid(float).name();
```

4. Given this fragment, show how to determine whether **p** is pointing to a **D2** object.

```
class B {
  virtual void f() {}
};

class D1: public B {
  void f() {}
};

class D2: public B {
  void f() {}
};

int main()
{
  B *p;
```

5. Assuming the **Num** class from Example 5, is the following expression true or false?

```
typeid(Num<int>) == typeid(Num<double>)
```

6. On your own, experiment with RTTI. Although its use might seem a bit esoteric in the context of simple, sample programs, it is a powerful construct that allows you to manage objects at run time.

12.2 USING dynamic_cast

Although C++ still fully supports the traditional casting operator defined by C, C++ adds four new ones. They are **dynamic_cast**, **const_cast**, **reinterpret_cast**, and **static_cast.** We will examine **dynamic_cast** first because it relates to RTTI. The other casting operators are examined in the following section.

The **dynamic_cast** operator performs a run-time cast that verifies the validity of a cast. If, at the time **dynamic_cast** is executed, the cast

is invalid, the cast fails. The general form of **dynamic_cast** is shown here:

dynamic_cast<*target-type*> (*expr*)

Here *target-type* specifies the target type of the cast and *expr* is the expression being cast into the new type. The target type must be a pointer or reference type, and the expression being cast must evaluate to a pointer or reference. Thus, **dynamic_cast** can be used to cast one type of pointer into another or one type of reference into another.

The purpose of **dynamic_cast** is to perform casts on polymorphic types. For example, given the two polymorphic classes **B** and **D**, with **D** derived from **B**, a **dynamic_cast** can always cast a **D*** pointer into a **B*** pointer. This is because a base pointer can always point to a derived object. But a **dynamic_cast** can cast a **B*** pointer into an **D*** pointer only if the object being pointed to actually is a **D** object. In general, **dynamic_cast** will succeed if the pointer (or reference) being cast is a pointer (or reference) to either an object of the target type or an object derived from the target type. Otherwise, the cast will fail. If the cast fails, **dynamic_cast** evaluates to null if the cast involves pointers. If a **dynamic_cast** on reference types fails, a **bad_cast** exception is thrown.

Here is a simple example. Assume that **Base** is a polymorphic class and that **Derived** is derived from **Base**.

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // base pointer points to Derived object
dp = dynamic_cast<Derived *> (bp);
if(dp) cout << "Cast OK";
```

Here the cast from the base pointer **bp** to the derived pointer **dp** works because **bp** is actually pointing to a **Derived** object. Thus, this fragment displays **Cast OK**. But in the next fragment, the cast fails because **bp** is pointing to a **Base** object and it is illegal to cast a base object into a derived object.

```
bp = &b_ob; // base pointer points to Base object
dp = dynamic_cast<Derived *> (bp);
if(!dp) cout << "Cast Fails";
```

Because the cast fails, this fragment displays **Cast Fails**.

The **dynamic_cast** operator can sometimes be used instead of **typeid** in certain cases. For example, again assume that **Base** is a polymorphic base class for **Derived.** The following fragment will assign **dp** the address of the object pointed to by **bp** if and only if the object is really a **Derived** object.

```
Base *bp;
Derived *dp;
// ...
if(typeid(*bp) == typeid(Derived)) dp =(Derived *) bp;
```

In this case, a C-style cast is used to actually perform the cast. This is safe because the **if** statement checks the legality of the cast using **typeid** before the cast actually occurs. However, a better way to accomplish this is to replace the **typeid** operators and **if** statement with this **dynamic_cast**:

```
dp = dynamic_cast<Derived *> (bp);
```

Because **dynamic_cast** succeeds only if the object being cast is either already an object of the target type or an object derived from the target type, after this statement executes **dp** will contain either a null or a pointer to an object of type **Derived**. Since **dynamic_cast** succeeds only if the cast is legal, it can simplify the logic in certain situations.

EXAMPLES

1. The following program demonstrates **dynamic_cast**:

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

class Base {
public:
  virtual void f() { cout << "Inside Base\n"; }
  // ...
};

class Derived : public Base {
public:
  void f() { cout << "Inside Derived\n"; }
};
```

```
int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;

  dp = dynamic_cast<Derived *> (&d_ob);
  if(dp) {
    cout << "Cast from Derived * to Derived * OK.\n";
    dp->f();
  } else
    cout << "Error\n";

  cout << endl;

  bp = dynamic_cast<Base *> (&d_ob);
  if(bp) {
    cout << "Cast from Derived * to Base * OK.\n";
    bp->f();
  } else
    cout << "Error\n";

  cout << endl;

  bp = dynamic_cast<Base *> (&b_ob);
  if(bp) {
    cout << "Cast from Base * to Base * OK.\n";
    bp->f();
  } else
    cout << "Error\n";

  cout << endl;

  dp = dynamic_cast<Derived *> (&b_ob);
  if(dp)
    cout << "Error\n";
  else
    cout << "Cast from Base * to Derived * not OK.\n";

  cout << endl;

  bp = &d_ob; // bp points to Derived object
  dp = dynamic_cast<Derived *> (bp);
  if(dp) {
    cout << "Casting bp to a Derived * OK\n" <<
      "because bp is really pointing\n" <<
```

```
        "to a Derived object.\n";
      dp->f();
    } else
      cout << "Error\n";

    cout << endl;

    bp = &b_ob; // bp points to Base object
    dp = dynamic_cast<Derived *> (bp);
    if(dp)
      cout << "Error";
    else {
      cout << "Now casting bp to a Derived *\n" <<
        "is not OK because bp is really \n" <<
        "pointing to a Base object.\n";
    }

    cout << endl;

    dp = &d_ob; // dp points to Derived object
    bp = dynamic_cast<Base *> (dp);
    if(bp) {
      cout << "Casting dp to a Base * is OK.\n";
      bp->f();
    } else
      cout << "Error\n";

    return 0;
}
```

The program produces the following output.

```
Cast from Derived * to Derived * OK.
Inside Derived

Cast from Derived * to Base * OK.
Inside Derived

Cast from Base * to Base * OK.
Inside Base

Cast from Base * to Derived * not OK.

Casting bp to a Derived * OK
because bp is really pointing
```

```
to a Derived object.
Inside Derived

Now casting bp to a Derived *
is not OK because bp is really
pointing to a Base object.

Casting dp to a Base * is OK.
Inside Derived
```

The following example illustrates how a **dynamic_cast** can be used to replace **typeid**.

```
// Use dynamic_cast to replace typeid.
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
public:
  virtual void f() {}
};

class Derived : public Base {
public:
  void derivedOnly() {
    cout << "Is a Derived Object\n";
  }
};

int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;

  // ************************************
  // use typeid
  // ************************************
  bp = &b_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else
    cout << "Cast from Base to Derived failed.\n";
```

```
  bp = &d_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else
    cout << "Error, cast should work!\n";

  // ************************************
  // use dynamic_cast
  // ************************************
  bp = &b_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else
    cout << "Cast from Base to Derived failed.\n";

  bp = &d_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else
    cout << "Error, cast should work!\n";

  return 0;
}
```

As you can see, the use of **dynamic_cast** simplifies the logic required to cast a base pointer into a derived pointer. The output from the program is shown here.

```
Cast from Base to Derived failed.
Is a Derived Object
Cast from Base to Derived failed.
Is a Derived Object
```

3. The **dynamic_cast** operator can also be used with template classes. For example, the following program reworks the template class from Example 5 in the preceding section so that it uses **dynamic_cast** to determine the type of object returned by the **generator( )** function.

```
// dynamic_cast can be used with templates, too.
#include <iostream>
#include <typeinfo>
#include <cmath>
#include <cstdlib>
```

```
using namespace std;

template <class T> class Num {
public:
  T x;
  Num(T i) { x = i; }
  virtual T get_val() { return x; };
};

template <class T>
class Square : public Num<T> {
public:
  Square(T i) : Num<T>(i) {}
  T get_val() { return x*x; }
};

template <class T>
class Sqr_root : public Num<T> {
public:
  Sqr_root(T i) : Num<T>(i) {}
  T get_val() { return sqrt((double) x); }
};

// A factory for objects derived from Num.
Num<double> *generator()
{
  switch(rand() % 2 ) {
    case 0: return new Square<double> (rand() % 100);
    case 1: return new Sqr_root<double> (rand() % 100);
  }
  return NULL;
}

int main()
{
  Num<double> ob1(10), *p1;
  Square<double> ob2(100.0), *p2;
  Sqr_root<double> ob3(999.2), *p3;
  int i;

  cout << "Generate some objects.\n";
  for(i=0; i<10; i++) {
    p1 = generator();

    p2 = dynamic_cast<Square<double> *> (p1);
```

```
    if(p2) cout << "Square object: ";
    p3 = dynamic_cast<Sqr_root<double> *> (p1);
    if(p3) cout << "Sqr_root object: ";

    cout << "Value is: " << p1->get_val();
    cout << endl;
  }

  return 0;
}
```

EXERCISES

1. In your own words, explain the purpose of **dynamic_cast**.
2. Given the following fragment and using **dynamic_cast**, show how you can assign **p** a pointer to some object **ob** if and only if **ob** is a **D2** object.

```
class B {
  virtual void f() {}
};

class D1: public B {
  void f() {}
};

class D2: public B {
  void f() {}
};

B *p;
```

3. Convert the **main( )** function in Section 12.1, Exercise 4, so that it uses **dynamic_cast** rather than **typeid** to prevent a **NullShape** object from being displayed.
4. Using the **Num** class hierarchy from Example 3 in this section, will the following work?

```
Num<int> *Bp;
Square<double> *Dp;
// ...
Dp = dynamic_cast<Num<int>> (Bp);
```

USING const_cast, reinterpret_cast, AND static_cast

Although **dynamic_cast** is the most important of the new casting operators, the other three are also valuable to the programmer. Their general forms are shown here:

const_cast<*target-type*> (*expr*)

reinterpret_cast<*target-ype*> (*expr*)

static_cast<*target-type*> (*expr*)

Here *target-type* specifies the target type of the cast and *expr* is the expression being cast into the new type. In general, these casting operators provide a safer, more explicit means of performing certain type conversions than that provided by the C-style cast.

The **const_cast** operator is used to explicitly override **const** and/or **volatile** in a cast. The target type must be the same as the source type except for the alteration of its **const** or **volatile** attributes. The most common use of **const_cast** is to remove **const**-ness.

The **static_cast** operator performs a non-polymorphic cast. For example, it can be used to cast a base class pointer into a derived class pointer. It can also be used for any standard conversion. No run-time checks are performed.

The **reinterpret_cast** operator changes one pointer type into another, fundamentally different, pointer type. It can also change a pointer into an integer and an integer into a pointer. A **reinterpret_cast** should be used for casting inherently incompatible pointer types.

Only **const_cast** can cast away **const**-ness. That is, neither **dynamic_cast, static_cast,** nor **reinterpret_cast** can alter the **const**-ness of an object.

EXAMPLES

1. The following program demonstrates the use of **reinterpret_cast**.

```
// An example that uses reinterpret_cast.
#include <iostream>
```

```
using namespace std;

int main()
{
  int i;
  char *p = "This is a string";

  i = reinterpret_cast<int> (p); // cast pointer to integer

  cout << i;

  return 0;
}
```

Here **reinterpret_cast** converts the pointer **p** into an integer. This conversion represents a fundamental type change and is a good use of **reinterpret_cast**.

2. The following program demonstrates **const_cast**.

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

void f(const int *p)
{
  int *v;

  // cast away const-ness.
  v = const_cast<int *> (p);

  *v = 100; // now, modify object through v
}


int main()
{
  int x = 99;

  cout << "x before call: " << x << endl;
  f(&x);
  cout << "x after call: " << x << endl;

  return 0;
}
```

The output produced by this program is shown here.

```
x before call: 99
x after call: 100
```

As you can see, **x** was modified by **f( )** even though the parameter to **f( )** was specified as a **const** pointer.

It must be stressed that the use of **const_cast** to cast way **const**-ness is a potentially dangerous feature. Use it with care.

3. The **static_cast** operator is essentially a substitute for the original cast operator. It simply performs a non-polymorphic cast. For example, the following casts a **float** into an **int**.

```
// use static_cast
#include <iostream>
using namespace std;

int main()
{
  int i;
  float f;

  f = 199.22;

  i = static_cast<int> (f);

  cout << i;

  return 0;
}
```

EXERCISES

1. Explain the rationale for **const_cast**, **reinterpret_cast**, and **static_cast**.
2. The following program contains an error. Show how to fix it using a **const_cast**.

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

```
void f(const double &i)
{
  i = 100; // Error -- fix using const_cast
}


int main()
{
  double x = 98.6;

  cout << x << endl;
  f(x);
  cout << x << endl;

  return 0;
}
```

3. Explain why **const_cast** should normally be reserved for special cases.

SKILLS CHECK



At this point you should be able to perform the following exercises and answer the questions.

1. Describe the operation of **typeid**.
2. What header must you include in order to use **typeid**?
3. In addition to the standard cast, C++ defines four casting operators. What are they and what are they for?
4. Complete the following partial program so that it reports which type of object has been selected by the user.

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

class A {
  virtual void f() {}
};

class B : public A {
};

class C: public B {
};

int main()
{
  A *p, a_ob;
  B b_ob;
  C c_ob;
  int i;

  cout << "Enter 0 for A objects, ";
  cout << "1 for B objects or ";
  cout << "2 for C objects.\n";

  cin >> i;

  if(i==1) p = &b_ob;
  else if(i==2) p = &c_ob;
  else p = &a_ob;

  // report type of object selected by user

  return 0;
}
```

5. Explain how **dynamic_cast** can sometimes be an alternative to **typeid**.
6. What type of object is obtained by the **typeid** operator?

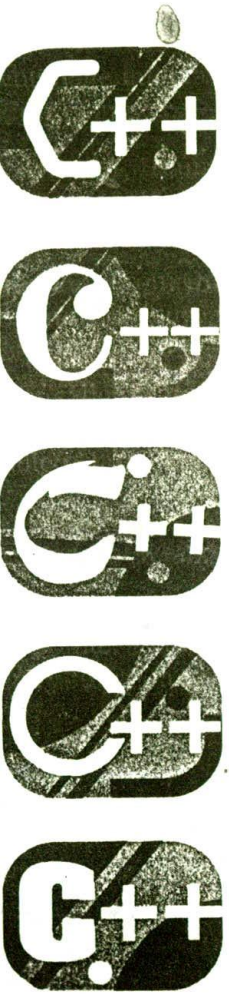


This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. Rework the program in Section 12.1, Example 4 so that it uses exception handling to watch for an allocation failure within the **generator( )** function.
2. Change the **generator( )** function from Question 1 so that it uses the **nothrow** version of **new**. Be sure to check for errors.
3. Special Challenge: On your own, create a class hierarchy that has at its top an abstract class called **DataStruct**. Create two concrete subclasses. Have one implement a stack, the other a queue. Create a function called **DataStructFactory( )** that has this prototype:

   ```
   DataStruct *DataStructFactory(char what);
   ```

   Have **DataStructFactory( )** create a stack if *what* is s and a queue if *what* is q. Return a pointer to the object created. Demonstrate that your factory function works.



13

Namespaces, Conversion Functions, and Miscellaneous Topics

chapter objectives

13.1 Namespaces

13.2 Creating a conversion function

13.3 **static** class members

13.4 **const** member functions and **mutable**

13.5 A final look at constructors

13.6 Using linkage specifiers and the **asm** keyword

13.7 Array-based I/O

THIS chapter discusses namespaces, conversion functions, static and const class members, and other specialized features of C++.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. What are the casting operators and what do they do?
2. What is **type_info**?
3. What operator determines the type of an object?
4. Given this fragment, show how to determine whether **p** points to an object of **Base** or an object of **Derived**.

```
class Base {
  virtual void f() {}
};

class Derived : public Base {
};

int main()
{
  Base *p, b_ob;
  Derived d_ob;

  // ...
```

5. A **dynamic_cast** succeeds only if the object being cast is a pointer to either an object of the target type or an object _____ from the target type. (Fill in the blank.)
6. Can a **dynamic_cast** cast away **const**-ness?

13.1 NAMESPACES

Namespaces were briefly introduced in Chapter 1. Now it is time to look at them in detail. Namespaces are a relatively recent addition to C++. Their purpose is to localize the names of identifiers to avoid name collisions. In the C++ programming environment, there has been 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 **toupper( )**, it could (depending upon its parameter list) override the standard library function **toupper( )** 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 another library.

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 giving rise to conflicts. Perhaps the most noticeable beneficiary of namespaces is the C++ standard library. In early versions of C++, the entire C++ library was defined within the global namespace (which was, of course, the only namespace). Now, however, the C++ library is defined within its own namespace, **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 might cause conflicts. This is especially important if you are creating class or function libraries.

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:

```
namespace MyNameSpace {
  int i, k;
  void myfunc(int j) { cout << j; }

  class myclass {
   public:
     void seti(int x) { i = x; }
     int geti() { return i; }
  };
}
```

Here **i**, **k**, **myfunc( )**, and the class **myclass** are part of the scope defined by the **MyNameSpace** namespace.

Identifiers declared within a namespace can be referred to directly within that namespace. For example, in **MyNameSpace** the **return i** statement uses **i** directly. 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 **i** from code outside **MyNameSpace**, you must use this statement:

```
MyNameSpace::i = 10;
```

Or, to declare an object of type **myclass** from outside **MyNameSpace**, you use a statement like this:

```
MyNameSpace::myclass 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.

As you can imagine, if your program includes frequent references to the members of a namespace, the need 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. When you use this form, all of the members defined within the specified namespace are brought into the current namespace and can be used without qualification. If you use the second form, only a specific member of the namespace is made visible. For example, assuming **MyNameSpace** as shown above, the following **using** statements and assignments are valid:

```
using MyNameSpace::k; // only k is made visible
k = 10; // OK because k is visible

using namespace MyNameSpace; // all members are visible
i = 10; // OK because all members of MyNameSpace are visible
```

There can 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. Consider the following example:

```
namespace NS {
  int i;
}

// ...

namespace NS {
  int j;
}
```

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

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

There is a special type of namespace, called an *unnamed namespace,* that allows you to create identifiers that are unique within a file. It has 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 can be used directly, without qualification. But outside the file, the identifiers are unknown.

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

EXAMPLES

1. Here is an example that illustrates the attributes of a namespace.

```
// Namespace Demo
#include <iostream>
using namespace std;

// define a namespace
namespace firstNS {
  class demo {
    int i;
  public:
    demo(int x) { i = x; }
    void seti(int x) { i = x; }
    int geti() { return i; }
  };

  char str[] = "Illustrating namespaces\n";
  int counter;
}

// define another namespace
namespace secondNS {
  int x, y;
}

int main()
{
  // use scope resolution
  firstNS::demo ob(10);
```

```
  /* Once ob has been declared, its member functions
     can be used without namespace qualification. */
  cout << "Value of ob is : " << ob.geti();
  cout << endl;
  ob.seti(99);
  cout << "Value of ob is now : " << ob.geti();
  cout << endl;

  // bring str into current scope
  using firstNS::str;
  cout << str;

  // bring all of firstNS into current scope
  using namespace firstNS;
  for(counter = 10; counter; counter--)
    cout << counter << " ";
  cout << endl;

  // use secondNS namespace
  secondNS::x = 10;
  secondNS::y = 20;

  cout << "x, y: " << secondNS:: x;
  cout << ", " << secondNS::y << endl;

  // bring another namespace into view
  using namespace secondNS;
  demo xob(x), yob(y);

  cout << "xob, yob: " << xob.geti() << ", ";
  cout << yob.geti() << endl;

  return 0;
}
```

The output produced by this program is shown here.

```
Value of ob is : 10
Value of ob is now : 99
Illustrating namespaces
10 9 8 7 6 5 4 3 2 1
x, y: 10, 20
xob, yob: 10, 20
```

The program illustrates one 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 this program the **std**, **firstNS**, and **secondNS** namespaces have been added to the global namespace.

2. As mentioned, a namespace can be split between files or within a single file; its contents are additive. Consider this example:

```
// Namespaces are additive
#include <iostream>
using namespace std;

namespace Demo {
  int a; // In Demo namespace
}

int x; // this is in global namespace

namespace Demo {
  int b; // this is in Demo namespace, too
}

int main()
{
  using namespace Demo;

  a = b = x = 100;

  cout << a << " " << b << " " << x;

  return 0;
}
```

Here the **Demo** namespace contains both **a** and **b**, but not **x**.

3. As explained, Standard C++ defines its entire library in its own namespace, **std**. This is the reason that most of the programs in this book have included 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()
{
  double val;

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

  std::cin >> val;

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

  return 0;
}
```

Here **cout** and **cin** are both explicitly qualified by their namespace. That is, to write to standard output you must specify **std::cout**, and to read from standard input you must use **std::cin**.

You might 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, including **std** in the current namespace is far easier than qualifying each name individually.

4. If you are using only a few names from the standard library, it might 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. Here's an example:

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

// gain access to cout and cin
```

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

int main()
{
  double val;

  cout << "Enter a number: ";

  cin >> val;
  cout << "This is your number: ";
  cout << val;

  return 0;
}
```

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

5. As explained, the original C++ library was defined in the global namespace. If you will be converting older C++ programs, 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.

6. In C, if you want to restrict the scope of a global name to the file in which it is declared, you declare that name as **static**. For example, consider the following two files that are part of the same program:

File One

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

File Two

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

Because **counter** is defined in File One, it can be used in File One. In File Two, even though **counter** is specified as **extern**, it is still unavailable, and any attempt to use it results in an error. By preceding **counter** with **static** in File One, the programmer has restricted its scope to that file.

Although the use of **static** global declarations is still allowed in C++, a better way to accomplish the same end is to use an unnamed namespace, as shown in this example:

File One

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

File Two

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

Here **counter** is also restricted to File One. The use of the unnamed namespace rather than **static** is the method recommended by Standard C++.

EXERCISES

1. Convert the following program from Chapter 9 so that it does not use the **using namespace std** statement.

```
// Convert spaces to |s.
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: CONVERT <input> <output>\n";
    return 1;
  }

  ifstream fin(argv[1]); // open input file
  ofstream fout(argv[2]);  // create output file

  if(!fout) {
    cout << "Cannot open output file.\n";
    return 1;
  }
  if(!fin) {
```

```
    cout << "Cannot open input file.\n";
    return 1;
  }

  char ch;

  fin.unsetf(ios::skipws);  // do not skip spa
  while(!fin.eof()) {
    fin >> ch;
    if(ch==' ') ch = '!';
    if(!fin.eof()) fout << ch;
  }

  fin.close();
  fout.close();

  return 0;
}
```

2. Explain the operation of an unnamed namespace.
3. Explain the difference between the two forms of **using**.
4. Explain why most programs in this book contain a **using** statement. Describe one alternative.
5. Explain why you might want to put reusable code that you create into its own namespace.

13.2 CREATING A CONVERSION FUNCT

Sometimes it is useful to convert an object of one type into an object another. Although it is possible to use an overloaded operator functio to accomplish such a conversion, there is often an easier (and better) way: using a conversion function. A *conversion function* converts an object into a value compatible with another type, which is often one the built-in C++ types. In essence, a conversion function automatica converts an object into a value that is compatible with the type of the expression in which the object is used.

The general form of a conversion function is shown here:

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

Here *type* is the target type you will be converting to and *value* is the value of the object after the conversion has been performed. Conversion functions return a value of type *type*. No parameters can be specified, and a conversion function must be a member of the class for which it performs the conversion.

As the examples will illustrate, a conversion function generally provides a cleaner approach to converting an object's value into another type than any other method available in C++ because it allows an object to be included directly in an expression involving the target type.

EXAMPLES

1. In the following program, the **coord** class contains a conversion function that converts an object to an integer. In this case, the function returns the product of the two coordinates; however, any conversion appropriate to your specific application is allowed.

```
// A simple conversion function example.
#include <iostream>
using namespace std;

class coord {
  int x, y;
public:
  coord(int i, int j) { x = i; y = j; }
  operator int() { return x*y; } // conversion function
};

int main()
{
  coord o1(2, 3), o2(4, 3);
  int i;

  i = o1;  // automatically convert to integer
  cout << i << '\n';

  i = 100 + o2;  // convert o2 to integer
  cout << i << '\n';
```

```
  return 0;
}
```

This program displays 6 and 112.

In this example, notice that the conversion function is called when **o1** is assigned to an integer and when **o2** is used as part of a larger integer expression. As stated, by using a conversion function, you allow classes that you create to be integrated into "normal" C++ expressions without having to create a series of complex overloaded operator functions.

2. Following is another example of a conversion function. This one converts a string of type **strtype** into a character pointer to **str**.

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

class strtype {
  char str[80];
  int len;
public:
  strtype(char *s) { strcpy(str, s); len = strlen(s); }
  operator char *() { return str; } // convert to char*
};

int main()
{
  strtype s("This is a test\n");
  char *p, s2[80];

  p = s; // convert to char *
  cout << "Here is string: " << p << '\n';

  // convert to char * in function call
  strcpy(s2, s);
  cout << "Here is copy of string: " << s2 << '\n';

  return 0;
}
```

This program displays the following:

```
Here is string: This is a test
Here is copy of string: This is a test
```

As you can see, not only is the conversion function invoked when object **s** is assigned to **p** (which is of type **char** *), but it is also called when **s** is used as a parameter to **strcpy( )**. Remember, **strcpy( )** has the following prototype:

```
char *strcpy(char *s1, const char *s2);
```

Because the prototype specifies that *s2* is of type **char** *, the conversion function to **char** * is automatically called. This illustrates how a conversion function can also help you seamlessly integrate your classes into C++'s standard library functions.

EXERCISES

1. Using the **strtype** class from Example 2, create a conversion that converts to type **int**. In this conversion, return the length of the string held in **str**. Illustrate that your conversion function works.
2. Given this class,

```
class pwr {
  int base;
  int exp;
public:
  pwr(int b, int e) { base = b; exp = e; }
  // create conversion to integer here
};
```

create a conversion function that converts an object of type **pwr** to type integer. Have the function return the result of $\mathbf{base}^{\mathbf{exp}}$.

13.3 STATIC CLASS MEMBERS

It is possible for a class member variable to be declared as **static**. By using **static** member variables, you can bypass a number of rather

tricky problems. When you declare a member variable as **static**, you cause only one copy of that variable to exist—no matter how many objects of that class are created. Each object simply shares that one variable. Remember, for a normal member variable, each time an object is created, a new copy of that variable is created, and that copy is accessible only by that object. (That is, when normal variables are used, each object possesses its own copies.) However, there is only one copy of a **static** member variable, and all objects of its class share it. Also, the same **static** variable will be used by any classes derived from the class that contains the **static** member.

Although it might seem odd when you first think about it, a **static** member variable exists *before* any object of its class is created. In essence, a **static** class member is a global variable that simply has its scope restricted to the class in which it is declared. In fact, as you will see in one of the following examples, it is actually possible to access a **static** member variable independent of any object.

When you declare a **static** data member within a class, you are not defining it. Instead, you must provide a definition for it elsewhere, outside the class. To do this, you redeclare the **static** variable, using the scope resolution operator to identify which class it belongs to.

All **static** member variables are initialized to 0 by default. However, you can give a **static** class variable an initial value of your choosing, if you like.

Keep in mind that the principal reason **static** member variables are supported by C++ is to avoid the need for global variables. As you can surmise, classes that rely upon global variables almost always violate the encapsulation principle that is so fundamental to OOP and C++.

It is also possible for a member function to be declared as **static**, but this usage is not common. A member function declared as **static** can access only other **static** members of its class. (Of course, a **static** member function can access non-**static** global data and functions.) A **static** member function does not have a **this** pointer. Virtual **static** member functions are not allowed. Also, **static** member functions cannot be declared as **const** or **volatile**. A **static** member function can be invoked by an object of its class, or it can be called independent of any object, via the class name and the scope resolution operator.

EXAMPLES

1. Here is a simple example that uses a **static** member variable:

```
// A static member variable example.
#include <iostream>
using namespace std;

class myclass {
  static int i;
public:
  void seti(int n) { i = n; }
  int geti() { return i; }
};

// Definition of myclass::i. i is still private to myclass.
int myclass::i;

int main()
{
  myclass o1, o2;

  o1.seti(10);

  cout << "o1.i: " << o1.geti() << '\n'; // displays 10
  cout << "o2.i: " << o2.geti() << '\n'; // also displays 10

  return 0;
}
```

This program displays the following:

```
o1.i: 10
o2.i: 10
```

Looking at this program, you can see that only object **o1** actually sets the value of **static** member **i**. However, since **i** is shared by both **o1** and **o2** (and, indeed, by any object of type **myclass**), both calls to **geti( )** display the same result.

Notice how **i** is declared within **myclass** but defined outside of the class. This second step ensures that storage for **i** is defined. Technically, a class declaration is just that, only a declaration. No memory is actually set aside because of a declaration. Because a **static** data member *implies* that memory

is allocated for that member, a separate definition is required that causes storage to be allocated.

2. Because a **static** member variable exists before any object of that class is created, it can be accessed within a program independent of any object. For example, the following variation of the preceding program sets the value of **i** to 100 without any reference to a specific object. Notice the use of the scope resolution operator and class name to access **i**.

```
// Use a static member variable independent of any object.
#include <iostream>
using namespace std;

class myclass {
public:
  static int i;
  void seti(int n) { i = n; }
  int geti() { return i; }
};

int myclass::i;

int main()

  myclass o1, o2;

  // set i directly
  myclass::i = 100; // no object is referenced.

  cout << "o1.i: " << o1.geti() << '\n'; // displays 100
  cout << "o2.i: " << o2.geti() << '\n'; // also displays 100
  return 0;
}
```

Because **i** is set to 100, the following output is displayed:

```
o1.i: 100
o2.i: 100
```

3. One very common use of a **static** class variable is to coordinate access to a shared resource, such as a disk file, printer, or network server. As you probably know from your previous programming experience, coordinating access to a shared

resource requires some means of sequencing events. To get an idea of how **static** member variables can be used to control access to a shared resource, examine the following program. It creates a class called **output**, which maintains a common output buffer called **outbuf** that is, itself, a **static** character array. This buffer is used to receive output sent by the **outbuf( )** member function. This function sends the contents of **str** one character at a time. It does so by first acquiring access to the buffer and then sending all the characters in **str**. It locks out access to the buffer by other objects until it is done outputting. You should be able to follow its operation by studying the code and reading the comments.

```
// A shared resource example.
#include <iostream>
#include <cstring>
using namespace std;

class output {
  static char outbuf[255]; // this is the shared resource
  static int inuse; // buffer available if 0;
  static int oindex; // index of outbuf
  char str[80];
  int i; // index of next char in str
  int who; // identifies the object, must be > 0
public:
  output(int w, char *s) { strcpy(str, s); i = 0; who = w; }

  /* This function returns -1 if waiting for buffer,
     it returns 0 if it is done outputting, and
     it returns who if it is still using the buffer.
  */
  int putbuf()
  {
    if(!str[i]) { // done outputting
      inuse = 0; // release buffer
      return 0; // signal termination
    }
    if(!inuse) inuse = who; // get buffer
    if(inuse != who) return -1; // in use by someone else
    if(str[i]) { // still chars to output
      outbuf[oindex] = str[i];
      i++; oindex++;
```

```
      outbuf[oindex] = '\0';  // always keep null-terminated
      return 1;
    }
    return 0;
  }
  void show() { cout << outbuf << '\n'; }
};

char output::outbuf[255]; // this is the shared resource
int output::inuse = 0; // buffer available if 0
int output::oindex = 0; // index of outbuf

int main()
{
  output o1(1, "This is a test"), o2(2, " of statics");

  while(o1.putbuf() | o2.putbuf()) ; // output chars

  o1.show();

  return 0;
}
```

4. **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_func_demo {
  static int i;
public:
  static void init(int x) {i = x;}
  void show() {cout << i;}
};

int static_func_demo::i; // define i

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

```
  static_func_demo x;
  x.show(); // displays 100

  return 0;
}
```

Here **i** is initialized by the call to **init( )** before an object of **static_func_demo** exists.

EXERCISES

1. Rework Example 3 so that it displays which object is currently outputting characters and which one or ones are blocked from outputting a character because the buffer is already in use by another.
2. One interesting use of a **static** member variable is to keep track of the number of objects of a class that are in existence at any given point in time. The way to do this is to increment a **static** member variable each time the class' constructor is called and decrement it each time the class' destructor is called. Implement such a scheme and demonstrate that it works.

13.4 const MEMBER FUNCTIONS AND mutable

Class member functions can be declared as **const**. When this is done, that function cannot modify the object that invokes it. Also, a **const** object cannot 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.

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**, which overrides **const**-ness. That is, a **mutable** member can be modified by a **const** member function.

EXAMPLES

1. 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.

2. To allow selected members to be modified by a **const** member function, specify them as **mutable**. Here's an 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 can be changed by the **seti( )** function. However, **j** is not **mutable**, so **setj( )** is unable to modify its value.

EXERCISES

1. The following program attempts to create a simple countdown timer that rings a bell when the time period is over. You can specify the time period and increment when a **CountDown** object is created. Unfortunately, the program will not compile as shown here. Fix it.

```
// This program contains an error.
#include <iostream>
using namespace std;

class CountDown {
  int incr;
  int target;
  int current;
public:
  CountDown(int delay, int i=1) {
    target = delay;
    incr = i;
    current = 0;
  }

  bool counting() const {
    current += incr;
    if(current >= target) {
      cout << "\a";
      return false;
    }
    cout << current << " ";
    return true;
  }
};

int main()
{
  CountDown ob(100, 2);

  while(ob.counting()) ;
```

```
  return 0;
}
```

2. Can a **const** member function call a non-**const** function? Why not?

13.5 A FINAL LOOK AT CONSTRUCTORS

Although constructors were described early on in this book, there are still a few points that need to be made. 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);

  cout << ob.geta();

  return 0;
}
```

Here the constructor for **myclass** takes one parameter. Pay special attention to how **ob** is declared in **main( )**. The value 4, specified in the parentheses following **ob**, is the argument that is passed to **myclass( )**'s parameter **x**, which is used to initialize **a**. This is the form of initialization that we have been using since the start of this book. However, there is an alternative. For example, the following statement also initializes **a** to 4:

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

As the comment suggests, this form of initialization is automatically converted into a call to the **myclass** constructor with 4 as the

argument. That is, the preceding statement is handled by the compiler as if it were written like this:

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

In general, any time that 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.

If you do not want implicit conversions to be made, you can prevent them by using **explicit.** The **explicit** specifier applies only to constructors. A constructor specified as **explicit** will be used only when an initialization uses the normal constructor syntax. It will not perform any automatic conversion. For example, if the **myclass** constructor is declared 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(110);
```

will be allowed.

EXAMPLES

1. There can be more than one converting constructor in a class. For example, consider this variation on **myclass**:

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

```
class myclass {
  int a;
public:
  myclass(int x) { a = x; }
  myclass(char *str) { a = atoi(str); }
  int geta() { return a; }
};

int main()
{
  myclass ob1 = 4;      // converts to myclass(4)
  myclass ob2 = "123"; // converts to myclass("123");

  cout << "ob1: " << ob1.geta() << endl;
  cout << "ob2: " << ob2.geta() << endl;

  return 0;
}
```

Since both constructors use different type arguments (as, of course, they must), each initialization statement is automatically converted into its equivalent constructor call.

2. The automatic conversion from the type of a constructor's first argument into a call to the constructor itself has interesting implications. For example, assuming **myclass** from Example 1, the following **main( )** function makes use of the conversions from **int** and **char *** to assign **ob1** and **ob2** new values.

```
int main()
{
  myclass ob1 = 4;      // converts to myclass(4)
  myclass ob2 = "123"; // converts to myclass("123");

  cout << "ob1: " << ob1.geta() << endl;
  cout << "ob2: " << ob2.geta() << endl;

  // use automatic conversions to assign new values
  ob1 = "1776"; // converts into ob1 = myclass("1776");
  ob2 = 2001;   // converts into ob2 = myclass(2001);

  cout << "ob1: " << ob1.geta() << endl;
  cout << "ob2: " << ob2.geta() << endl;
```

```
  return 0;
}
```

3. To prevent the conversions just shown, you could specify the constructors as **explicit**, as shown here:

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

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

EXERCISES

1. In Example 3, if only **myclass(int)** is made explicit, will **myclass(char *)** still allow implicit conversions? (Hint: Try it.)
2. Will the following fragment work?

```
class Demo {
  double x
public:
  Demo(double i) { x = i; }
  // ...
};

// ...
Demo counter = 10;
```

3. Justify the inclusion of the **explicit** keyword. (In other words, explain why implicit constructor conversions might not be a desirable feature of C++ in some cases.)

USING LINKAGE SPECIFIERS AND THE asm KEYWORD

C++ provides two important mechanisms that make it easier to link C++ to other languages. One is the *linkage specifier*, which tells the compiler that one or more functions in your C++ program will be linked with another language that might have a different approach to naming, parameter passing, stack restoration, and the like. The second is the **asm** keyword, which allows you to embed assembly language instructions in your C++ source code. Both are examined here.

By default, all functions in a C++ program are compiled and linked as C++ functions. However, you can tell the C++ compiler to link a function so that it is compatible with another type of language. All C++ compilers allow functions to be linked as either C or C++ functions. Some also allow you to link functions with languages such as Pascal, Ada, or FORTRAN. To cause a function to be linked for a different language, use this general form of the linkage specification:

```
extern "language" function-prototype;
```

Here *language* is the name of the language with which you want the specified function to link. If you want to specify linkage for more than one function, use this form of the linkage specification:

```
extern "language" {
  function-prototypes
}
```

All linkage specifications must be global; they cannot be used inside a function.

The most common use of linkage specifications occurs when linking C++ programs to C code. By specifying "C" linkage you prevent the compiler from *mangling* (also known as *decorating*) the names of functions with embedded type information. Because of C++'s ability to overload functions and create member functions, the link-name of a function usually has type information added to it. Since C does not support overloading or member functions, it cannot recognize a mangled name. Using "C" linkage avoids this problem.

Although it is generally possible to link assembly language routines with a C++ program, there is often an easier way to use assembly language. C++ supports the special keyword **asm**, which allows you to embed assembly language instructions within a C++ function. These instructions are then compiled as is. The advantage of using an in-line assembler is that your entire program is completely defined as a C++ program and there is no need to link separate assembly language files. 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.

It's important to note that several compilers accept these three slightly different forms of the **asm** statement:

```
asm op-code;

asm op-code newline

asm {
  instruction sequence
}
```

Here *op-code* is not enclosed in double quotes. Because embedded assembly language instruction tends to be implementation dependent, you will want to read your compiler's user manual on this issue.

Microsoft Visual C++ uses ***__asm*** *for embedding assembly code. It is otherwise similar to* ***asm****.*

EXAMPLES

1. This program links **func( )** as a C, rather than a C++, function:

```
// Illustrate linkage specifier.
#include <iostream>
using namespace std;

extern "C" int func(int x); // link as C function

// This function now links as a C function.
int func(int x)
```

```
{
  return x/3;
}
```

This function can now be linked with code compiled by a C compiler.

2. The following fragment tells the compiler that **f1( )**, **f2( )**, and **f3( )** should be linked as C functions:

```
extern "C" {
  void f1();
  int f2(int x);
  double f3(double x, int *p);
}
```

3. This fragment embeds several assembly language instructions into **func( )**:

```
// Don't try this function!
void func()
{
  asm("mov bp, sp");
  asm("push ax");
  asm("mov cl, 4");
  // ...
}
```



You must be an accomplished assembly language programmer in order to successfully use in-line assembly language. Also, be sure to check your compiler's user manual for details regarding assembly language usage.

EXERCISE

1. On your own, study the sections in your compiler's user manual that refer to linkage specifications and assembly language interfacing.

13.7 ARRAY-BASED I/O

In addition to console and file I/O, C++ supports a full set of functions that use character arrays as the input or output device. Although C++'s array-based I/O parallels, in concept, the array-based I/O found in C (specifically, C's **sscanf( )** and **sprintf( )** functions), C++'s array-based I/O is more flexible and useful because it allows user-defined types to be integrated into it. Although it is not possible to cover every aspect of array-based I/O here, the most important and commonly used features are examined.

It is important to understand from the outset that array-based I/O still operates through streams. Everything you learned about C++ I/O in Chapters 8 and 9 is applicable to array-based I/O. In fact, you need to learn to use just a few new functions to take full advantage of array-based I/O. These functions link a stream to a region of memory. Once this has been accomplished, all I/O takes place through the I/O functions you know already.

Before you can use array-based I/O, you must be sure to include the header **<strstream>** in your file. In this header are defined the classes **istrstream**, **ostrstream**, and **strstream**. These classes create array-based input, output, and input/output streams, respectively. These classes have as a base **ios**, so all the functions and manipulators included in **istream**, **ostream**, and **iostream** are also available in **istrstream**, **ostrstream**, and **strstream**.

To use a character array for output, use this general form of the **ostrstream** constructor:

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

Here *ostr* will be the stream associated with the array *buf*. The size of the array is specified by *size*. Generally, *mode* is simply defaulted to output, but you can use any output mode flag defined by **ios** if you like. (Refer to Chapter 9 for details.)

Once an array has been opened for output, characters will be put into the array until it is full. The array will not be overrun. Any attempt to overfill the array will result in an I/O error. To find out how many characters have been written to the array, use the **pcount( )** member function, shown here:

```
streamsize pcount();
```

You must call this function in conjunction with a stream, and it will return the number of characters written to the array, including any null terminator.

To open an array for input, use this form of the **istrstream** constructor:

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

Here *buf* is a pointer to the array that will be used for input. The input stream will be called *istr*. When input is being read from an array, **eof( )** will return true when the end of the array has been reached.

To open an array for input/output operations, use this form of the **strstream** constructor:

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

Here *iostr* will be an input/output stream that uses the array pointed to by *buf*, which is *size* characters long.

The character-based streams are also referred to as *char * streams* in some C++ literature.

It is important to remember that all I/O functions described earlier operate with array-based I/O, including the binary I/O functions and the random-access functions.

The character-based stream classes have been deprecated by Standard C++. This means that they are still valid, but future versions of the C++ language might not support them. They are included in this book because they are still widely used. However, for new code you will probably want to use one of the containers described in Chapter 14.

EXAMPLES

1. Here is a short example that shows how to open an array for output and write data to it:

```
// A short example using array-based output.
#include <iostream>
#include <strstream>
using namespace std;

int main()
```

```
{
  char buf[255]; // output buffer

  ostrstream ostr(buf, sizeof buf); // open output array

  ostr << "Array-based I/O uses streams just like ";
  ostr << "'normal' I/O\n" << 100;
  ostr << ' ' << 123.23 << '\n';

  // you can use manipulators, too
  ostr << hex << 100 << ' ';
  // or format flags
  ostr.setf(ios::scientific);
  ostr << dec << 123.23;
  ostr << endl << ends;

  // show resultant string
  cout << buf;

  return 0;
}
```

This program displays

```
Array-based I/O uses streams just like 'normal' I/O
100 123.23
64 01.2323e+02
```

As you can see, the overloaded I/O operators, built-in I/O manipulators, member functions, and format flags are fully functional when you use array-based I/O. (This is also true of any manipulators or overloaded I/O operators you create relative to your own classes.)

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

2. Here is an example of array-based input:

```
// An example using array-based input.
#include <iostream>
#include <strstream>
using namespace std;
```

```
int main()
{
  char buf[] = "Hello 100 123.125 a";

  istrstream istr(buf); // open input array

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

  istr >> str >> i >> f >> c;

  cout << str << ' ' << i << ' ' << f;
  cout << ' ' << c << '\n';
  return 0;
}
```

This program reads and then redisplays the values contained in the input array **buf**.

3. Keep in mind that an input array, once linked to a stream, will appear the same as a file. For example, this program uses **get( )** and the **eof( )** function to read the contents of **buf**:

```
// Demonstrate get() and eof() with array-based I/O.
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char buf[] = "Hello 100 123.125 a";

  istrstream istr(buf);
  char c;

  while(!istr.eof()) {
    istr.get(c);
    if(!istr.eof()) cout << c;
  }

  return 0;
}
```

4. This program performs input and output on an array:

```
// Demonstrate an input/output array.
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char iobuf[255];

  strstream iostr(iobuf, sizeof iobuf);

  iostr << "This is a test\n";
  iostr << 100 << hex << ' ' << 100 << ends;

  char str[80];
  int i;

  iostr.getline(str, 79); // read string up to \n
  iostr >> dec >> i; // read 100

  cout << str << ' ' << i << ' ';

  iostr >> hex >> i;
  cout <<  hex << i;

  return 0;
}
```

The program first writes output to **iobuf**. It then reads it back. It first reads the entire line "This is a test" using the **getline( )** function. It then reads the decimal value 100 and the hexadecimal value 0x64.

EXERCISES

1. Modify Example 1 so it displays the number of characters written to **buf** prior to termination.

2. Write an application that uses array-based I/O to copy the contents of one array to another. (This is, of course, not the most efficient way to accomplish this task.)
3. Using array-based I/O, write a program that converts a string that contains a floating-point value into its internal representation.

SKILLS CHECK



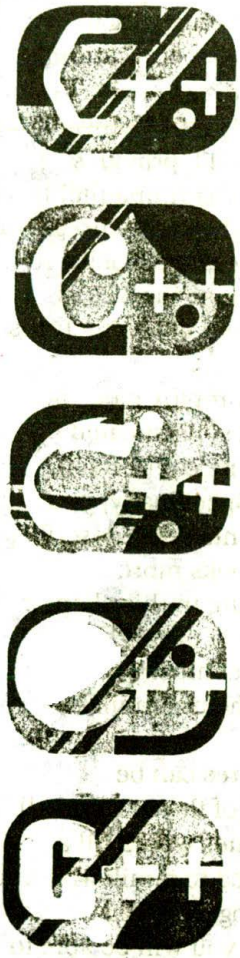
At this point you should be able to perform the following exercises and answer the questions.

1. What makes a **static** member variable different from other member variables?
2. What header must be included in your program when you use array-based I/O?
3. Aside from the fact that array-based I/O uses memory as an input and/or output device, is there any difference between it and "normal" I/O in C++?
4. Given a function called **counter( )**, show the statement that causes the compiler to compile this function for C language linkage.
5. What does a conversion function do?
6. Explain the purpose of **explicit**.
7. What is the principal restriction placed on a **const** member function?
8. Explain **namespace**.
9. What does **mutable** do?



This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. Since a constructor that requires only one argument provides an automatic conversion of the type of that argument to its class type, is there a situation in which this feature eliminates the need to create an overloaded assignment operator?
2. Can a **const_cast** be used within a **const** member function to allow it to modify its invoking object?
3. Thought question: Since the original C++ library was contained in the global namespace and all old C++ programs have already dealt with this fact, what is the advantage to moving the library into the **std** namespace "after the fact," so to speak?
4. Look back at the examples in the first twelve chapters. Think about what member functions can be made **const** or **static**. Are there examples in which defining a namespace would be appropriate?



14

Introducing the Standard Template Library

chapter objectives

14.1 An overview of the Standard Template Library

14.2 The container classes

14.3 Vectors

14.4 Lists

14.5 Maps

14.6 Algorithms

14.7 The **string** class

CONGRATULATIONS! If you have worked your way through the preceding chapters of this book, you can definitely call yourself a C++ programmer. In this, the final chapter of the book, we will explore one of C++'s most exciting—and most advanced—features: the Standard Template Library.

The inclusion of the Standard Template Library, or STL, was one of the major efforts that took place during the standardization of C++. The STL was not part of the original specification for C++ but was added during the standardization proceedings. The STL provides general-purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures. For example, it includes support for vectors, lists, queues, and stacks. It also defines various routines that access them. Because the STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.

It must be stated at the outset that the STL is a complex piece of software engineering that uses some of C++'s most sophisticated features. To understand and use the STL you must be comfortable with all of the material in the preceding chapters. Specifically, you must feel at home with templates. The template syntax that describes the STL can seem quite intimidating—although it looks more complicated than it actually is. While there is nothing in this chapter that is any more difficult than the material in the rest of this book, don't be surprised or dismayed if you find the STL confusing at first. Just be patient, study the examples, and don't let the unfamiliar syntax distract you from the STL's basic simplicity.

The STL is a large library, and not all of its features can be described in this chapter. In fact, a full description of the STL and all of its features, nuances, and programming techniques would fill a large book. The overview presented here is intended to familiarize you with its basic operation, design philosophy, and programming fundamentals. After working through this chapter, you will be able to easily explore the remainder of the STL on your own.

This chapter also describes one of C++'s most important new classes: **string**. The **string** class defines a string data type that allows you to work with character strings much as you do with other data types, using operators.



Before proceeding, you should be able to correctly answer the following questions and do the exercises.

1. Explain why **namespace** was added to C++.
2. How do you specify a **const** member function?
3. The **mutable** modifier allows a library function to be changed by the user of your program. True or false?
4. Given this class,

```
class X {
  int a, b;
public:
  X(int i, int j) { a = i, b = j; }
  // create conversion to int here.
};
```

   create an integer conversion function that returns the sum of **a** and **b**.

5. A **static** member variable can be used before an object of its class exists. True or false?
6. Given this class,

```
class Demo {
  int a;
public:
  explicit Demo(int i) { a = i; }
  int geta() { return a; }
};
```

   is the following declaration legal?

```
Demo o = 10;
```

AN OVERVIEW OF THE STANDARD TEMPLATE LIBRARY

Although the Standard Template Library is large and its syntax is, at times, rather intimidating, it is actually quite easy to use once you understand how it is constructed and what elements it employs. Therefore, before looking at any code examples, an overview of the STL is warranted.

At the core of the Standard Template Library are three foundational items: containers, algorithms, and iterators. These items work in conjunction with one another to provide off-the-shelf solutions to a variety of programming problems.

Containers are objects that hold other objects. There are several different types of containers. For example, the **vector** class defines a dynamic array, **queue** creates a queue, and **list** provides a linear list. In addition to the basic containers, the STL also defines *associative containers*, which allow efficient retrieval of values based on keys. For example, the **map** class defines a map that provides access to values with unique keys. Thus, a map stores a key/value pair and allows a value to be retrieved when its key is given.

Each container class defines a set of functions that can be applied to the container. For example, a list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.

Algorithms act on containers. Some of the services algorithms perform are initializing, sorting, searching, and transforming the contents of containers. Many algorithms operate on a *sequence*, which is a linear list of elements within a container.

Iterators are objects that are, more or less, pointers. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array. The five types of iterators are described in the following table.

Iterator	Access Allowed
Random access	Stores and retrieves values. Elements can be accessed randomly.
Bidirectional	Stores and retrieves values. Forward and backward moving.
Forward	Stores and retrieves values. Forward moving only.

Iterator	Access Allowed
Input	Retrieves but does not store values. Forward moving only.
Output	Stores but does not retrieve values. Forward moving only.

In general, an iterator that has greater access capabilities can be used in place of one that has lesser capabilities. For example, a forward iterator can be used in place of an input iterator.

Iterators are handled just like pointers. You can increment and decrement them. You can apply the * operator to them. Iterators are declared using the **iterator** type defined by the various containers.

The STL also supports *reverse iterators*. Reverse iterators are either bidirectional or random-access iterators that move through a sequence in reverse direction. Thus, if a reverse iterator points to the end of a sequence, incrementing that iterator will cause it to point to one element before the end.

When referring to the various iterator types in template descriptions, this book will use the terms listed in the following table:

Term	Iterator Type
BiIter	Bidirectional iterator
ForIter	Forward iterator
InIter	Input iterator
OutIter	Output iterator
RandIter	Random-access iterator

In addition to containers, algorithms, and iterators, the STL relies upon several other standard components for support. Chief among these are allocators, predicates, and comparison functions.

Each container has an *allocator* defined for it. Allocators manage memory allocation for containers. The default allocator is an object of class **allocator**, but you can define your own allocators if you need them for specialized applications. For most uses, the default allocator is sufficient.

Several of the algorithms and containers use a special type of function called a *predicate*. There are two variations of predicates: unary and binary. A unary predicate takes one argument, and a binary

predicate has two arguments. These functions return true or false; the precise conditions that make them return true or false are defined by the programmer. In this chapter, when a unary predicate function is used, it will be notated with the type **UnPred**. When a binary predicate is used, it will be of type **BinPred**. In a binary predicate, the arguments are always in the order of *first,second*. For both unary and binary predicates, the arguments will contain values of the same type as the objects being stored by the container.

Some algorithms and classes use a special type of binary predicate that compares two elements. Called a *comparison function,* this type of predicate returns true if its first argument is less than its second. Comparison functions will be notated by the type **Comp**.

In addition to the headers required by the various STL classes, the C++ standard library includes the **<utility>** and **<functional>** headers, which provide support for the STL. For example, **<utility>** contains the definition of the template class **pair,** which can hold a pair of values. We will make use of **pair** later in this chapter.

The templates in **<functional>** help you construct objects that define **operator( )**. These are called *function objects,* and they can be used in place of function pointers in many places. There are several predefined function objects declared within **<functional>**. Some are shown in the following table.

plus	minus	multiplies	divides	modulus
negate	equal_to	not_equal_to	greater	greater_equal
less	less_equal	logical_and	logical_or	logical_not

Perhaps the most widely used function object is **less,** which determines whether the value of one object is less than the value of another. Function objects can be used in place of actual function pointers in the STL algorithms described later. Using function objects rather than function pointers allows the STL to generate more efficient code. However, for the purposes of this chapter, function objects are not needed and we won't be using them directly. Although function objects are not difficult, a detailed discussion of function objects is quite lengthy and is beyond the scope of this book. They are something that you will need, however, to get maximum efficiency from the STL.

EXERCISES

1. As they relate to the STL, what are containers, algorithms, and iterators?
2. What are the two types of predicates?
3. What are the five types of iterators?

14.2 THE CONTAINER CLASSES

As explained, containers are the STL objects that actually store data. The containers defined by the STL are shown in Table 14-1. Also shown are the headers you must include to use each container. The **string** class, which manages character strings, is also a container, but it is discussed later in this chapter.

Container	Description	Required Header
bitset	A set of bits	<bitset>
deque	A double-ended queue	<deque>
list	A linear list	<list>
map	Stores key/value pairs in which each key is associated with only one value	<map>
multimap	Stores key/value pairs in which one key can be associated with two or more values	<map>
multiset	A set in which each element is not necessarily unique	<set>
priority_queue	A priority queue	<queue>
queue	A queue	<queue>
set	A set in which each element is unique	<set>
stack	A stack	<stack>
vector	A dynamic array	<vector>

TABLE 14-1 *The Containers Defined by the STL* ▼

Because the names of the placeholder types in a template class declaration are arbitrary, the container classes declare **typedef**ed versions of these types. This makes the type names concrete. Some o the most common **typedef** names are shown in the following table.

typedef Name	Description
size_type	An integral type equivalent to **size_t**
reference	A reference to an element
const_reference	A **const** reference to an element
iterator	An iterator
const_iterator	A **const** iterator
reverse_iterator	A reverse iterator
const_reverse_iterator	A **const** reverse iterator
value_type	The type of a value stored in a container
allocator_type	The type of the allocator
key_type	The type of a key
key_compare	The type of a function that compares two keys
value_compare	The type of a function that compares two values

Although it is not possible to examine each container in this chapter, the next sections explore three representatives: **vector**, **list**, and **map**. Once you understand how these containers work, you will have no trouble using the others.

VECTORS

Perhaps the most general-purpose of the containers is the vector. The **vector** class supports a dynamic array. This is an array that can grow as needed. As you know, in C++ the size of an array is fixed at compil time. Although this is by far the most efficient way to implement arrays, it is also the most restrictive, because the size of the array cannot be adjusted at run time to accommodate changing program conditions. A vector solves this problem by allocating memory as needed. Although a vector is dynamic, you can still use the standard array subscript notation to access its elements.

The template specification for **vector** is shown here:

```
template <class T, class Allocator = allocator<T>> class vector
```

Here **T** is the type of data being stored and **Allocator** specifies the allocator, which defaults to the standard allocator. **vector** has the following constructors:

```
explicit vector(const Allocator &a = Allocator( ) );

explicit vector(size_type num, const T &val = T ( ),
                const Allocator &a = Allocator( ));

vector(const vector<T, Allocator> &ob);

template <class InIter> vector(InIter start, InIter end,
                const Allocator &a = Allocator( ));
```

The first form constructs an empty vector. The second form constructs a vector that has *num* elements with the value *val*. The value of *val* can be allowed to default. The third form constructs a vector that contains the same elements as *ob*. The fourth form constructs a vector that contains the elements in the range specified by the iterators *start* and *end*.

Any object that will be stored in a vector must define a default constructor. It must also define the < and == operations. Some compilers might require that other comparison operators be defined. (Because implementations vary, consult your compiler's documentation for precise information.) All of the built-in types automatically satisfy these requirements.

Although the template syntax looks rather complex, there is nothing difficult about declaring a vector. Here are some examples:

```
vector<int> iv; // creates a zero-length int vector
vector<char> cv(5); // creates a 5-element char vector
vector<char> cv(5, 'x'); // initializes a 5-element char vector
vector<int> iv2(iv); // creates an int vector from an int vector
```

The following comparison operators are defined for **vector**:

```
==, <, <=, !=, >, >=
```

Member Function	Description
template <class InIter> void assign(InIter *start*, InIter *end*);	Assigns the list the sequence defined by *start* and *end*.
template <class Size, class T> void assign(Size *num*, const T &*val* = T());	Assigns the list *num* elements of value *val*.
reference back(); const_reference back() const;	Returns a reference to the last element in the list.
iterator begin(); const_iterator begin() const;	Returns an iterator to the first element in the list.
void clear();	Removes all elements from the list.
bool empty() const;	Returns true if the invoking list is empty and false otherwise.
iterator end(); const_iterator end() const;	Returns an iterator to the end of the list.
iterator erase(iterator *i*);	Removes the element pointed to by *i*. Returns an iterator to the element after the one removed.
iterator erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*. Returns an iterator to the element after the last element removed.
reference front(); const_reference front() const;	Returns a reference to the first element in the list.
allocator_type get_allocator() const;	Returns the list's allocator.
iterator insert(iterator *i*, const T &*val* = T());	Inserts *val* immediately before the element specified by *i*. An iterator to the element is returned.
void insert(iterator *i*, size_type *num*, const T &*val*)	Inserts *num* copies of *val* immediately before the element specified by *i*.
template <class InIter> void insert(iterator *i* InIter *start*, InIter *end*);	Inserts the sequence defined by *start* and *end* immediately before the element specified by *i*.
size_type max_size() const;	Returns the maximum number of elements that the list can hold.

TABLE 14-3 *The* ***list*** *Member Functions* ▼

Member Function	Description
iterator erase(iterator *i*);	Removes the element pointed to by *i*. Returns an iterator to the element after the one removed.
iterator erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*. Returns an iterator to the element after the last element removed.
reference front(); const_reference front() const;	Returns a reference to the first element in the vector.
allocator_type get_allocator() const;	Returns the vector's allocator.
iterator insert(iterator *i*, const T &*val* = T());	Inserts *val* immediately before the element specified by *i*. An iterator to the element is returned.
void insert(iterator *i*, size_type *num*, const T &*val*)	Inserts *num* copies of *val* immediately before the element specified by *i*.
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	Inserts the sequence defined by *start* and *end* immediately before the element specified by *i*.
size_type max_size() const;	Returns the maximum number of elements that the vector can hold.
reference operator[ ](size_type *i*) const; const_reference operator[ ](size_type *i*) const;	Returns a reference to the element specified by *i*.
void pop_back();	Removes the last element in the vector.
void push_back(const T &*val*);	Adds an element with the value specified by *val* to the end of the vector.
reverse_iterator rbegin(); const_reverse_iterator rbegin() const;	Returns a reverse iterator to the end of the vector.
reverse_iterator rend(); const_reverse_iterator rend() const;	Returns a reverse iterator to the start of the vector.
void reserve(size_type *num*);	Sets the capacity of the vector so that it is equal to at least *num*.
void resize(size_type *num*, T *val* = T ());	Changes the size of the vector to that specified by *num*. If the vector must be lengthened, elements with the value specified by *val* are added to the end.
size_type size() const;	Returns the number of elements currently in the vector.
void swap(vector<T, Allocator> &*ob*)	Exchanges the elements stored in the invoking vector with those in *ob*.

TABLE 14-2 *The* ***vector*** *Member Functions* (continued) ▼

The **push_back( )** function puts a value onto the end of the vector. If necessary, the vector is increased in length to accommodate the new element. You can also add elements to the middle using **insert( )**. A vector can also be initialized. In any event, once a vector contains elements, you can use array subscripting to access or modify those elements. You can remove elements from a vector using **erase( )**.

EXAMPLES

1. Here is a short example that illustrates the basic operation of a vector.

```
// Vector basics.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v; // create zero-length vector
  int i;

  // display original size of v
  cout << "Size = " << v.size() << endl;

  /* put values onto end of vector --
     vector will grow as needed */
  for(i=0; i<10; i++) v.push_back(i);

  // display current size of v
  cout << "Size now = " << v.size() << endl;

  // display contents of vector
  cout << "Current contents:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  /* put more values onto end of vector --
     again, vector will grow as needed */
  for(i=0; i<10; i++) v.push_back(i+10);

  // display current size of v
  cout << "Size now = " << v.size() << endl;
```

```
  // display contents of vector
  cout << "Current contents:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  // change contents of vector
  for(i=0; i<v.size(); i++) v[i] = v[i] + v[i];

  // display contents of vector
  cout << "Contents doubled:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  return 0;
}
```

The output of this program is shown here:

```
Size = 0
Size now = 10
Current contents:
0 1 2 3 4 5 6 7 8 9
Size now = 20
Current contents:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Contents doubled:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

Let's look at this program carefully. In **main( )**, an integer vector called **v** is created. Since no initialization is used, it is an empty vector with an initial capacity of zero. That is, it is a zero-length vector. The program confirms this by calling the **size( )** member function. Next, ten elements are added to the end of **v** with the member function **push_back( )**. This causes **v** to grow in order to accommodate the new elements. As the output shows, its size after these additions is 10. Next, the contents of **v** are displayed. Notice that the standard array subscripting notation is employed. Next, ten more elements are added and **v** is automatically increased in size to handle them. Finally, the values of **v**'s elements are altered using standard subscripting notation.

There is one other point of interest in this program. Notice that the loops that display the contents of **v** use as their target **v.size( )**. One of the advantages that vectors have over arrays is

that it is possible to find the current size of a vector. As you can imagine, this is quite useful in a variety of situations.

2. As you know, arrays and pointers are tightly linked in C++. An array can be accessed either through subscripting or through a pointer. The parallel to this in the STL is the link between vectors and iterators. You can access the members of a vector by using subscripting or by using an iterator. The following example shows both of these approaches.

```
// Access a vector using an iterator.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v; // create zero-length vector
  int i;

  // put values into a vector
  for(i=0; i<10; i++) v.push_back(i);

  // can access vector contents using subscripting
  for(i=0; i<10; i++) cout << v[i] << " ";
  cout << endl;

  // access via iterator
  vector<int>::iterator p = v.begin();
  while(p != v.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

The output from this program is:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

In this program, the vector is initially created with zero length. The **push_back( )** member function puts values onto the end of the vector, expanding its size as needed.

Notice how the iterator **p** is declared. The type **iterator** is defined by the container classes. Thus, to obtain an iterator for a particular container, you will use a declaration similar to that shown in the example: simply qualify **iterator** with the name of the container. In the program, **p** is initialized to point to the start of the vector by using the **begin( )** member function. This function returns an iterator to the start of the vector. This iterator can then be used to access the vector an element at a time by incrementing it as needed. This process is directly parallel to the way a pointer can be used to access the elements of an array. To determine when the end of the vector has been reached, the **end( )** member function is employed. This function returns an iterator to the location that is one past the last element in the vector. Thus, when **p** equals **v.end( )**, the end of the vector has been reached.

3. In addition to putting new values on the end of a vector, you can insert elements into the middle using the **insert( )** function. You can also remove elements using **erase( )**. The following program demonstrates **insert( )** and **erase( )**.

```
// Demonstrate insert and erase.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v(5, 1); // create 5-element vector of 1s
  int i;

  // display original contents of vector
  cout << "Size = " << v.size() << endl;
  cout << "Original contents:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl << endl;

  vector<int>::iterator p = v.begin();
  p += 2; // point to 3rd element

  // insert 10 elements with value 9
  v.insert(p, 10, 9);

  // display contents after insertion
```

```
  cout << "Size after insert = " << v.size() << endl;
  cout << "Contents after insert:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl << endl;

  // remove those elements
  p = v.begin();
  p += 2; // point to 3rd element
  v.erase(p, p+10); // remove next 10 elements

  // display contents after deletion
  cout << "Size after erase = " << v.size() << endl;
  cout << "Contents after erase:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  return 0;
}
```

This program produces the following output.

```
Size = 5
Original contents:
1 1 1 1 1

Size after insert = 15
Contents after insert:
1 1 9 9 9 9 9 9 9 9 9 9 1 1 1

Size after erase = 5
Contents after erase:
1 1 1 1 1
```

4. Here is an example that uses a vector to store objects of a programmer-defined class. Notice that the class defines the default constructor and that overloaded versions of < and == are provided. Remember, depending upon how your compiler implements the STL, other comparison operators might need to be defined.

```
// Store a class object in a vector.
#include <iostream>
#include <vector>
using namespace std;
```

```
class Demo {
  double d;
public:
  Demo() { d = 0.0; }
  Demo(double x) { d = x; }

  Demo &operator=(double x) {
    d = x; return *this;
  }

  double getd() { return d; }
};

bool operator<(Demo a, Demo b)
{
  return a.getd() < b.getd();
}

bool operator==(Demo a, Demo b)
{
  return a.getd() == b.getd();
}

int main()
{
  vector<Demo> v;
  int i;

  for(i=0; i<10; i++)
    v.push_back(Demo(i/3.0));

  for(i=0; i<v.size(); i++)
    cout << v[i].getd() << " ";

  cout << endl;

  for(i=0; i<v.size(); i++)
    v[i] = v[i].getd() * 2.1;

  for(i=0; i<v.size(); i++)
    cout << v[i].getd() << " "
```

The output from this program is shown here.

```
0 0.333333 0.666667 1 1.33333 1.66667 2 2.33333 2.66667 3
0 0.7 1.4 2.1 2.8 3.5 4.2 4.9 5.6 6.3
```

EXERCISES

1. Try the examples just shown, making small modifications and observing their effects.
2. In Example 4, both a default (i.e., parameterless) and a parameterized constructor were defined for **Demo**. Can you explain why this is important?
3. Here is a simple **Coord** class. Write a program that stores objects of type **Coord** in a vector. (Hint: Remember to define the < and == operators relative to **Coord**.)

```
class Coord {
public:
  int x, y;
  Coord() { x = y = 0; }
  Coord(int a, int b) { x = a; y = b; }
};
```

14.4 LISTS

The **list** class supports a bidirectional, linear list. Unlike a vector, which supports random access, a list can be accessed sequentially only. Because lists are bidirectional, they can be accessed front to back or back to front.

The **list** class has this template specification:

template <class T, class Allocator = allocator<T>> class list

Here **T** is the type of data stored in the list. The allocator is specified by **Allocator**, which defaults to the standard allocator. This class has the following constructors:

```
explicit list(const Allocator &a = Allocator( ) );

explicit list(size_type num, const T &val = T ( ),
              const Allocator &a = Allocator( ));

list(const list<T, Allocator> &ob);

template <class InIter> list(InIter start, InIter end,
              const Allocator &a = Allocator( ));
```

The first form constructs an empty list. The second form constructs a list that has *num* elements with the value *val*, which can be allowed to default. The third form constructs a list that contains the same elements as *ob*. The fourth form constructs a list that contains the elements in the range specified by the iterators *start* and *end*.

The following comparison operators are defined for **list**:

```
==, <, <=, !=, >, >=
```

The member functions defined for **list** are shown in Table 14-3. Like a vector, a list can have elements put into it with the **push_back( )** function. You can put elements on the front of the list by using **push_front( )**, and you can insert an element into the middle of a list by using **insert( )**. You can use **splice( )** to join two lists, and you can merge one list into another by using **merge( )**.

Any data type that will be held in a list must define a default constructor. It must also define the various comparison operators. At the time of this writing, the precise requirements for an object that will be stored in a list vary from compiler to compiler and are subject to change, so you will need to check your compiler's documentation.

The subscripting operator [ ] is also defined for **vector**. This allows you to access the elements of a vector using standard array subscripting notation.

The member functions defined by **vector** are shown in Table 14-2. (Again, it is important not to be put off by the syntax.) Some of the most important member functions are **size( )**, **begin( )**, **end( )**, **push_back( )**, **insert( )**, and **erase( )**. The **size( )** function returns the current size of the vector. This function is quite useful because it allows you to determine the size of a vector at run time. Remember, vectors will increase in size as needed, so the size of a vector must be determined during execution, not during compilation.

The **begin( )** function returns an iterator to the start of the vector. The **end( )** function returns an iterator to the end of the vector. As explained, iterators are similar to pointers, and it is through the use of the **begin( )** and **end( )** functions that you obtain an iterator to the beginning and end of a vector.

Member Function	Description
template <class InIter> void assign(InIter *start*, InIter *end*);	Assigns the vector the sequence defined by *start* and *end*.
template <class Size, class T> void assign(Size *num*, const T &*val* = T());	Assigns the vector *num* elements of value *val*.
reference at(size_type *i*); const_reference at(size_type *i*) const;	Returns a reference to an element specified by *i*.
reference back(); const_reference back() const;	Returns a reference to the last element in the vector.
iterator begin(); const_iterator begin() const;	Returns an iterator to the first element in the vector.
size_type capacity() const;	Returns the current capacity of the vector. This is the number of elements it can hold before it will need to allocate more memory.
void clear();	Removes all elements from the vector.
bool empty() const;	Returns true if the invoking vector is empty and false otherwise.
iterator end(); const_iterator end() const;	Returns an iterator to the end of the vector.

TABLE 14-2 *The **vector** Member Functions* ▼

Member Function	Description
void merge(list<T, Allocator> &*ob*); template <class Comp> void merge(<list<T, Allocator> &*ob*, Comp *cmpfn*);	Merges the ordered list contained in *ob* with the invoking ordered list. The result is ordered. After the merge, the list contained in *ob* is empty. In the second form, a comparison function can be specified to determine whether the value of one element is less than that of another.
void pop_back();	Removes the last element in the list.
void pop_front();	Removes the first element in the list.
void push_back(const T &*val*);	Adds an element with the value specified by *val* to the end of the list.
void push_front(const T &*val*);	Adds an element with the value specified by *val* to the front of the list.
reverse_iterator rbegin(); const_reverse_iterator rbegin() const;	Returns a reverse iterator to the end of the list.
void remove(const T &*val*);	Removes elements with the value *val* from the list.
template <class UnPred> void remove_if(UnPred *pr*);	Removes elements for which the unary predicate *pr* is true.
reverse_iterator rend(); const_reverse_iterator rend() const;	Returns a reverse iterator to the start of the list.
void resize(size_type *num*, T *val* = T ());	Changes the size of the list to that specified by *num*. If the list must be lengthened, elements with the value specified by *val* are added to the end.
void reverse();	Reverses the invoking list.
size_type size() const;	Returns the number of elements currently in the list.

void sort();
template <class Comp>
void sort(Comp *cmpfn*);
	Sorts the list. The second form sorts the list using the comparison function *cmpfn* to determine whether the value of one element is less than that of another.

void splice(iterator *i*, list<T, Allocator> &*ob*);	Inserts the contents of *ob* into the invoking list at the location pointed to by *i*. After the operation, *ob* is empty.

TABLE 14-3 *The* ***list*** *Member Functions (continued)*

Member Function	Description
void splice(iterator *i*, list<T, Allocator> &*ob*, iterator *el*);	Removes the element pointed to by *el* from the list *ob* and stores it in the invoking list at the location pointed to by *i*.
void splice(iterator *i*, list<T, Allocator> &*ob*, iterator *start*, iterator *end*);	Removes the range defined by *start* and *end* fr *ob* and stores it in the invoking list beginning a the location pointed to by *i*.
void swap(list<T, Allocator> &*ob*)	Exchanges the elements stored in the invoking list with those in *ob*.

void unique( );
template <class BinPred>
void unique(BinPred *pr*);
	Removes duplicate elements from the invoking list. The second form uses *pr* to determine uniqueness.

TABLE 14-3 *The list Member Functions* (continued) ▼

EXAMPLES

1. Here is a simple example of a list.

```
// List basics.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> lst; // create an empty list
  int i;

  for(i=0; i<10; i++) lst.push_back('A'+i);

  cout << "Size = " << lst.size() << endl;

  list<char>::iterator p;

  cout << "Contents: ";
  while(!lst.empty()) {
    p = lst.begin();
    cout << *p;
    lst.pop_front();
  }
```

```
  return 0;
}
```

The output produced by this program is shown here.

```
Size = 10
Contents: ABCDEFGHIJ
```

This program creates a list of characters. First, an empty **list** object is created. Next, ten characters, the letters A through J, are put into the list. This is accomplished with the **push_back( )** function, which puts each new value on the end of the existing list. Next, the size of the list is displayed. Then, the contents of the list are output by repeatedly obtaining, displaying, and then removing the first element in the list. This process stops when the list is empty.

2. In the previous example, the list was emptied as it was traversed. This is, of course, not necessary. For example, the loop that displays the list could be recoded as shown here.

```
list<char>::iterator p = lst.begin();

while(p != lst.end()) {
  cout << *p;
  p++;
}
```

Here the iterator **p** is initialized to point to the start of the list. Each time through the loop, **p** is incremented, causing it to point to the next element. The loop ends when **p** points to the end of the list.

3. Because lists are bidirectional, elements can be put on a list either at the front or at the back. For example, the following program creates two lists, with the first being the reverse of the second.

```
// Elements can be put on the front or end of a list.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> lst;
```

```
  list<char> revlst;
  int i;

  for(i=0; i<10; i++) lst.push_back('A'+i);

  cout << "Size of lst = " << lst.size() << endl;
  cout << "Original contents: ";

  list<char>::iterator p;

  /* Remove elements from lst and put them
     into revlst in reverse order. */
  while(!lst.empty()) {
    p = lst.begin();
    cout << *p;
    lst.pop_front();
    revlst.push_front(*p);
  }
  cout << endl << endl;

  cout << "Size of revlst = ";
  cout << revlst.size() << endl;
  cout << "Reversed contents: ";
  p = revlst.begin();
  while(p != revlst.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

This program produces the following output.

```
Size of lst = 10
Original contents: ABCDEFGHIJ

Size of revlst = 10
Reversed contents: JIHGFEDCBA
```

In the program, the list is reversed by removing elements from the start of **lst** and pushing them onto the front of **revlst**. This causes the elements to be stored in reverse order in **revlst**.

You can sort a list by calling the **sort( )** member function. The following program creates a list of random characters and then puts the list into sorted order.

```
// Sort a list.
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
  list<char> lst;
  int i;

  // create a list of random characters
  for(i=0; i<10; i++)
    lst.push_back('A'+ (rand()%26));

  cout << "Original contents: ";
  list<char>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  // sort the list
  lst.sort();

  cout << "Sorted contents: ";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

Here is sample output produced by the program.

```
Original contents: PHQGHUMEAY
Sorted contents: AEGHHMPQUY
```

5. One ordered list can be merged with another. The result is an ordered list that contains the contents of the two original lists. The new list is left in the invoking list and the second list is left empty. This example merges two lists. The first contains the letters ACEGI and the second BDFHJ. These lists are then merged to produce the sequence ABCDEFGHIJ.

```
// Merge two lists.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> lst1, lst2;
  int i;

  for(i=0; i<10; i+=2) lst1.push_back('A'+i);
  for(i=1; i<11; i+=2) lst2.push_back('A'+i);

  cout << "Contents of lst1: ";
  list<char>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  cout << "Contents of lst2: ";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  // now, merge the two lists
  lst1.merge(lst2);
  if(lst2.empty())
    cout << "lst2 is now empty\n";

  cout << "Contents of lst1 after merge:\n";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p;
```

```
    p++;
  }

  return 0;
}
```

The output produced by this program is shown here.

```
Contents of lst1: ACEGI

Contents of lst2: BDFHJ

lst2 is now empty
Contents of lst1 after merge:
ABCDEFGHIJ
```

6. Here is an example that uses a list to store objects of type **Project**, which is a class that helps manage software projects. Notice that the <, >, !=, and == operators are overloaded for objects of type **Project**. These are the operators that were required by Microsoft's Visual C++ 5 (the compiler used to test the STL examples in this chapter). Other compilers might require you to overload additional operators. The STL uses these functions to determine the ordering and equality of objects in a container. Even though a list is not an ordered container, it still needs a way to compare elements when searching, sorting, or merging.

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

class Project {
public:
  char name[40];
  int days_to_completion;
  Project() {
    strcpy(name, "");
    days_to_completion = 0;
  }
  Project(char *n, int d) {
    strcpy(name, n);
    days_to_completion = d;
  }
```

```
    void add_days(int i) {
      days_to_completion += i;
    }

    void sub_days(int i) {
      days_to_completion -= i;
    }

    bool completed() { return !days_to_completion; }

    void report() {
      cout << name << ": ";
      cout << days_to_completion;
      cout << " days left.\n";
    }
};

bool operator<(const Project &a, const Project &b)
{
  return a.days_to_completion < b.days_to_completion;
}

bool operator>(const Project &a, const Project &b)
{
  return a.days_to_completion > b.days_to_completion;
}

bool operator==(const Project &a, const Project &b)
{
  return a.days_to_completion == b.days_to_completion;
}

bool operator!=(const Project &a, const Project &b)
{
  return a.days_to_completion != b.days_to_completion;
}

int main()
{
  list<Project> proj;

  proj.push_back(Project("Compiler", 35));
  proj.push_back(Project("Spreadsheet", 190));
  proj.push_back(Project("STL Implementation", 1000));
```

```
  list<Project>::iterator p = proj.begin();

  /* display projects */
  while(p != proj.end()) {
    p->report();
    p++;
  }

  // add 10 days to 1st project
  p = proj.begin();
  p->add_days(10);

  // move 1st project to completion
  do {
    p->sub_days(5);
    p->report();
  } while (!p->completed());

  return 0;
}
```

The output from this program is shown here.

```
Compiler: 35 days left.
Spreadsheet: 190 days left.
STL Implementation: 1000 days left.
Compiler: 40 days left.
Compiler: 35 days left.
Compiler: 30 days left.
Compiler: 25 days left.
Compiler: 20 days left.
Compiler: 15 days left.
Compiler: 10 days left.
Compiler: 5 days left.
Compiler: 0 days left.
```

EXERCISES

1. Experiment with the examples, trying minor variations.
2. In Example 1, the list was emptied in the process of displaying it. In Example 2, you saw another way to traverse a list that does

not destroy it. Can you think of another way to traverse a list without emptying it? Show that your solution works by substituting it into the program in Example 1.

3. Expand Example 6 by creating another list of projects that consists of the following:

Project	Days to Completion
Database	780
Mail merge	50
COM objects	300

Next, sort both lists and then merge them together. Display the final result.

14.5 MAPS

The **map** class supports an associative container in which unique keys are mapped with values. In essence, a key is simply a name that you give to a value. Once a value has been stored, you can retrieve it by using its key. Thus, in its most general sense a map is a list of key/value pairs. The power of a map is that you can look up a value given its key. For example, you could define a map that uses a person's name as its key and stores that person's telephone number as its value. Associative containers are becoming more popular in programming.

As mentioned, a map can hold only unique keys. Duplicate keys are not allowed. To create a map that allows nonunique keys, use **multimap**.

The **map** container has the following template specification:

```
template <class Key, class T, class Comp = less<Key>,
          class Allocator = allocator<T>> class map
```

Here **Key** is the data type of the keys, **T** is the data type of the values being stored (mapped), and **Comp** is a function that compares two

keys. This defaults to the standard **less( )** utility function object. **Allocator** is the allocator (which defaults to **allocator**).

The **map** class has the following constructors:

```
explicit map(const Comp &cmpfn = Comp(),
             const Allocator &a = Allocator() );

map(const map<Key, T, Comp, Allocator> &ob);

template <class InIter> map(InIter start, InIter end,
             const Comp &cmpfn = Comp(), const Allocator &a = Allocator());
```

The first form constructs an empty map. The second form constructs a map that contains the same elements as *ob*. The third form constructs a map that contains the elements in the range specified by the iterators *start* and *end*. The function specified by *cmpfn*, if present, determines the ordering of the map.

In general, any object used as a key must define a default constructor and overload any necessary comparison operators.

The following comparison operators are defined for **map**.

==, <, <=, !=, >, >=

The member functions defined by **map** are shown in Table 14-4. In the descriptions, **key_type** is the type of the key and **value_type** represents **pair< Key, T >**.

Member Function	Description
iterator begin(); const_iterator begin() const;	Returns an iterator to the first element in the map.
void clear();	Removes all elements from the map.
size_type count(const key_type &*k*) const;	Returns the number of times *k* occurs in the map (1 or 0).
bool empty() const;	Returns true if the invoking map is empty and false otherwise.
iterator end(); const_iterator end() const;	Returns an iterator to the end of the map.

TABLE 14-4 *The **map** Member Functions* ▼

Member Function	Description
pair<iterator, iterator> equal_range(const key_type &*k*); pair<const_iterator, const_iterator> equal_range(const key_type &*k*) const;	Returns a pair of iterators that point to the first and last elements in the map that contain the specified key.
void erase(iterator *i*);	Removes the element pointed to by *i*.
void erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*.
size_type erase(const key_type &*k*)	Removes elements that have keys with the value *k*.
iterator find(const key_type &*k*); const_iterator find(const key_type &*k*) const;	Returns an iterator to the specified key. If the key is not found, an iterator to the end of the map is returned.
allocator_type get_allocator( ) const;	Returns the map's allocator.
iterator insert(iterator *i*, const value_type &*val*);	Inserts *val* at or after the element specified by *i*. An iterator to the element is returned.
template <class InIter> void insert(InIter *start*, InIter *end*)	Inserts a range of elements.
pair<iterator, bool> insert(const value_type &*val*);	Inserts *val* into the invoking map. An iterator to the element is returned. The element is inserted only if it does not already exist. If the element was inserted, **pair<iterator, true>** is returned. Otherwise, **pair<iterator, false>** is returned.
key_compare key_comp( ) const;	Returns the function object that compares keys.
iterator lower_bound(const key_type &*k*); const_iterator lower_bound(const key_type &*k*) const;	Returns an iterator to the first element in the map with the key equal to or greater than *k*.
size_type max_size( ) const;	Returns the maximum number of elements that the map can hold.
reference operator[ ](const key_type &*i*)	Returns a reference to the element specified by *i*. If this element does not exist, it is inserted.
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	Returns a reverse iterator to the end of the map.
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	Returns a reverse iterator to the start of the map.

TABLE 14-4 *The **map** Member Functions (continued)* ▼

Member Function	Description
size_type size( ) const;	Returns the number of elements currently in the map.
void swap(map<Key, T, Comp, Allocator> &*ob*)	Exchanges the elements stored in the invoking map with those in *ob*.
iterator upper_bound(const key_type &*k*); const_iterator upper_bound(const key_type &*k*) const;	Returns an iterator to the first element in the map with the key greater than *k*.
value_compare value_comp( ) const;	Returns the function object that compares values.

TABLE 14-4 *The* ***map*** *Member Functions (continued)* ▼

Key/value pairs are stored in a map as objects of type **pair**, which has this template specification:

```
template <class Ktype, class Vtype> struct pair {
  typedef Ktype first_type; // type of key
  typedef Vtype second_type; // type of value
  Ktype first; // contains the key
  Vtype second; // contains the value

  // constructors
  pair();
  pair(const Ktype &k, const Vtype &v);
  template<class A, class B> pair(const<A, B> &ob);
}
```

As the comments suggest, the value in **first** contains the key and the value in **second** contains the value associated with that key.

You can construct a pair using either one of **pair**'s constructors or by using **make_pair( )**, which constructs a **pair** object based upon the types of the data used as parameters. **make_pair( )** is a generic function that has this prototype:

```
template <class Ktype, class Vtype>
  pair<Ktype, Vtype> make_pair(const Ktype &k, const Vtype &v);
```

As you can see, it returns a **pair** object consisting of values of the types specified by *Ktype* and *Vtype*. The advantage of **make_pair( )** is that it allows the types of the objects being stored to be determined automatically by the compiler rather than being explicitly specified by you.

EXAMPLES

1. The following program illustrates the basics of using a map. It stores ten key/value pairs. The key is a character and the value is an integer. The key/value pairs stored are

A	0
B	1
C	2

and so on. Once the pairs have been stored, the user is prompted for a key (i.e., a letter from A through J), and the value associated with that key is displayed.

```
// A simple map demonstration.
#include <iostream>
#include <map>
using namespace std;

int main()
{
  map<char, int> m;
  int i;

  // put pairs into map
  for(i=0; i<10; i++) {
    m.insert(pair<char, int>('A'+i, i));
  }

  char ch;
  cout << "Enter key: ";
  cin >> ch;

  map<char, int>::iterator p;

  // find value given key
```

```
  p = m.find(ch);
  if(p != m.end())
    cout << p->second;
  else
    cout << "Key not in map.\n";

  return 0;
}
```

Notice the use of the **pair** template class to construct the key/value pairs. The data types specified by **pair** must match those of the map into which the pairs are being inserted.

Once the map has been initialized with keys and values, you can search for a value given its key by using the **find( )** function. **find( )** returns an iterator to the matching element or to the end of the map if the key is not found. When a match is found, the value associated with the key is contained in the **second** member of **pair**.

2. In the preceding example, key/value pairs were constructed explicitly, using **pair<char, int>**. Although there is nothing wrong with this approach, it is often easier to use **make_pair( )**, which constructs a **pair** object based upon the types of the data used as parameters. For example, assuming the previous program, this line of code will also insert key/value pairs into **m**:

```
m.insert(make_pair((char)('A'+i), i));
```

Here the cast to **char** is needed to override the automatic conversion to **int** when **i** is added to 'A'. Otherwise, the type determination is automatic.

3. Like all of the containers, maps can be used to store objects of types that you create. For example, the program shown here creates a map of words with their opposites. To do this it creates two classes called **word** and **opposite**. Since a map maintains a sorted list of keys, the program also defines the < operator for objects of type **word**. In general, you must define the < operator for any classes that you will use as keys. (Some compilers might require that additional comparison operators be defined.)

```
// A map of opposites.
#include <iostream>
```

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

class word {
  char str[20];
public:
  word() { strcpy(str, ""); }
  word(char *s) { strcpy(str, s); }
  char *get() { return str; }

};

// must define less than relative to word objects
bool operator<(word a, word b)
{
   return strcmp(a.get(), b.get()) < 0;
}

class opposite {
  char str[20];
public:
  opposite() { strcmp(str, ""); }
  opposite(char *s) { strcpy(str, s); }
  char *get() { return str; }
};


int main()
{
  map<word, opposite> m;

  // put words and opposites into map
  m.insert(pair<word,
    opposite>(word("yes"), opposite("no")));
  m.insert(pair<word,
    opposite>(word("good"), opposite("bad")));
  m.insert(pair<word,
    opposite>(word("left"), opposite("right")));
  m.insert(pair<word,
    opposite>(word("up"), opposite("down")));

  // given a word, find opposite
  char str[80];
  cout << "Enter word: ";
```

```
  cin >> str;

  map<word, opposite>::iterator p;

  p = m.find(word(str));
  if(p != m.end())
    cout << "Opposite: " <<  p->second.get();
  else
    cout << "Word not in map.\n";

  return 0;
}
```

In this example, each entry in the map is a character array that holds a null-terminated string. Later in this chapter, you will see an easier way to write this program that uses the standard **string** type.

EXERCISES

1. Experiment with the examples, trying small variations.
2. Create a map that contains names and telephone numbers. Allow names and numbers to be entered, and set up your program so that a number can be found when a name is given. (Hint: Use Example 3 as a model.)
3. Do you need to define the < operator for objects used as keys in a map?

14.6 ALGORITHMS

As explained, algorithms act on containers. Although each container provides support for its own basic operations, the standard algorithms provide more extended or complex actions. They also allow you to work with two different types of containers at the same time. To have access to the STL algorithms, you must include **<algorithm>** in your program.

The STL defines a large number of algorithms, which are summarized in Table 14-5. All of the algorithms are template functions. This means that they can be applied to any type of container. The examples that follow demonstrate a representative sample.

Algorithm	Purpose
adjacent_find	Searches for adjacent matching elements within a sequence and returns an iterator to the first match.
binary_search	Performs a binary search on an ordered sequence.
copy	Copies a sequence.
copy_backward	Same as **copy( )** except that it moves the elements from the end of the sequence first.
count	Returns the number of elements in the sequence.
count_if	Returns the number of elements in the sequence that satisfy some predicate.
equal	Determines whether two ranges are the same.
equal_range	Returns a range in which an element can be inserted into a sequence without disrupting the ordering of the sequence.
fill fill_n	Fills a range with the specified value.
find	Searches a range for a value and returns an iterator to the first occurrence of the element.
find_end	Searches a range for a subsequence. This function returns an iterator to the end of the subsequence within the range.
find_first_of	Finds the first element within a sequence that matches an element within a range.
find_if	Searches a range for an element for which a user-defined unary predicate returns true.
for_each	Applies a function to a range of elements.
generate generate_n	Assigns to elements in a range the values returned by a generator function.
includes	Determines whether one sequence includes all of the elements in another sequence.
inplace_merge	Merges a range with another range. Both ranges must be sorted in increasing order. The resulting sequence is sorted.

TABLE 14-5 *The STL Algorithms* ▼

Algorithm	Purpose
iter_swap	Exchanges the values pointed to by its two iterator arguments.
lexicographical_compare	Alphabetically compares one sequence with another.
lower_bound	Finds the first point in the sequence that is not less than a specified value.
make_heap	Constructs a heap from a sequence.
max	Returns the maximum of two values.
max_element	Returns an iterator to the maximum element within a range.
merge	Merges two ordered sequences, placing the result into a third sequence.
min	Returns the minimum of two values.
min_element	Returns an iterator to the minimum element within a range.
mismatch	Finds the first mismatch between the elements in two sequences. Iterators to the two elements are returned.
next_permutation	Constructs the next permutation of a sequence.
nth_element	Arranges a sequence such that all elements less than a specified element *E* come before that element and all elements greater than *E* come after it.
partial_sort	Sorts a range.
partial_sort_copy	Sorts a range and then copies as many elements as will fit into a result sequence.
partition	Arranges a sequence such that all elements for which a predicate returns true come before those for which the predicate returns false.
pop_heap	Exchanges the first and last-1 elements and then rebuilds the heap.
prev_permutation	Constructs the previous permutation of a sequence.
push_heap	Pushes an element onto the end of a heap.
random_shuffle	Randomizes a sequence.
remove remove_if remove_copy remove_copy_if	Removes elements from a specified range.
replace replace_copy replace_if replace_copy_if	Replaces elements within a range.

TABLE 14-5 *The STL Algorithms (continued)* ▼

Algorithm	Purpose
reverse reverse_copy	Reverses the order of a range.
rotate rotate_copy	Left-rotates the elements in a range.
search	Searches for a subsequence within a sequence.
search_n	Searches for a sequence of a specified number of similar elements.
set_difference	Produces a sequence that contains the difference between two ordered sets.
set_intersection	Produces a sequence that contains the intersection of two ordered sets.
set_symmetric_difference	Produces a sequence that contains the symmetric difference between two ordered sets.
set_union	Produces a sequence that contains the union of two ordered sets.
sort	Sorts a range.
sort_heap	Sorts a heap within a specified range.
stable_partition	Arranges a sequence such that all elements for which a predicate returns true come before those for which the predicate returns false. The partitioning is stable; the relative ordering of the sequence is preserved.
stable_sort	Sorts a range. The sort is stable; equal elements are not rearranged.
swap	Exchanges two values.
swap_ranges	Exchanges elements in a range.
transform	Applies a function to a range of elements and stores the outcome in a new sequence.
unique unique_copy	Eliminates duplicate elements from a range.
upper_bound	Finds the last point in a sequence that is not greater than some value.

EXAMPLES

1. Two of the simplest algorithms are **count( )** and **count_if( )**. Their general forms are shown here:

```
template <class InIter, class T>
    size_t count(InIter start, InIter end, const T &val);

template <class InIter, class T>
    size_t count(InIter start, InIter end, UnPred pfn);
```

The **count( )** algorithm returns the number of elements in the sequence beginning at *start* and ending at *end* that match *val*. The **count_if( )** algorithm returns the number of elements in the sequence beginning at *start* and ending at *end* for which the unary predicate *pfn* returns true.

The following program demonstrates **count( )** and **count_if( )**.

```
// Demonstrate count and count_if.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

/* This is a unary predicate that determines
   if a value is even. */
bool even(int x)
{
  return !(x%2);
}

int main()
{
  vector<int> v;
  int i;


  for(i=0; i<20; i++) {
    if(i%2) v.push_back(1);
    else v.push_back(2);
  }
```

```
  cout << "Sequence: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  int n;
  n = count(v.begin(), v.end(), 1);
  cout << n << " elements are 1\n";

  n = count_if(v.begin(), v.end(), even);
  cout << n << " elements are even.\n";

  return 0;
}
```

This program displays the following output:

```
Sequence: 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
10 elements are 1
10 elements are even.
```

The program begins by creating a 20-element vector that contains alternating 1s and 2s. Next, **count( )** is used to count the number of 1s. Then, **count_if( )** counts the number of elements that are even. Notice how the unary predicate **even( )** is coded. All unary predicates receive as a parameter an object that is of the same type as that stored in the container upon which the predicate is operating. The predicate must then return a true or false result based upon this object.

2. Sometimes it is useful to generate a new sequence that consists of only certain items from an original sequence. One algorithm that does this is **remove_copy( )**. Its general form is shown here:

```
template <class InIter, class OutIter, class T>
  OutIter remove_copy(InIter start, InIter end,
                      OutIter result, const T &val);
```

The **remove_copy( )** algorithm copies elements from the specified range that are equal to *val* and puts the result into the sequence pointed to by *result*. It returns an iterator to the end of the result. The output container must be large enough to hold the result.

The following program demonstrates **remove_copy( )**. It creates a sequence of 1s and 2s. It then removes all of the 1s from the sequence.

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

int main()
{
  vector<int> v, v2(20);
  int i;

  for(i=0; i<20; i++) {
    if(i%2) v.push_back(1);
    else v.push_back(2);
  }

  cout << "Sequence: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  // Remove 1s
  remove_copy(v.begin(), v.end(), v2.begin(), 1);
  cout << "Result: ";
  for(i=0; i<v2.size(); i++) cout << v2[i] << " ";
  cout << endl << endl;

  return 0;
}
```

The output produced by this program is shown here.

```
Sequence: 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
Result: 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0
```

3. An often useful algorithm is **reverse( )**, which reverses a sequence. Its general form is

   template <class Bilter> void reverse(Bilter *start*, Bilter *end*);

   The **reverse( )** algorithm reverses the order of the range specified by *start* and *end*.

The following program demonstrates **reverse( )**:

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

int main()
{
  vector<int> v;
  int i;

  for(i=0; i<10; i++) v.push_back(i);

  cout << "Initial: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  reverse(v.begin(), v.end());

  cout << "Reversed: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";

  return 0;
}
```

The output from this program is shown here.

```
Initial: 0 1 2 3 4 5 6 7 8 9
Reversed: 9 8 7 6 5 4 3 2 1 0
```

4. One of the more interesting algorithms is **transform( )**, which modifies each element in a range according to a function that you provide. The **transform( )** algorithm has these two general forms:

```
template <class InIter, class OutIter, class Func)
  OutIter transform(InIter start, InIter end, OutIter result, Func unaryfunc);

template <class InIter1, class InIter2, class OutIter, class Func)
  OutIter transform(InIter1 start1, InIter1 end1, InIter2 start2,
                    OutIter result, Func binaryfunc);
```

The **transform( )** algorithm applies a function to a range of elements and stores the outcome in *result*. In the first form, the range is specified by *start* and *end*. The function to be applied is specified by *unaryfunc*. This function receives the value of an element in its parameter, and it must return the element's transformation. In the second form, the transformation is applied using a binary operator function that receives the value of an element from the sequence to be transformed in its first parameter and an element from a second sequence as its second parameter. Both versions return an iterator to the end of the resulting sequence.

The following program uses a simple transformation function called **xform( )** to square the contents of a list. Notice that the resulting sequence is stored in the same list that provided the original sequence.

```
// An example of the transform algorithm.
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

// A simple transformation function.
int xform(int i) {
  return i * i; // square original value
}

int main()
{
  list<int> xl;
  int i;

  // put values into list
  for(i=0; i<10; i++) xl.push_back(i);

  cout << "Original contents of xl: ";
  list<int>::iterator p = xl.begin();
  while(p != xl.end()) {
    cout << *p << " ";
    p++;
  }

  cout << endl;
```

```
  // transform xl
  p = transform(xl.begin(), xl.end(), xl.begin(), xform);

  cout << "Transformed contents of xl: ";
  p = xl.begin();
  while(p != xl.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

The output produced by the program is shown here:

```
Original contents of xl: 0 1 2 3 4 5 6 7 8 9
Transformed contents of xl: 0 1 4 9 16 25 36 49 64 81
```

As you can see, each element in the **xl** list has been squared.

EXERCISES

1. The **sort( )** algorithm has these forms:

   template <class RandIter> void sort(RandIter *start*, RandIter *end*);
   template <class RandIter, class Comp>
   void sort(RandIter *start*, RandIter *end*, Comp *cmpfn*);

   It sorts the range specified by *start* and *end*. The second form allows you to specify a comparison function that determines whether one element is less than another. Write a program that demonstrates **sort( )**. (Use either form you like.)

2. The **merge( )** algorithm merges two ordered sequences and places the result into a third. One of its general forms is shown here:

   template <class InIter1, class InIter2, class OutIter>
   OutIter merge(InIter1 *start1*, InIter1 *end1*, InIter2 *start2*, InIter2 *end2*,
   OutIter *result*);

The sequences to be merged are defined by *start1*, *end1* and *start2*, *end2*. The result is put into the sequence pointed to by *result*. An iterator to the end of the resulting sequence is returned. Demonstrate this algorithm.

14.7 THE STRING CLASS

As you know, C++ does not support a built-in string type, per se. It does, however, provide two ways to handle strings. First, you can use the traditional, null-terminated character array with which you are already familiar. This is sometimes referred to as a *C string*. The second method is to use a class object of type **string**. This is the approach that is examined here.

Actually, the **string** class is a specialization of a more general template class called **basic_string**. In fact, there are two specializations of **basic_string**: **string**, which supports 8-bit character strings, and **wstring**, which supports wide character strings. Since 8-bit characters are by far the most commonly used characters in normal programming, **string** is the version of **basic_string** examined here.

Before you look at the **string** class, it is important that you understand why it is part of the C++ library. Standard classes have not been casually added to C++. In fact, a significant amount of thought and debate has accompanied each new addition. Given that C++ already contains some support for strings as null-terminated character arrays, it might at first seem that the inclusion of the **string** class is an exception to this rule. However, this is actually far from the truth. Here is why: Null-terminated strings cannot be manipulated by any of the standard C++ operators, nor can they take part in normal C++ expressions. For example, consider this fragment:

```
char s1[80], s2[80], s3[80];

s1 = "one"; // can't do
s2 = "two"; // can't do
s3 = s1 + s2; // error, not allowed
```

As the comments show, in C++ it is not possible to use the assignment operator to give a character array a new value (except during initialization), nor is it possible to use the + operator to concatenate two strings. These operations must be written using library functions. as shown here:

```
strcpy(s1, "one");
strcpy(s2, "two");
strcpy(s3, s1);
strcat(s3, s2);
```

Since null-terminated character arrays are not technically data types in their own right, the C++ operators cannot be applied to them. This makes even the most rudimentary string operations clumsy. More than anything else, it is the inability to operate on null-terminated strings using the standard C++ operators that has driven the development of a standard **string** class. Remember, when you define a class in C++, you are defining a new data type that can be fully integrated into the C++ environment. This, of course, means that the operators can be overloaded relative to the new class. Therefore, with the addition of a standard **string** class, it becomes possible to manage strings in the same way that any other type of data is managed: through the use of operators.

There is, however, one other reason for the standard **string** class: safety. An inexperienced or careless programmer can very easily overrun the end of an array that holds a null-terminated string. For example, consider the standard string copy function **strcpy( )**. This function contains no provision for checking the boundary of the target array. If the source array contains more characters than the target array can hold, a program error or system crash is possible (likely). As you will see, the standard **string** class prevents such errors.

In the final analysis, there are three reasons for the inclusion of the standard **string** class: consistency (a string now defines a data type), convenience (you can use the standard C++ operators), and safety (array boundaries will not be overrun). Keep in mind that there is no reason that you should abandon normal, null-terminated strings altogether. They are still the most efficient way in which to implement strings. However, when speed is not an overriding concern, the new **string** class gives you access to a safe and fully integrated way to manage strings.

Although not traditionally thought of as part of the STL, **string** is another container class defined by C++. This means that it supports the algorithms described in the previous section. However, strings have additional capabilities. To have access to the **string** class you must include **<string>** in your program.

The **string** class is very large, with many constructors and member functions. Also, many member functions have multiple overloaded forms. For this reason, it is not possible to look at the entire contents of **string** in this chapter. Instead, we will examine several of its most commonly used features. Once you have a general understanding of how **string** works, you will be able to easily explore the rest of it on your own.

The **string** class supports several constructors. The prototypes for three of its most commonly used constructors are shown here.

```
string();
string(const char *str);
string(const string &str);
```

The first form creates an empty **string** object. The second creates a **string** object from the null-terminated string pointed to by *str*. This form provides a conversion from null-terminated strings to **string** objects. The third form creates a **string** object from another **string** object.

A number of operators that apply to strings are defined for **string** objects, including those listed in the following table:

Operator	Meaning
=	Assignment
+	Concatenation
+=	Concatenation assignment
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
[]	Subscripting

Operator	Meaning
<<	Output
>>	Input

These operators allow the use of **string** objects in normal expressions and eliminate the need for calls to functions such as **strcpy(.)** or **strcat( )**, for example. In general, you can mix **string** objects with normal, null-terminated strings in expressions. For example, a **string** object can be assigned a null-terminated string.

The + operator can be used to concatenate a **string** object with another **string** object or a **string** object with a C-style string. That is, the following variations are supported:

```
string + string
string + C-string
C-string + string
```

The + operator can also be used to concatenate a character onto the end of a string.

The **string** class defines the constant **npos**, which is usually -1. This constant represents the length of the longest possible string.

Although most simple string operations can be accomplished with the string operators, more complex or subtle ones are accomplished with **string** class member functions. Although there are far too many to discuss in this chapter, we will examine several of the most common ones. To assign one string to another, use the **assign( )** function. Two of its forms are shown here:

```
string &assign(const string &strob, size_type start, size_type num);

string &assign(const char *str, size_type num);
```

In the first form, *num* characters from *strob* beginning at the index specified by *start* will be assigned to the invoking object. In the second form, the first *num* characters of the null-terminated string *str* are assigned to the invoking object. In each case, a reference to the invoking object is returned. Of course, it is much easier to use the = operator to assign one entire string to another. You will need to use the **assign( )** function only when assigning a partial string.

You can append part of one string to another using the **append( )** member function. Two of its forms are shown here:

```
string &append(const string &strob, size_type start, size_type num);

string &append(const char *str, size_type num);
```

In the first form, *num* characters from *strob*, beginning at the index specified by *start*, will be appended to the invoking object. In the second form, the first *num* characters of the null-terminated string *str* are appended to the invoking object. In each case, a reference to the invoking object is returned. Of course, it is much easier to use the + operator to append one entire string to another. You will need to use the **append( )** function only when appending a partial string.

You can insert or replace characters within a string using **insert( )** and **replace( )**. The prototypes for their most common forms are shown here:

```
string &insert(size_type start, const string &strob);

string &insert(size_type start, const string &strob,
               size_type insStart, size_type num);

string &replace(size_type start, size_type num, const string &strob);

string &replace(size_type start, size_type orgNum, const string &strob,
                size_type replaceStart, size_type replaceNum);
```

The first form of **insert( )** inserts *strob* into the invoking string at the index specified by *start*. The second form of **insert( )** inserts *num* characters from *strob* beginning at *insStart* into the invoking string at the index specified by *start*.

Beginning at *start*, the first form of **replace( )** replaces *num* characters from the invoking string with *strob*. The second form replaces *orgNum* characters, beginning at *start* in the invoking string, with *replaceNum* characters from the string specified by *strob* beginning at *replaceStart*. In both cases, a reference to the invoking object is returned.

You can remove characters from a string using **erase( )**. One of its forms is shown here:

```
string &erase(size_type start = 0, size_type num = npos);
```

It removes *num* characters from the invoking string beginning at *start*. A reference to the invoking string is returned.

The **string** class provides several member functions, including **find( )** and **rfind( )**, that search a string. Here are the prototypes for the most common versions of these two functions:

```
size_type find(const string &strob, size_type start=0) const;

size_type rfind(const string &strob, size_type start=npos) const;
```

Beginning at *start*, **find( )** searches the invoking string for the first occurrence of the string contained in *strob*. If the search string is found, **find( )** returns the index at which the match occurs within the invoking string. If no match is found, **npos** is returned. **rfind( )** is the opposite of **find( )**. Beginning at *start*, it searches the invoking string in the reverse direction for the first occurrence of the string contained in *strob*. (I.e, it finds the last occurrence of *strob* within the invoking string.) If the search string is found, **rfind( )** returns the index at which the match occurs within the invoking string. If no match is found, **npos** is returned.

To compare the entire contents of one string object to those of another, you will normally use the overloaded relational operators described earlier. However, if you want to compare a portion of one string to another, you will need to use the **compare( )** member function, shown here:

```
int compare(size_type start, size_type num, const string &strob) const;
```

Here *num* characters in *strob*, beginning at *start*, will be compared against the invoking string. If the invoking string is less than *strob*, **compare( )** will return less than 0. If the invoking string is greater than *strob*, it will return greater than 0. If *strob* is equal to the invoking string, **compare( )** will return 0.

Although **string** objects are useful in their own right, there will be times when you will need to obtain a null-terminated character array version of the string. For example, you might use a **string** object to construct a file name. However, when opening a file, you will need to specify a pointer to a standard, null-terminated string. To solve this problem, the member function **c_str( )** is provided. Its prototype is shown here:

```
const char *c_str() const;
```

This function returns a pointer to a null-terminated version of the string contained in the invoking **string** object. The null-terminated string must not be altered. It is also not guaranteed to be valid after any other operations have taken place on the **string** object.

Because it is a container, **string** supports the **begin( )** and **end( )** functions that return an iterator to the start and end of a string, respectively. Also included is the **size( )** function, which returns the number of characters currently in a string.

EXAMPLES

1. Although the traditional, C-style strings have always been simple to use, the C++ **string** class makes string handling extraordinarily easy. For example, with **string** objects you can use the assignment operator to assign a quoted string to a **string**, the + operator to concatenate strings, and the comparison operators to compare strings. The following program illustrates these operations.

```
// A short string demonstration.
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("Demonstrating Strings");
  string str2("String Two");
  string str3;

  // assign a string
  str3 = str1;
  cout << str1 << "\n" << str3 << "\n";

  // concatenate two strings
  str3 = str1 + str2;
  cout << str3 << "\n";

  // compare strings
  if(str3 > str1) cout << "str3 > str1\n";
  if(str3 == str1+str2)
    cout << "str3 == str1+str2\n";
```

```
  /* A string object can also be
     assigned a normal string. */
  str1 = "This is a normal string.\n";
  cout << str1;

  // create a string object using another string object
  string str4(str1);
  cout << str4;

  // input a string
  cout << "Enter a string: ";
  cin >> str4;
  cout << str4;

  return 0;
}
```

This program produces the following output:

```
Demonstrating Strings
Demonstrating Strings
Demonstrating StringsString Two
str3 > str1
str3 == str1+str2
This is a normal string.
This is a normal string.
Enter a string: Hello
Hello
```

As you can see, objects of type **string** can be manipulated with techniques similar to those used to work with C++'s built-in data types. In fact, this is the main advantage to the string class.

Notice the ease with which the string handling is accomplished. For example, the + is used to concatenate strings and the > is use to compare two strings. To accomplish these operations using C-style, null-terminated strings, you must use the less convenient calls to the **strcat( )** and **strcmp( )** functions. Because C++ **string** objects can be freely mixed with C-style, null-terminated strings, there is no disadvantage to using them in your program—and there are considerable benefits to be gained.

There is one other thing to notice in the preceding program: the sizes of the strings are not specified. A **string** object is

automatically sized to hold the string that it is given. Thus, when you are assigning or concatenating strings, the target string will grow as needed to accommodate the size of the new string. It is not possible to overrun the end of the string. This dynamic aspect of **string** objects is one of the reasons that they are considered better than standard null-terminated strings (which *are* subject to boundary overruns).

2. The following program demonstrates the **insert( )**, **erase( )**, and **replace( )** functions.

```
// Demonstrate insert(), erase(), and replace().
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("This is a test");
  string str2("ABCDEFG");

  cout << "Initial strings:\n";
  cout << "str1: " << str1 << endl;
  cout << "str2: " << str2 << "\n\n";

  // demonstrate insert()
  cout << "Insert str2 into str1:\n";
  str1.insert(5, str2);
  cout << str1 << "\n\n";

  // demonstrate erase()
  cout << "Remove 7 characters from str1:\n";
  str1.erase(5, 7);
  cout << str1 <<"\n\n";

  // demonstrate replace
  cout << "Replace 2 characters in str1 with str2:\n";
  str1.replace(5, 2, str2);
  cout << str1 << endl;

  return 0;
}
```

The output produced by this program is shown here:

```
Initial strings:
str1: This is a test
```

```
str2: ABCDEFG

Insert str2 into str1:
This ABCDEFGis a test

Remove 7 characters from str1:
This is a test

Replace 2 characters in str1 with str2:
This ABCDEFG a test
```

3. Since **string** defines a data type, it is possible to create containers that hold objects of type **string**. For example, here is a better way to write the word/opposite program shown in Example 3 of section 14.5.

```
// A map of word opposites, using strings.
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  map<string, string> m;
  int i;

  m.insert(pair<string, string>("yes", "no"));
  m.insert(pair<string, string>("up", "down"));
  m.insert(pair<string, string>("left", "right"));
  m.insert(pair<string, string>("good", "bad"));

  string s;
  cout << "Enter word: ";
  cin >> s;

  map<string, string>::iterator p;

  p = m.find(s);
  if(p != m.end())
    cout << "Opposite: " << p->second;
  else
    cout << "Word not in map.\n";

  return 0;
```

EXERCISES

1. Using objects of type **string**, store the following strings in a **list**.

one	two	three	four	five
six	seven	eight	nine	ten

   Next, sort the list. Finally, display the sorted list.
2. Since **string** is a container, it can be used with the standard algorithms. Create a program that inputs a string from the user. Then, using **count( )**, report how many e's are in the string.
3. Modify your answer to Exercise 2 so that it reports the number of characters that are lowercase. (Hint: use **count_if( )**.)
4. The **string** class is a specialization of what template class?

SKILLS CHECK



At this point you should be able to perform the following exercises and answer the questions.

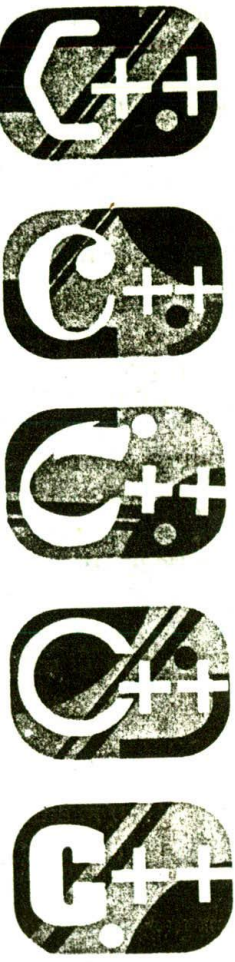
1. How does the STL make it easier for you to create more reliable programs?
2. Define a container, an iterator, and an algorithm as they relate to the STL.
3. Write a program that creates a ten-element vector that contains the numbers 1 through 10. Next, copy only the even elements from this vector into a list.
4. Give one advantage of using the **string** data type. Give one disadvantage.
5. What is a predicate?

6. On your own, recode your answer to Exercise 2 in Section 14.5 so that it uses **string**.
7. Begin exploring the STL function objects. To get started, examine the two standard classes **unary_function** and **binary_function**, which aid in the construction of function objects.
8. Study the STL documentation that comes with your compiler. You will find several interesting features and techniques.



This section checks how well you have integrated material in this chapter with that from the preceding chapters.

1. You have come a long way since Chapter 1. Take some time to skim through the book again. As you do so, think about ways you can improve the examples (especially those in the first six chapters) so that they take advantage of all the features of C++ you have learned.
2. Programming is learned best by doing. Write many C++ programs. Try to exercise those features of C++ that are unique to it.
3. Continue to explore the STL. In the future, many programming tasks will be framed in terms of the STL because often a program can be reduced to manipulations of containers by algorithms.
4. Finally, remember: C++ gives you unprecedented power. It is important that you learn to use this power wisely. Because of this power, C++ lets you push the limits of your programming ability. However, if this power is misused, you can also create programs that are hard to understand, nearly impossible to follow, and extremely difficult to maintain. C++ is a powerful tool. But, like any other tool, it is only as good as the person using it.

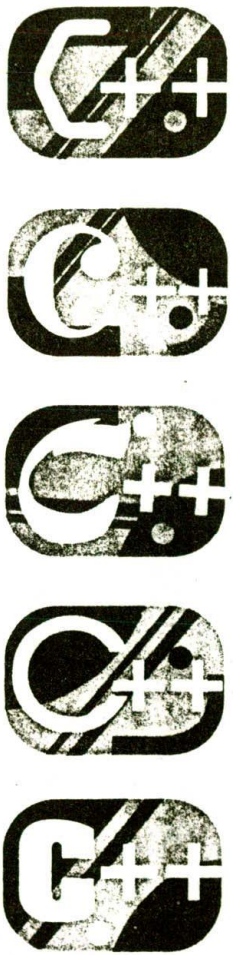


A

A Few More Differences Between C and C++

FOR the most part, C++ is a superset of ANSI-standard C, and virtually all C programs are also C++ programs. However, a few differences do exist, several of which were mentioned in Chapter 1. Here are a few more that you should be aware of:

- A small but potentially important difference between C and C++ is that in C, a character constant is automatically elevated to an integer, whereas in C++, it is not.
- In C, it is not an error to declare a global variable several times, even though it is bad programming practice. In C++, this is an error.
- In C, an identifier will have at least 31 significant characters. In C++, all characters are considered significant. However, from a practical point of view, extremely long identifiers are unwieldy and are seldom needed.
- In C, you can call **main( )** from within a program, although this would be unusual. In C++, this is not allowed.
- In C, you cannot take the address of a **register** variable. In C++, you can.
- In C, the type **wchar_t** is defined with a **typedef**. In C++, **wchar_t** is a keyword.



B

Answers

1.3 EXERCISES

1.
```
#include <iostream>
using namespace std;

int main()
{
  double hours, wage;

  cout << "Enter hours worked: ";
  cin >> hours;

  cout << "Enter wage per hour: ";
  cin >> wage;

  cout << "Pay is: $" << wage * hours;

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

int main()
{
  double feet;

  do {
     cout << "Enter feet (0 to quit): ";
     cin >> feet;

     cout << feet * 12 << " inches\n";
  } while (feet != 0.0);

  return 0;
}
```

3.
```
/*
   This program computes the lowest
   common denominator.
*/
#include <iostream>
using namespace std;

int main()
```

```
{
  int a, b, d, min;

  cout << "Enter two numbers: ";
  cin >> a >> b;

  min = a > b ? b : a;

  for(d = 2; d<min; d++)
    if(((a%d)==0) && ((b%d)==0)) break;
  if(d==min) {
    cout << "No common denominators\n";
    return 0;
  }
  cout << "The lowest common denominator is " << d << ".\n";

  return 0;
}
```

1.4 EXERCISES

1. The comment, although strange, is valid.

1.5 EXERCISES

2.
```
#include <iostream>
#include <cstring>
using namespace std;

class card {
  char title[80]; // book title
  char author[40]; // author
  int number; // number in library
public:
  void store(char *t, char *name, int num);
  void show();
};

void card::store(char *t, char *name, int num)
{
  strcpy(title, t);
  strcpy(author, name);
  number = num;
```

```
void card::show()
{
  cout << "Title: " << title << "\n";
  cout << "Author: " << author << "\n";
  cout << "Number on hand: " << number << "\n";
}

int main()
{
  card book1, book2, book3;

  book1.store("Dune", "Frank Herbert", 2);
  book2.store("The Foundation Trilogy", "Isaac Asimov", 2);
  book3.store("The Rainbow", "D. H. Lawrence", 1);

  book1.show();
  book2.show();
  book3.show();

  return 0;
}
```

3.
```
#include <iostream>
using namespace std;

#define SIZE 100

class q_type {
  int queue[SIZE]; // holds the queue
  int head, tail; // indices of head and tail
public:
  void init(); // initialize
  void q(int num); // store
  int deq();  // retrieve
};

// Initialize
void q_type::init()
{
  head = tail = 0;
}

// Put value on a queue.
```

```
void q_type::q(int num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "Queue is full\n";
    return;
  }
  tail++;
  if(tail==SIZE) tail = 0; // cycle around
  queue[tail] = num;
}

// Remove a value from a queue.
int q_type::deq()
{
  if(head == tail) {
    cout << "Queue is empty\n";
    return 0;  // or some other error indicator
  }
  head++;
  if(head==SIZE) head = 0; // cycle around
  return queue[head];
}

int main()
{
  q_type q1, q2;
  int i;

  q1.init();
  q2.init();

  for(i=1; i<=10; i++) {
    q1.q(i);
    q2.q(i*i);
  }

  for(i=1; i<=10; i++) {
    cout << "Dequeue 1: " << q1.deq() << "\n";
    cout << "Dequeue 2: " << q2.deq() << "\n";
  }

  return 0;
```

1.6 EXERCISES

1. The function f( ) is not prototyped.

1.7 EXERCISES

1.
```
#include <iostream>
#include <cmath>
using namespace std;

// Overload sroot() for integers, longs, and doubles.

int sroot(int i);
long sroot(long i);
double sroot(double i);

int main()
{
  cout << "Square root of 90.34 is : " << sroot(90.34);
  cout << "\n";
  cout << "Square root of 90L is : " << sroot(90L);
  cout << "\n";
  cout << "Square root of 90 is : " << sroot(90);

  return 0;
}

// Return square root of integer.
int sroot(int i)
{
  cout << "computing integer root\n";
  return (int) sqrt((double) i);
}

// Return square root of long.
long sroot(long i)
{
  cout << "computing long root\n";
  return (long) sqrt((double) i);
}

// Retur square root of double.
double sroot(double i)
```

```
{
  cout << "computing double root\n";
  return sqrt(i);
}
```

2. The functions **atof( )**, **atoi( )**, and **atol( )** cannot be overloaded because they differ only in the type of data they return. Function overloading requires that either the type or the number of arguments differs.

3.
```
// Overload the min() function.

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

char min(char a, char b);
int min(int a, int b);
double min(double a, double b);

int main()
{
  cout << "Min is: " << min('x', 'a') << "\n";
  cout << "Min is: " << min(10, 20) << "\n";
  cout << "Min is: " << min(0.2234, 99.2) << "\n";

  return 0;
}

// min() for chars
char min(char a, char b)
{
  return tolower(a)<tolower(b) ? a : b;
}

// min() for ints
int min(int a, int b)
{
  return a<b ? a : b;
}

// min() for doubles
double min(double a, double b)
{
```

```
    return a<b ? a : b;
}
```

4.
```
#include <iostream>
using namespace std;

// Overload sleep to accept integer or char * argument
void sleep(int n);
void sleep(char *n);

// Change this value to fit your processor speed.
#define DELAY 100000

int main()
{
  cout << '.';
  sleep(3);
  cout << '.';
  sleep("2");
  cout << '.';

  return 0;
}

// Sleep() with integer argument.
void sleep(int n)
{
  long i;

  for( ; n; n--)
    for(i=0; i<DELAY; i++) ;
}

// Sleep() with char * argument.
void sleep(char *n)
{
  long i;
  int j;

  j = atoi(n);

  for( ; j; j--)
    for(i=0; i<DELAY; i++) ;
```

MASTERY SKILLS CHECK: Chapter 1

1. Polymorphism is the mechanism by which one general interface can be used to access many specific implementations. Encapsulation provides a protected linkage between code and its related data. Access to encapsulated routines can be tightly controlled, thus preventing unwanted tampering. Inheritance is the process by which one object can acquire the traits of another. Inheritance is used to support a system of hierarchical classification.
2. You can include comments in a C++ program by using either the normal C-style comments or the C++-specific single-line comments.
3.
```
#include <iostream>
using namespace std;

int main()
{
  int b, e, r;
  cout << "Enter base: ";
  cin >> b;
  cout << "Enter exponent: ";
  cin >> e;

  r = 1;
  for( ; e; e--) r = r * b;

  cout << "Result: " << r;

  return 0;
}
```
4.
```
#include <iostream>
#include <cstring>
using namespace std;

// Overload string reversal function.
void rev_str(char *s); // reverse string in place
void rev_str(char *in, char *out); // put reversal into out

int main()
{
  char s1[80], s2[80];
```

```
  strcpy(s1, "This is a test");

  rev_str(s1, s2);
  cout << s2 << "\n";

  rev_str(s1);
  cout << s1 << "\n";

  return 0;
}

// Reverse string, put result in s.
void rev_str(char *s)
{
  char temp[80];
  int i, j;

  for(i=strlen(s)-1, j=0; i>=0; i--, j++)
    temp[j] = s[i];

  temp[j] = '\0'; // null-terminate result

  strcpy(s, temp);
}

// Reverse string, put result into out.
void rev_str(char *in, char *out)
{
  int i, j;

  for(i=strlen(in)-1, j=0; i>=0; i--, j++)
    out[j] = in[i];

  out[j] = '\0'; // null-terminate result
}
```

5.

```
#include <iostream.h>

int f(int a);

int main()
{
  cout << f(10);

  return 0;
```

```
int f(int a)
{
  return a * 3.1416;
}
```

6. The **bool** data type stores Boolean values. The only values an object of type **bool** can have are **true** and **false**.

REVIEW SKILLS CHECK: Chapter 2

1.
```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
  char s[80];

  cout << "Enter a string: ";
  cin >> s;

  cout << "Length: " << strlen(s) << "\n";

  return 0;
}
```

2.
```
#include <iostream>
#include <cstring>
using namespace std;

class addr {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[10];
public:
  void store(char *n, char *s, char *c, char *t,
             char *z);
  void display();
};
```

```
void addr::store(char *n, char *s, char *c, char *t,
                 char *z)
{
  strcpy(name, n);
  strcpy(street, s);
  strcpy(city, c);
  strcpy(state, t);
  strcpy(zip, z);
}

void addr::display()
{
  cout << name << "\n";
  cout << street << "\n";
  cout << city << "\n";
  cout << state << "\n";
  cout << zip << "\n\n";
}

int main()
{
  addr a;

  a.store("C. B. Turkle", "11 Pinetree Lane", "Wausau
          "In", "46576");

  a.display();

  return 0;
}
```

3.
```
#include <iostream>
using namespace std;

int rotate(int i);
long rotate(long i);

int main()
{
  int a;
  long b;

  a = 0x8000;
  b = 8;
```

```
  cout << rotate(a);
  cout << "\n";
  cout << rotate(b);

  return 0;
}

int rotate(int i)
{
  int x;

  if(i & 0x8000) x = 1;
  else x = 0;

  i = i << 1;
  i += x;

  return i;
}

long rotate(long i)
{
  int x;

  if(i & 0x80000000) x = 1;
  else x = 0;

  i = i << 1;
  i += x;

  return i;
}
```

4. The integer **i** is private to **myclass** and cannot be accessed inside **main( )**.

2.1 EXERCISES

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

   #define SIZE 100

   class q_type {
```

```
  int queue[SIZE]; // holds the queue
  int head, tail; // indices of head and tail
public:
  q_type(); // constructor
  void q(int num); // store
  int deq();  // retrieve
};

// Constructor
q_type::q_type()
{
  head = tail = 0;
}

// Put value on queue.
void q_type::q(int num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "Queue is full\n";
    return;
  }
  tail++;
  if(tail==SIZE) tail = 0; // cycle around
  queue[tail] = num;
}

// Remove value from queue.
int q_type::deq()
{
  if(head == tail) {
    cout << "Queue is empty\n";
    return 0;  // or some other error indicator
  }
  head++;
  if(head==SIZE) head = 0; // cycle around
  return queue[head];
}

int main()
{
  q_type q1, q2;
  int i;

  for(i=1; i<=10; i++) {
    q1.q(i);
```

```
    q2.q(i*i);
  }

  for(i=1; i<=10; i++) {
    cout << "Dequeue 1: " << q1.deq() << "\n";
    cout << "Dequeue 2: " << q2.deq() << "\n";
  }

  return 0;
}
```

2.
```
// Stopwatch emulator
#include <iostream>
#include <ctime>
using namespace std;

class stopwatch {
  double begin, end;
public:
  stopwatch();
  ~stopwatch();
  void start();
  void stop();
  void show();
};

stopwatch::stopwatch()
{
  begin = end = 0.0;
}

stopwatch::~stopwatch()
{
  cout << "Stopwatch object being destroyed...";
  show();
}

void stopwatch::start()
{
  begin = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::stop()
{
  end = (double) clock() / CLOCKS_PER_SEC;
```

```
}

void stopwatch::show()
{
  cout << "Elapsed time: " << end - begin;
  cout << "\n";
}

int main()
{
  stopwatch watch;
  long i;

  watch.start();
  for(i=0; i<320000; i++) ; // time a for loop
  watch.stop();

  watch.show();

  return 0;
}
```

3. A constructor cannot have a return type.

2.2 EXERCISES

1.
```
// Dynamically allocated stack.
#include <iostream>
#include <cstdlib>
using namespace std;

// Declare a stack class for characters
class stack {
  char *stck; // holds the stack
  int tos; // index of top of stack
  int size; // size of stack
public:
  stack(int s); // constructor
  ~stack(); // destructor
  void push(char ch); // push character on stack
  char pop(); // pop character from stack
};

// Initialize the stack
```

```
stack::stack(int s)
{
  cout << "Constructing a stack\n";
  tos = 0;
  stck = (char *) malloc(s);
  if(!stck) {
    cout << "Allocation error\n";
    exit(1);
  }
  size = s;
}

stack::~stack()
{
  free(stck);
}

// Push a character.
void stack::push(char ch)
{
  if(tos==size) {
    cout << "Stack is full\n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Pop a character.
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty\n";
    return 0; // return null on empty stack
  }
  tos--;
  return stck[tos];
}

int main()
{
  // Create two stacks that are automatically initialized.
  stack s1(10), s2(10);
  int i;
```

```
  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

  return 0;
}
```

2.
```
#include <iostream>
#include <ctime>
using namespace std;

class t_and_d {
  time_t systime;
public:
  t_and_d(time_t t); // constructor
  void show();
};

t_and_d::t_and_d(time_t t)
{
  systime = t;
}

void t_and_d::show()
{
  cout << ctime(&systime);
}

int main()
{
  time_t x;

  x = time(NULL);

  t_and_d ob(x);

  ob.show();
```

```
  return 0;
}
```

3.
```
#include <iostream>
using namespace std;

class box {
  double l, w, h;
  double volume;
public:
  box(double a, double b, double c);
  void vol();
};

box::box(double a, double b, double c)
{
  l = a;
  w = b;
  h = c;

  volume = l * w * h;
}

void box::vol()
{
  cout << "Volume is: " << volume << "\n";
}

int main()
{
  box x(2.2, 3.97, 8.09), y(1.0, 2.0, 3.0);

  x.vol();
  y.vol();
  return 0;
}
```

2.3 EXERCISE

1.
```
#include <iostream>
using namespace std;

class area_cl {
public:
```

```
  double height;
  double width;
};

class rectangle : public area_cl {
public:
  rectangle(double h, double w);
  double area();
};

class isosceles : public area_cl {
public:
  isosceles(double h, double w);
  double area();
};

rectangle::rectangle(double h, double w)
{
  height = h;
  width = w;
}

isosceles::isosceles(double h, double w)
{
  height = h;
  width = w;
}

double rectangle::area()
{
  return width * height;
}

double isosceles::area()
{
  return 0.5 * width * height;
}

int main()
{
  rectangle b(10.0, 5.0);
  isosceles i(4.0, 6.0);
```

```
  cout << "Rectangle: " << b.area() << "\n";
  cout << "Triangle: " << i.area() << "\n";

  return 0;
```

2.5 EXERCISES

1.
```
// Stack class using a structure.
#include <iostream>
using namespace std;

#define SIZE 10

// Declare a stack class for characters using a structure
struct stack {
  stack(); // constructor
  void push(char ch); // push character on stack
  char pop(); // pop character from stack
private:
  char stck[SIZE]; // holds the stack
  int tos; // index of top of stack
};

// Initialize the stack.
stack::stack()
{
  cout << "Constructing a stack\n";
  tos = 0;
}

// Push a character.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full\n";
    return;
  }
  stck[tos] = ch;
  tos++;
}
```

```
// Pop a character.
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty\n";
    return 0; // return null on empty stack
  }
  tos--;
  return stck[tos];
}

int main()
{
  // Create two stacks that are automatically initialized.
  stack s1, s2;
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

union swapbytes {
  unsigned char c[2];
  unsigned i;
  swapbytes(unsigned x);
  void swp();
};

swapbytes::swapbytes(unsigned x)
{
  i = x;
}
```

```
void swapbytes::swp()
{
  unsigned char temp;

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

int main()
{
  swapbytes ob(1);

  ob.swp();
  cout << ob.i;

  return 0;
}
```

3. An anonymous union is the syntactic mechanism that allows two variables to share the same memory space. The members of an anonymous union are accessed directly, without reference to an object. They are at the same scope level as the union itself.

2.6 EXERCISES

1.
```
#include <iostream>
using namespace std;

// Overload abs() three ways:

// abs() for ints
inline int abs(int n)
{
  cout << "In integer abs()\n";
  return n<0 ? -n : n;
}

// abs() for longs
inline long abs(long n)
{
  cout << "In long abs()\n";
  return n<0 ? -n : n;
```

```
// abs() for doubles
inline double abs(double n)
{
  cout << "In double abs()\n";
  return n<0 ? -n : n;
}

int main()
{
  cout << "Absolute value of -10: " << abs(-10) << "\n";
  cout << "Absolute value of -10L: " << abs(-10L) << "\n";
  cout << "Absolute value of -10.01: " << abs(-10.01) << "\n";

  return 0;
}
```

2. The function might not be able to be in-lined because it contains a **for** loop. Some compilers will not in-line functions containing loops.

2.7 EXERCISES

1.
```
#include <iostream>
using namespace std;

#define SIZE 10

// Declare a stack class for characters.
class stack {
  char stck[SIZE]; // holds the stack
  int tos; // index of top of stack
public:
  stack() { tos = 0; }
  void push(char ch)
  {
    if(tos==SIZE) {
      cout << "Stack is full\n";
      return;
    }
    stck[tos] = ch;
    tos++;
```

```
  char pop()
  {
    if(tos==0) {
      cout << "Stack is empty\n";
      return 0; // return null on empty stack
    }
    tos--;
    return stck[tos];
  }
};

int main()
{
  // Create two stacks that are automatically initialized.
  stack s1, s2;
  int i;

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

  return 0;
}
```

2.
```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr)
  {
    len = strlen(ptr);
    p = (char *) malloc(len+1);
    if(!p) {
```

```
      cout << "Allocation error\n";
      exit(1);
    }
    strcpy(p, ptr);
  }
  ~strtype() { cout << "Freeing p\n"; free(p); }

  void show()
  {
    cout << p << " - length: " << len;
    cout << "\n";
  }
};

int main()
{
  strtype s1("This is a test."), s2("I like C++.");

  s1.show();
  s2.show();

  return 0;
}
```

MASTERY SKILLS CHECK: Chapter 2

1. A constructor is the function that is called when an object is created. A destructor is the function that is called when an object is destroyed.

2.
```
#include <iostream>
using namespace std;

class line {
  int len;
public:
  line(int l);
};

line::line(int l)
{
  len = l;

  int i;
```

```
    for(i=0; i<len; i++) cout  << '*';
  }

  int main()
  {
    line l(10);

    return 0;
  }
```

3. 10 1000000 -0.0009

4.
```
#include <iostream>
using namespace std;

class area_cl {
public:
  double height;
  double width;
};

class rectangle : public area_cl {
public:
  rectangle(double h, double w) { height = h; width = w; }
  double area() { return height * width; }
};

class isosceles : public area_cl {
public:
  isosceles(double h, double w) { height = h; width = w; }
  double area() { return 0.5 * width * height; }};

class cylinder : public area_cl {
public:
  cylinder(double h, double w) { height = h; width = w; }
  double area()
  {
    return (2 * 3.1416 * (width/2) * (width/2)) +
           (3.1416 * width * height);
  }
};

int main()

  rectangle b(10.0, 5.0);
```

```
    isosceles i(4.0, 6.0);
    cylinder c(3.0, 4.0);

    cout << "Rectangle: " << b.area() << "\n";
    cout << "Triangle: " << i.area() << "\n";
    cout << "Cylinder: " << c.area() << "\n";

    return 0;
}
```

5. An in-line function's code is expanded in line. This means that the function is not actually called. This avoids the overhead associated with the function call and return mechanism. Its advantage is that it increases the execution speed. Its disadvantage is that it can increase the size of the program.

6.
```
#include <iostream>
using namespace std;

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

int main()
{
  myclass count(2, 3);

  count.show();

  return 0;
}
```

7. In a class, members are private by default. In a structure, members are public by default.

8. Yes. It defines an anonymous union.

CUMULATIVE SKILLS CHECK: Chapter

1.
```
#include <iostream>
using namespace std;
```

```
class prompt {
  int count;
public:
  prompt(char *s) { cout << s; cin >> count; };
  ~prompt();
};

prompt::~prompt() {
  int i, j;

  for(i=0; i<count; i++) {
    cout << '\a';
    for(j=0; j<32000; j++) ; // delay
  }
}

int main()
{
  prompt ob("Enter a number: ");

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class ftoi {
  double feet;
  double inches;
public:
  ftoi(double f);
};

ftoi::ftoi(double f)
{
  feet = f;
  inches = feet * 12;
  cout << feet << " is " << inches << " inches.\n";
}

int main()
{
  ftoi a(12.0), b(99.0);
```

```
  return 0;
}
```

3.
```
#include <iostream>
#include <cstdlib>
using namespace std;

class dice {
  int val;
public:
  void roll();
};

void dice::roll()
{
  val = (rand() % 6) + 1; // generate 1 through 6
  cout << val << "\n";
}

int main()
{
  dice one, two;

  one.roll();
  two.roll();
  one.roll();
  two.roll();
  one.roll();
  two.roll();

  return 0;
}
```

REVIEW SKILLS CHECK: Chapter 3

1. The constructor is called **widgit( )** and the destructor is called **~widgit( )**.
2. The constructor function is called when an object is created (that is, when an object comes into existence). The destructor is called when an object is destroyed.
3.
```
class Mars : public planet {
// ...
};
```

4. You can expand a function in line either by preceding its definition with the **inline** specifier or by including its definition within a class declaration.

5. An in-line function must be defined before it is first used. Other common restrictions include the following: It cannot contain any loops. It must not be recursive. It cannot contain a **goto** or a **switch** statement. It cannot contain any **static** variables.

6. `sample ob(100, 'X');`

3.1 EXERCISES

1. The assignment statement **x = y** is wrong because **cl1** and **cl2** are two different types of classes, and objects of differing class types cannot be assigned.

2.
```
#include <iostream>
using namespace std;

#define SIZE 100

class q_type {
  int queue[SIZE]; // holds the queue
  int head, tail; // indices of head and tail
public:
  q_type(); // constructor
  void q(int num); // store
  int deq();  // retrieve
};

// Constructor
q_type::q_type()
{
  head = tail = 0;
}

// Put value on queue.
void q_type::q(int num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "Queue is full\n";
    return;
  }
```

```
  tail++;
  if(tail==SIZE) tail = 0; // cycle around
  queue[tail] = num;
}

// Remove value from queue.
int q_type::deq()
{
  if(head == tail) {
    cout << "Queue is empty\n";
    return 0;  // or some other error indicator
  }
  head++;
  if(head==SIZE) head = 0; // cycle around
  return queue[head];
}

int main()
{
  q_type q1, q2;
  int i;

  for(i=1; i<=10; i++) {
    q1.q(i);
  }

  // assign one queue to another
  q2 = q1;

  // show that both have the same contents
  for(i=1; i<=10; i++)
    cout << "Dequeue 1: " << q1.deq() << "\n";

  for(i=1; i<=10; i++)
    cout << "Dequeue 2: " << q2.deq() << "\n";

  return 0;
}
```

3. If memory to hold a queue is dynamically allocated, assigning one queue to another causes the dynamic memory allocated to the queue on the left side of the assignment statement to be lost and the memory allocated to the queue on the right side to be freed twice when the objects are destroyed. Either of these two conditions is an unacceptable error.

3.2 EXERCISES

1.
```
#include <iostream>
using namespace std;

#define SIZE 10

// Declare a stack class for characters.
class stack {
  char stck[SIZE]; // holds the stack
  int tos; // index of top of stack
public:
  stack(); // constructor
  void push(char ch); // push character on stack
  char pop(); // pop character from stack
};

// Initialize the stack.
stack::stack()
{
  cout << "Constructing a stack\n";
  tos = 0;
}

// Push a character.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full\n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Pop a character.
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty\n";
    return 0; // return null on empty stack
  }
  tos--;
  return stck[tos];
}
```

```
void showstack(stack o);

int main()
{
  stack s1;
  int i;

  s1.push('a');
  s1.push('b');
  s1.push('c');

  showstack(s1);

  // s1 in main is still existent
  cout << "s1 stack still contains this: \n";
  for(i=0; i<3; i++) cout << s1.pop() << "\n";

  return 0;
}

// Display the contents of a stack.
void showstack(stack o)
{
  char c;

  // when this statement ends, the o stack is empty
  while(c=o.pop()) cout << c << "\n";
  cout << "\n";
}
```

This program displays the following:

```
Constructing a stack
c
b
a
Stack is empty
s1 stack still contains this:
c
b
a
```

2. The memory used to hold the integer pointed to by **p** in object **o** that is used to call **neg( )** is freed when the copy of **o** is destroyed when **neg( )** terminates, even though this memory is still needed by **o** inside **main( )**.

3.3 EXERCISES

1.
```
#include <iostream>
using namespace std;

class who {
  char name;
public:
  who(char c) {
     name = c;
     cout << "Constructing who #";
     cout << name << "\n";
  }
  ~who() { cout << "Destructing who #" << name << "\n"; }
};

who makewho()
{
  who temp('B');
  return temp;
}

int main()
{
  who ob('A');

  makewho();

  return 0;
}
```

2. There are several situations in which it would be improper to return an object. Here is one: if an object opens a disk file when it is created and closes that file when it is destroyed, if that object is returned from a function, the file will be closed when the temporary object is destroyed.

3.4 EXERCISE

1.
```
#include <iostream>
using namespace std;
```

```
class pr2;  // forward declaration

class pr1 {
  int printing;
  // ...
public:
  pr1() { printing = 0; }
  void set_print(int status) { printing = status; }
  // ...
  friend int inuse(pr1 o1, pr2 o2);
};

class pr2 {
  int printing;
  // ...
public:
  pr2() { printing = 0; }
  void set_print(int status) { printing = status; }
  // ...
  friend int inuse(pr1 o1, pr2 o2);
};

// Return true if printer is in use.
int inuse(pr1 o1, pr2 o2)
{
  if(o1.printing || o2.printing) return 1;
  else return 0;
}

int main()
{
  pr1 p1;
  pr2 p2;

  if(!inuse(p1, p2)) cout << "Printer idle\n";

  cout << "Setting p1 to printing...\n";
  p1.set_print(1);
  if(inuse(p1, p2)) cout << "Now, printer in use.\n";

  cout << "Turn off p1...\n";
  p1.set_print(0);
  if(!inuse(p1, p2)) cout << "Printer idle\n";

  cout << "Turn on p2...\n";
```

```
p2.set_print(1);
if(inuse(p1, p2)) cout << "Now, printer in use.\n";

return 0;
```

MASTERY SKILLS CHECK: Chapter

1. For one object to be assigned to another, both must be of the same class type.
2. The trouble with the assignment of **ob1** to **ob2** is that the memory pointed to by **ob2**'s initial value of **p** is now lost because this value is overwritten by the assignment. This memory thus becomes impossible to free, and the memory pointed to by **ob1**'s **p** is freed twice when it is destroyed—possibly causing damage to the dynamic allocation system.
3.
```
int light(planet p)
{
  return p.get_miles() / 186000;
}
```
4. Yes.
5.
```
// Load a stack with the alphabet.
#include <iostream>
using namespace std;

#define SIZE 27

// Declare a stack class for characters.
class stack {
  char stck[SIZE]; // holds the stack
  int tos; // index of top of stack
public:
  stack(); // constructor
  void push(char ch); // push character on stack
  char pop(); // pop character from stack
};

// Initialize the stack.
stack::stack()
```

```
    cout << "Constructing a stack\n";
    tos = 0;
  }

  // Push a character.
  void stack::push(char ch)
  {
    if(tos==SIZE) {
      cout << "Stack is full\n";
      return;
    }
    stck[tos] = ch;
    tos++;
  }

  // Pop a character.
  char stack::pop()
  {
    if(tos==0) {
      cout << "Stack is empty\n";
      return 0; // return null on empty stack
    }
    tos--;
    return stck[tos];
  }

  void showstack(stack o);
  stack loadstack();

  int main()
  {
    stack s1;

    s1 = loadstack();
    showstack(s1);

    return 0;
  }

  // Display the contents of a stack.
  void showstack(stack o)

  {
    char c;
```

```
  // when this statement ends, the o stack is empty
  while(c=o.pop()) cout << c << "\n";
  cout << "\n";
}

// Load a stack with the letters of the alphabet.
stack loadstack()
{
  stack t;
  char c;
  for(c = 'a'; c <= 'z'; c++) t.push(c);
  return t;
}
```

6. When passing an object to a function or returning an object from a function, temporary copies of the object are created that will be destroyed when the function terminates. When a temporary copy of an object is destroyed, the destructor function might destroy something that is needed elsewhere in the program.
7. A friend is a nonmember function that is granted access to the private members of the class for which it is a friend.

CUMULATIVE SKILLS CHECK: Chapter 3

1.
```
// Load a stack with the alphabet.
#include <iostream>
#include <cctype>
using namespace std;

#define SIZE 27

// Declare a stack class for characters.
class stack {
  char stck[SIZE]; // holds the stack
  int tos; // index of top of stack
public:
  stack(); // constructor
  void push(char ch); // push character on stack
  char pop(); // pop character from stack
};

// Initialize the stack.
```

```
stack::stack()
{
  cout << "Constructing a stack\n";
  tos = 0;
}

// Push a character.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full\n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Pop a character.
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty\n";
    return 0; // return null on empty stack
  }
  tos--;
  return stck[tos];
}

void showstack(stack o);
stack loadstack();
stack loadstack(int upper);

int main()
{
  stack s1, s2, s3;

  s1 = loadstack();
  showstack(s1);

  // get uppercase letters
  s2 = loadstack(1);
  showstack(s2);

  // use lowercase letters
  s3 = loadstack(0);
```

```
  showstack(s3);

  return 0;
}

// Display the contents of a stack.
void showstack(stack o)
{
  char c;

  // when this statement ends, the o stack is empty
  while(c=o.pop()) cout << c << "\n";
  cout << "\n";
}

// Load a stack with the letters of the alphabet.
stack loadstack()
{
  stack t;
  char c;

  for(c = 'a'; c<='z'; c++) t.push(c);
  return t;
}

/* Load a stack with the letters of the alphabet. Uppercase
   letters if upper is 1; lowercase otherwise. */
stack loadstack(int upper)
{
  stack t;
  char c;

  if(upper) c = 'A';
  else c = 'a';

  for(; toupper(c)<='Z'; c++) t.push(c);
  return t;
}
```

2.
```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
```

```
class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
  friend char *get_string(strtype *ob);
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = (char *) malloc(len+1);
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  cout << "Freeing p\n";
  free(p);
}

void strtype::show()
{
  cout << p << " - length: " << len;
  cout << "\n";
}

char *get_string(strtype *ob)
{
  return ob->p;
}

int main()
{
  strtype s1("This is a test.");

  char *s;
```

```
  s1.show();

  // get pointer to string
  s = get_string(&s1);
  cout << "Here is string contained in s1: ";
  cout << s << "\n";

  return 0;
}
```

3. The outcome of the experiment is as follows: Yes, data from the base class is also copied when an object of a derived class is assigned to another object of the same derived class. Here is a program that demonstrates this fact:

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

class base {
  int a;
public:
  void load_a(int n) { a = n; }
  int get_a() { return a; }
};

class derived : public base {
  int b;
public:
  void load_b(int n) { b = n; }
  int get_b() { return b; }
};

int main()
{
  derived ob1, ob2;

  ob1.load_a(5);
  ob1.load_b(10);

  // assign ob1 to ob2
  ob2 = ob1;

  cout << "Here is ob1's a and b: ";
  cout << ob1.get_a() << ' ' << ob1.get_b() << "\n";
```

```
    cout << "Here is ob2's a and b: ";
    cout << ob2.get_a() << ' ' << ob2.get_b() << "\n";

    /* As you can probably guess, the output is the same for
       each object. */

    return 0;
}
```

REVIEW SKILLS CHECK: Chapter 4

1. When one object is assigned to another of the same type, the current values of all data members of the object on the right are assigned to the corresponding data members on the left.
2. Trouble can occur when one object is assigned to another if that assignment overwrites important data already existing in the target object. For example, a pointer to dynamic memory or to an open file can be overwritten and, therefore, lost.
3. When an object is passed to a function, a copy is made. However, the copy's constructor function is not called. The copy's destructor is called when the object is destroyed by the termination of the function.
4. The violation of the separation between an argument and its copy when passed to a parameter can be caused by several situations. For example, if dynamic memory is freed by a destructor, that memory will also be lost to the argument. In general, if the destructor function destroys anything that the original argument requires, damage to the argument will occur.
5.

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

class summation {
  int num;
  long sum; // summation of num
public:
  void set_sum(int n);
  void show_sum() {
    cout << num << " summed is " << sum << "\n";
```

```
void summation::set_sum(int n)
{
  int i;

  num = n;

  sum = 0;
  for(i=1; i<=n; i++)
    sum += i;
}

summation make_sum()
{
  int i;
  summation temp;

  cout << "Enter number: ";
  cin >> i;

  temp.set_sum(i);

  return temp;
}

int main()
{
  summation s;

  s = make_sum();

  s.show_sum();

  return 0;
}
```

6. For some compilers, in-line functions cannot contain loops.

7.
```
#include <iostream>
using namespace std;

class myclass {
  int num;
public:
  myclass(int x) { num = x; }
  friend int isneg(myclass ob);
```

```
};

int isneg(myclass ob)
{
  return (ob.num < 0) ? 1 : 0;
}

int main()
{
  myclass a(-1), b(2);

  cout << isneg(a) << ' ' << isneg(b);
  cout << "\n";

  return 0;
}
```

8. Yes, a friend function can be friends with more than one class.

4.1 EXERCISES

1.
```
#include <iostream>
using namespace std;

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};

int main()
{
  letters ob[10] = { 'a', 'b', 'c','d', 'e', 'f',
                     'g', 'h', 'i', 'j' };

  int i;

  for(i=0; i<10; i++)
    cout << ob[i].get_ch() << ' ';

  cout << "\n";
```

```
  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class squares {
  int num, sqr;
public:
  squares(int a, int b) { num = a; sqr = b; }
  void show() {cout << num << ' ' << sqr << "\n"; }
};

int main()
{
  squares ob[10] = {
    squares(1, 1),
    squares(2, 4),
    squares(3, 9),
    squares(4, 16),
    squares(5, 25),
    squares(6, 36),
    squares(7, 49),
    squares(8, 64),
    squares(9, 81),
    squares(10, 100)
  };
  int i;

  for(i=0; i<10; i++) ob[i].show();

  return 0;
}
```

3.
```
#include <iostream>
using namespace std;

class letters {
  char ch;
public:
  letters(char c) { ch = c; }
  char get_ch() { return ch; }
};

int main()
```

```
  letters ob[10] = {
    letters('a'),
    letters('b'),
    letters('c'),
    letters('d'),
    letters('e'),
    letters('f'),
    letters('g'),
    letters('h'),
    letters('i'),
    letters('j')
  };

  int i;

  for(i=0; i<10; i++)
    cout << ob[i].get_ch() << ' ';

  cout << "\n";

  return 0;
}
```

4.2 EXERCISES

1.
```
// Display in reverse order.
#include <iostream>
using namespace std;

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; }
  int get_a() { return a; }
  int get_b() { return b; }
};

int main()
{
  samp ob[4] = {
    samp(1, 2),
    samp(3, 4),
    samp(5, 6),
```

```
    samp(7, 8)
  };
  int i;

  samp *p;

  p = &ob[3]; // get address of last element

  for(i=0; i<4; i++) {
    cout << p->get_a() << ' ';
    cout << p->get_b() << "\n";
    p--;  // advance to previous object
  }

  cout << "\n";

  return 0;
}
```

2.
```
/* Create a two-dimensional array of objects.
   Access via a pointer. */
#include <iostream>
using namespace std;

class samp {
  int a;
public:
  samp(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob[4][2] = {
    1, 2,
    3, 4,
    5, 6,
    7, 8
  };
  int i;

  samp *p;

  p = (samp *) ob;
```

```
  for(i=0; i<4; i++) {
    cout << p->get_a() << ' ';
    p++;
    cout << p->get_a() << "\n";
    p++;
  }

  cout << "\n";

  return 0;
}
```

4.3 EXERCISE

1.
```
// Use this pointer.
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  myclass(int n, int m) { this->a = n; this->b = m; }
  int add() { return this->a + this->b; }
  void show();
};

void myclass::show()
{
  int t;

  t = this->add(); // call member function
  cout << t << "\n";
}

int main()
{
  myclass ob(10, 14);

  ob.show();

  return 0;
}
```

4.4 EXERCISES

1.
```
#include <iostream>
using namespace std;

int main()
{
  float *f;
  long *l;
  char *c;

  f = new float;
  l = new long;
  c = new char;

  if(!f || !l || !c) {
    cout << "Allocation error.";
    return 1;
  }

  *f = 10.102;
  *l = 100000;
  *c = 'A';

  cout << *f << ' ' << *l << ' ' << *c;
  cout << '\n';

  delete f; delete l; delete c;

  return 0;
}
```

2.
```
#include <iostream>
#include <cstring>
using namespace std;

class phone {
  char name[40];
  char number[14];
public:
  void store(char *n, char *num);
  void show();
};
```

```
void phone::store(char *n, char *num)
{
  strcpy(name, n);
  strcpy(number, num);
}

void phone::show()
{
  cout << name << ": " << number;
  cout << "\n";
}

int main()
{
  phone *p;

  p = new phone;

  if(!p) {
    cout << "Allocation error.";
    return 1;
  }

  p->store("Isaac Newton", "111 555-2323");

  p->show();

  delete p;

  return 0;
}
```

3. On failure, **new** will either return a null pointer or generate an exception. You must check your compiler's documentation to determine which approach is used. In Standard C++, **new** generates an exception by default.

4.5 EXERCISES

1.
```
char *p;

p = new char [100];
```

```
// ...
strcpy(p, "This is a test");
```

2.
```
#include <iostream>
using namespace std;

int main()
{
  double *p;

  p = new double (-123.0987);

  cout << *p << '\n';

  return 0;
}
```

4.6 EXERCISES

1.
```
#include <iostream>
using namespace std;

void rneg(int &i); // reference version
void pneg(int *i); // pointer version

int main()
{
  int i = 10;
  int j = 20;

  rneg(i);
  pneg(&j);

  cout << i << ' ' << j << '\n';

  return 0;
}

// using a reference parameter
void rneg(int &i)
{
  i = -i;
```

```
// using a pointer parameter
void pneg(int *i)
{
  *i = - *i;
}
```

2. When **triple( )** is called, the address of **d** is explicitly obtained with the **&** operator. This is neither necessary nor legal. When a reference parameter is used, the argument is not preceded by the **&**.
3. The address of a reference parameter is automatically passed to the function. You need not obtain the address manually. Passing by reference is often faster than passing by value. No copy of the argument is generated. Therefore, there is no chance of a side effect occurring because the copy's destructor is called.

4.7 EXERCISE

1. In the original program, the object is passed to **show( )** by value. Thus, a copy is made. When **show( )** returns, the copy is destroyed and its destructor is called. This causes **p** to be released, but the memory pointed to it is still needed by the arguments to **show( )**. Here is a corrected version that uses a reference parameter to prevent a copy from being made when the function is called:

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

class strtype {
  char *p;
public:
  strtype(char *s);
  ~strtype() { delete [] p; }
  char *get() { return p; }
};

strtype::strtype(char *s)
{
```

```
  int l;

  l = strlen(s)+1;

  p = new char [l];
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }

  strcpy(p, s);
}

// Fix by using a reference parameter.
void show(strtype &x)
{
  char *s;

  s = x.get();
  cout << s << "\n";
}

int main()
{
  strtype a("Hello"), b("There");

  show(a);
  show(b);

  return 0;
}
```

4.8 EXERCISES

1.
```
// A simple bounded two-dimensional array example.
#include <iostream>
#include <cstdlib>
using namespace std;

class array {
  int isize, jsize;
  int *p;
public:
  array(int i, int j);
```

```
    int &put(int i, int j);
    int get(int i, int j);
};

array::array(int i, int j)
{
  p = new int [i*j];
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }
  isize = i;
  jsize = j;
}

// Put something into the array.
int &array::put(int i, int j)
{
  if(i<0 || i>=isize || j<0 || j>=jsize) {
    cout << "Bounds error!!!\n";
    exit(1);
  }
  return p[i*jsize + j];  // return reference to p[i]
}

// Get something from the array.
int array::get(int i, int j)
{
  if(i<0 || i>=isize || j<0 || j>=jsize) {
    cout << "Bounds error!!!\n";
    exit(1);
  }
  return p[i*jsize +j]; // return character
}

int main()
{
  array a(2, 3);
  int i, j;

  for(i=0; i<2; i++)
    for(j=0; j<3; j++)
      a.put(i, j) = i+j;

  for(i=0; i<2; i++)
```

```
    for(j=0; j<3; j++)
      cout << a.get(i, j) << ' ';

  // generate out of bounds
  a.put(10, 10);

  return 0;
}
```

2. No. A reference returned by a function cannot be assigned to a pointer.

MASTERY SKILLS CHECK: CHAPTER 4

1.
```
#include <iostream>
using namespace std;

class a_type {
  double a, b;
public:
  a_type(double x, double y) {
    a = x;
    b = y;
  }
  void show() { cout << a << ' ' << b << '\n'; }
};

int main()
{
  a_type ob[2][5] = {
    a_type(1, 1), a_type(2, 2),
    a_type(3, 3), a_type(4, 4),
    a_type(5, 5), a_type(6, 6),
    a_type(7, 7), a_type(8, 8),
    a_type(9, 9), a_type(10, 10)
  };

  int i, j;

  for(i=0; i<2; i++)
    for(j=0; j<5; j++)
      ob[i][j].show();
```

```
  cout << '\n';

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class a_type {
  double a, b;
public:
  a_type(double x, double y) {
    a = x;
    b = y;
  }
  void show() { cout << a << ' ' << b << '\n'; }
};

int main()
{
  a_type ob[2][5] = {
    a_type(1, 1), a_type(2, 2),
    a_type(3, 3), a_type(4, 4),
    a_type(5, 5), a_type(6, 6),
    a_type(7, 7), a_type(8, 8),
    a_type(9, 9), a_type(10, 10)
  };

  a_type *p;

  p = (a_type *) ob;

  int i, j;

  for(i=0; i<2; i++)
    for(j=0; j<5; j++) {
      p->show();
      p++;
    }

  cout << '\n';

  return 0;
}
```

3. The **this** pointer is a pointer that is automatically passed to a member function and that points to the object that generated the call.

4. The general forms of **new** and **delete** are

   p-var = new *type*;
   delete *p-var*;

   When using **new**, you don't need to use a type cast. The size of the object is automatically determined; you don't need to use **sizeof**. Also, you don't need to include **<cstdlib>** in your program.

5. A reference is essentially an implicit pointer constant that is effectively a different name for another variable or argument. One advantage of using a reference parameter is that no copy of the argument is made.

6.
```
#include <iostream>
using namespace std;

void recip(double &d);

int main()
{
  double x = 100.0;

  cout << "x is " << x << '\n';

  recip(x);

  cout << "Reciprocal is " << x << '\n';

  return 0;
}

void recip(double &d)
{
  d = 1/d;
}
```

CUMULATIVE SKILLS CHECK: Chapter 4

1. To access a member of an object by using a pointer, use the arrow (->) operator.

2.
```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int.len;
public:
  strtype(char *ptr);
  ~strtype();
  void show();
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr);
  p = new char [len+1];
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(p, ptr);
}

strtype::~strtype()
{
  cout << "Freeing p\n";
  delete [] p;
}

void strtype::show()
{
  cout << p << " - length: " << len;
  cout << "\n";
}

int main()
{
```

```
  strtype s1("This is a test."), s2("I like C++.");

  s1.show();
  s2.show();

  return 0;
}
```

REVIEW SKILLS CHECK: Chapter 5

1. A reference is a special type of pointer that is automatically dereferenced and that can be used interchangeably with the object it is pointing to. There are three types of references: parameter references, independent references, and references that are returned by functions.

2.
```
#include <iostream>
using namespace std;

int main()
{
  float *f;
  int *i;

  f = new float;
  i = new int;

  if(!f || !i) {
    cout << "Allocation error\n";
    return 1;
  }

  *f = 10.101;
  *i = 100;

  cout << *f << ' ' << *i << '\n';

  delete f;
  delete i;

  return 0;
}
```

3. The general form of **new** that includes an initializer is shown here:

   p-var = new *type* (*initializer*);

   For example, this allocates an integer and gives it the value 10:

```
int *p;

p = new int (10);
```

4.
```
#include <iostream>
using namespace std;

class samp {
  int x;
public:
  samp(int n) { x = n; }
  int getx() { return x; }
};

int main()
{
  samp A[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int i;

  for(i=0; i<10; i++) cout << A[i].getx() << ' ';

  cout << "\n";

  return 0;
}
```

5. Advantages: A reference parameter does not cause a copy of the object used in the call to be made. A reference is often faster to pass than a value. The reference parameter streamlines the call-by-reference syntax and procedure, reducing the chance for errors.

   Disadvantages: Changes to a reference parameter alter the variable used in the call. A reference parameter opens the possibility of side effects in the calling routine.

6. No.

7.
```
#include <iostream>
using namespace std;

void mag(long &num, long order);

int main()
{
  long n = 4;
  long o = 2;

  cout << "4 raised to the 2nd order of magnitude is ";
  mag(n, o);
  cout << n << '\n';

  return 0;
}

void mag(long &num, long order)
{
  for( ; order; order--) num = num * 10;
}
```

5.1 EXERCISES

1.
```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype();
  strtype(char *s, int l);
  char *getstring() { return p; }
  int getlength() { return len; }
};

strtype::strtype()
{
  p = new char [255];
  if(!p) {
    cout << "Allocation error\n";
```

```
    exit(1);
  }
  *p = '\0'; // null string
  len = 255;
}

strtype::strtype(char *s, int l)
{
  if(strlen(s) >= l) {
    cout << "Allocating too little memory!\n";
    exit(1);
  }

  p = new char [l];
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(p, s);
  len = l;
}

int main()
{
  strtype s1;
  strtype s2("This is a test", 100);

  cout << "s1: " << s1.getstring() << " - Length: ";
  cout << s1.getlength() << '\n';

  cout << "s2: " << s2.getstring() << " - Length: ";
  cout << s2.getlength() << '\n';

  return 0;
}
```

2.
```
// Stopwatch emulator
#include <iostream>
#include <ctime>
using namespace std;

class stopwatch {
  double begin, end;
public:
  stopwatch();
```

```
    stopwatch(clock_t t);
    ~stopwatch();
    void start();
    void stop();
    void show();
};

stopwatch::stopwatch()
{
    begin = end = 0.0;
}

stopwatch::stopwatch(clock_t t)
{
    begin = (double) t / CLOCKS_PER_SEC;
    end = 0.0;
}

stopwatch::~stopwatch()
{
    cout << "Stopwatch object being destroyed..."
    show();
}

void stopwatch::start()
{
    begin = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::stop()
{
    end = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::show()
{
    cout << "Elapsed time: " << end - begin;
    cout << "\n";
}

int main()
{
    stopwatch watch;
    long i;
```

```
  watch.start();
  for(i=0; i<320000; i++) ; // time a for loop
  watch.stop();
  watch.show();

  // create object using initial value
  stopwatch s2(clock());
  for(i=0; i<250000; i++) ; // time a for loop
  s2.stop();
  s2.show();

  return 0;
}
```

5.2 EXERCISES

1. The **obj** and **temp** objects are constructed normally. However, when **temp** is returned by **f( )**, a temporary object is made, and it is this temporary object that generates the call to the copy constructor.

2. As the program is written, when an object is passed to **getval( )**, a bitwise copy is made. When **getval( )** returns and that copy is destroyed, the memory allocated to that object (which is pointed to by **p**) is released. However, this is the same memory still required by the object used in the call to **getval( )**. The correct version of the program is shown here. It uses a copy constructor to avoid this problem.

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

class myclass {
  int *p;
public:
  myclass(int i);
  myclass(const myclass &o); // copy constructor
  ~myclass() { delete p; }
  friend int getval(myclass o);
};
```

```
myclass::myclass(int i)
{
  p = new int;

  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }
  *p = i;
}

// Copy constructor
myclass::myclass(const myclass &o)
{
  p = new int; // allocate copy's own memory

  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }
  *p = *o.p;
}

int getval(myclass o)
{
  return *o.p; // get value
}

int main()
{
  myclass a(1), b(2);

  cout << getval(a) << " " << getval(b);
  cout << "\n";
  cout << getval(a) << " " << getval(b);

  return 0;
}
```

3. A copy constructor is invoked when one object is used to initialize another. A normal constructor is called when an object is created.

5.4 EXERCISES

1.
```
#include <iostream>
#include <cstdlib>
using namespace std;

long mystrtol(const char *s, char **end, int base = 10)
{
  return strtol(s, end, base);
}

int main()
{
  long x;
  char *s1 = "100234";
  char *p;

  x = mystrtol(s1, &p, 16);
  cout << "Base 16: " << x << '\n';

  x = mystrtol(s1, &p, 10);
  cout << "Base 10: " << x << '\n';

  x = mystrtol(s1, &p); // use default base of 10
  cout << "Base 10 by default: " << x << '\n';

  return 0;
}
```

2. All parameters taking default arguments must appear to the right of those that do not. That is, once you begin giving parameters defaults, all subsequent parameters must also have defaults. In the question, **q** is not given a default.

3. Since cursor positioning functions differ from compiler to compiler and environment to environment, only one possible solution is shown. The following program works for Borland C++ under a command-prompt environment.

```
// Note: This program is Borland C++-specific.
#include <iostream>
#include <conio.h>
using namespace std;
```

```
void myclreol(int len = -1);

int main()
{
  int i;

  gotoxy(1, 1);
  for(i=0; i<24; i++)
    cout << "abcdefghijklmnopqrstuvwxyz1234567890\n";

  gotoxy(1, 2);
  myclreol();
  gotoxy(1, 4);
  myclreol(20);

  return 0;
}
// Clear to end of line unless len parameter is specified.
void myclreol(int len)
{
  int x, y;

  x = wherex();  // get x position
  y = wherey();  // get y position

  if(len == -1) len = 80-x;

  int i = x;

  for( ; i<=len; i++) cout << ' ';

  gotoxy(x, y); // reset the cursor
}
```

4. A default argument cannot be another parameter or a local variable.

5.6 EXERCISE

1.
```
#include <iostream>
using namespace std;

int dif(int a, int b)
```

```
{
  return a-b;
}

float dif(float a, float b)
{
  return a-b;
}

int main()
{
  int (*p1)(int, int);
  float (*p2)(float, float);

  p1 = dif; // address of dif(int, int)
  p2 = dif; // address of dif(float, float);

  cout << p1(10, 5) << ' ';
  cout << p2(10.5, 8.9) << '\n';

  return 0;
}
```

MASTERY SKILLS CHECK: Chapter 5

1.
```
// Overload date() for time_t.
#include <iostream>
#include <cstdio> // included for sscanf()
#include <ctime>
using namespace std;

class date {
  int day, month, year;
public:
  date(char *str);
  date(int m, int d, int y) {
    day = d;
    month = m;
    year = y;
  }
  // overload for parameter of type time_t
  date(time_t t);
  void show() {
    cout << month << '/' << day << '/';
```

```
    cout << year << '\n';
  }
};

date::date(char *str)
{
  sscanf(str, "%d%*c%d%*c%d", &month, &day, &year);
}

date::date(time_t t)
{
  struct tm *p;

  p = localtime(&t); // convert to broken down time
  day = p->tm_mday;
  month = p->tm_mon;
  year = p->tm_year;
}

int main()
{
  // construct date object using string
  date sdate("12/31/99");

  // construct date object using integers
  date idate(12, 31, 99);

  /* construct date object using time_t - this
     creates an object using the system date */
  date tdate(time(NULL));

  sdate.show();
  idate.show();
  tdate.show();

  return 0;
}
```

2. The class **samp** defines only one constructor, and this constructor requires an initializer. Therefore, it is improper to declare an object of type **samp** without one. (That is, **samp x** is an invalid declaration.)

3. One reason to overload a constructor is to provide flexibility, allowing you to choose the most appropriate constructor in the specific instance. Another is to allow both initialized and uninitialized objects to be declared. You might want to overload a constructor so that dynamic arrays can be allocated.
4. The most common general form of a copy constructor is shown here:

```
classname (const classname &obj) {
  // body of constructor
}
```

5. A copy constructor is called when an initialization takes place—specifically, when one object is explicitly used to initialize another, when an object is passed as a parameter to a function, and when a temporary object is created when an object is returned by a function.
6. The **overload** keyword is obsolete. In early versions of C++ it was used to tell the compiler that a function will be overloaded. It is not supported by modern compilers.
7. A default argument is a value that is given to a function parameter when no corresponding argument appears when the function is called.
8.

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

void reverse(char *str, int count = 0);

int main()
{
  char *s1 = "This is a test.";
  char *s2 = "I like C++.";

  reverse(s1); // reverse entire string
  reverse(s2, 7);  // reverse first 7 chars

  cout << s1 << '\n';
  cout << s2 << '\n';

  return 0;
```

```
void reverse(char *str, int count)
{
  int i, j;
  char temp;

  if(!count) count = strlen(str)-1;

  for(i=0, j=count; i<j; i++, j--) {
    temp = str[i];
    str[i] = str[j];
    str[j] = temp;
  }
}
```

9. All parameters receiving default arguments must appear to the right of those that do not.
10. Ambiguity can be introduced by default type conversions, reference parameters, and default arguments.
11. It is ambiguous because the compiler cannot know which version of **compute( )** to call. Is it the first version, with **divisor** defaulting? Or is it the second version, which takes only one parameter?
12. When you are obtaining the address of an overloaded function, it is the type specification of the pointer that determines which function is used.

CUMULATIVE SKILLS CHECK: Chapter 5

1.
```
#include <iostream>
using namespace std;

void order(int &a, int &b)
{
  int t;

  if(a<b) return;
  else { // swap a and b
    t = a;
    a = b;
    b = t;
```

```
  }
}

int main()
{
  int x=10, y=5;

  cout << "x: " << x << ", y: " << y << "\n";

  order(x, y);
  cout << "x: " << x << ", y: " << y << "\n";

  return 0;
}
```

2. The syntax for calling a function that takes a reference parameter is identical to the syntax for calling a function that takes a value parameter.
3. A default argument is essentially a shorthand approach to function overloading because the net result is the same. For example,

```
int f(int a, int b = 0);
```

is functionally equivalent to these two overloaded functions:

```
int f(int a);

int f(int a, int b);
```

4.
```
#include <iostream>
using namespace std;

class samp {
  int a;
public:
  samp() { a = 0; }
  samp(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  samp ob(88);
```

```
  samp obarray[10];

  // ...
}
```

5. Copy constructors are needed when you, the programmer, must control precisely how a copy of an object is made. This is important only when the default bitwise copy creates undesired side effects.

REVIEW SKILLS CHECK: Chapter 6

1.
```
class myclass {
  int x, y;
public:
  myclass(int i, int j) { x=i; y=j; }
  myclass() { x=0; y=0; }
};
```

2.
```
class myclass {
  int x, y;
public:
  myclass(int i=0, int j=0) { x=i; y=j; }
};
```

3. Once default arguments have begun, a nondefaulting parameter cannot occur.
4. A function cannot be overloaded when the only difference is that one version takes a value parameter and the other takes a reference parameter. (The compiler cannot tell them apart.)
5. It is appropriate to use default arguments when there are one or more values that will occur frequently. It is inappropriate when there is no value or values that have a greater likelihood of occurring.
6. No, because there is no way to initialize a dynamic array. This class has only one constructor, and it requires initializers.
7. A copy constructor is a special constructor that is called when one object initializes another. This circumstance can occur in any of the following three ways: When one object is explicitly used to initialize another, when an object is passed to a function, or when a temporary object is created as a function return value.

6.2 EXERCISES

1.
```
// Overload the * and / relative to coord class.
#include <iostream>
using namespace std;

class coord {
  int x, y; // coordinate values
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator*(coord ob2);
  coord operator/(coord ob2);
};

// Overload * relative to coord class.
coord coord::operator*(coord ob2)
{
  coord temp;

  temp.x = x * ob2.x;
  temp.y = y * ob2.y;

  return temp;
}

// Overload / relative to coord class.
coord coord::operator/(coord ob2)
{
  coord temp;

  temp.x = x / ob2.x;
  temp.y = y / ob2.y;

  return temp;
}

int main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 * o2;
```

```
  o3.get_xy(x, y);
  cout << "(o1*o2) X: " << x << ", Y: " << y << "\n";

  o3 = o1 / o2;
  o3.get_xy(x, y);
  cout << "(o1/o2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

2. The overloading of the **%** operator is inappropriate because its operation is unrelated to the traditional use.

6.3 EXERCISE

1.
```
// Overload the < and > relative to coord class.
#include <iostream>
using namespace std;

class coord {
  int x, y; // coordinate values
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  int operator<(coord ob2);
  int operator>(coord ob2);
};

// Overload the < operator for coord.
int coord::operator<(coord ob2)
{
  return x<ob2.x && y<ob2.y;
}

// Overload the > operator for coord.
int coord::operator>(coord ob2)
{
  return x>ob2.x && y>ob2.y;
}

int main()
{
```

```
  coord o1(10, 10), o2(5, 3);

  if(o1>o2) cout << "o1 > o2\n";
  else cout << "o1 <= o2 \n";

  if(o1<o2) cout << "o1 < o2\n";
  else cout << "o1 >= o2\n";

  return 0;
}
```

EXERCISES

1.
```
// Overload the -- relative to coord class.
#include <iostream>
using namespace std;

class coord {
  int x, y; // coordinate values
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator--(); // prefix
  coord operator--(int notused); // postfix
};

// Overload prefix -- for coord class.
coord coord::operator--()
{
  x--;
  y--;
  return *this;
}

// Overload postfix -- for coord class.
coord coord::operator--(int notused)
{
  x--;
  y--;
  return *this;
}

int main()
```

```
{
  coord o1(10, 10);
  int x, y;

  o1--; // decrement an object
  o1.get_xy(x, y);
  cout << "(o1--) X: " << x << ", Y: " << y << "\n";

  --o1; // decrement an object
  o1.get_xy(x, y);
  cout << "(--o1) X: " << x << ", Y: " << y << "\n";
  return 0;
}
```

```
// Overload the + relative to coord class.
#include <iostream>
using namespace std;

class coord {
  int x, y; // coordinate values
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator+(coord ob2); // binary plus
  coord operator+(); // unary plus
};

// Overload + relative to coord class.
coord coord::operator+(coord ob2)
{
  coord temp;

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;

  return temp;
}

// Overload unary + for coord class.
coord coord::operator+()
{
  if(x<0) x = -x;
  if(y<0) y = -y;
```

```
  return *this;
}

int main()
{
  coord o1(10, 10), o2(-2, -2);
  int x, y;
  o1 = o1 + o2; // addition
  o1.get_xy(x, y);
  cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";

  o2 = +o2; // absolute value
  o2.get_xy(x, y);
  cout << "(+o2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

6.5 EXERCISES

1.
```
/* Overload the - and / relative to coord class
   using friend functions. */
#include <iostream>
using namespace std;

class coord {
  int x, y; // coordinate values
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator-(coord ob1, coord ob2);
  friend coord operator/(coord ob1, coord ob2);
};

// Overload - relative to coord class using friend.
coord operator-(coord ob1, coord ob2)
{
  coord temp;

  temp.x = ob1.x - ob2.x;
  temp.y = ob1.y - ob2.y;

  return temp;
```

```
// Overload / relative to coord class using friend.
coord operator/(coord ob1, coord ob2)
{
  coord temp;

  temp.x = ob1.x / ob2.x;
  temp.y = ob1.y / ob2.y;

  return temp;
}

int main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 - o2;
  o3.get_xy(x, y);
  cout << "(o1 - o2) X: " << x << ", Y: " << y << "\n";

  o3 = o1 / o2;
  o3.get_xy(x, y);
  cout << "(o1/o2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

2.
```
// Overload the * for ob*int and int*ob.
#include <iostream>
using namespace std;

class coord {
  int x, y; // coordinate values
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator*(coord ob1, int i);
  friend coord operator*(int i, coord ob2);
};

// Overload * one way.
coord operator*(coord ob1, int i)
```

```
  coord temp;

  temp.x = ob1.x * i;
  temp.y = ob1.y * i;

  return temp;
}

// Overload * another way.
coord operator*(int i, coord ob2)
{
  coord temp;

  temp.x = ob2.x * i;
  temp.y = ob2.y * i;

  return temp;
}

int main()
{
  coord o1(10, 10), o2;
  int x, y;

  o2 = o1 * 2;  // ob * int
  o2.get_xy(x, y);
  cout << "(o1*2) X: " << x << ", Y: " << y << "\n";

  o2 = 3 * o1; // int * ob
  o2.get_xy(x, y);
  cout << "(3*o1) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

3. By using friend functions, you make it possible to have a built-in type as the left operand. When member functions are used, the left operand must be an object of the class for which the operator is defined.

4.
```
// Overload the -- relative to coord class using a friend.
#include <iostream>
using namespace std;
```

```
class coord {
  int x, y; // coordinate values
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  friend coord operator--(coord &ob); // prefix
  friend coord operator--(coord &ob, int notused); // postfix
};

// Overload -- (prefix) for coord class using a friend.
coord operator--(coord &ob)
{
  ob.x--;
  ob.y--;
  return ob;
}

// Overload -- (postfix) for coord class using a friend.
coord operator--(coord &ob, int notused)
{
  ob.x--;
  ob.y--;
  return ob;
}

int main()
{
  coord o1(10, 10);
  int x, y;

  --o1; // decrement an object
  o1.get_xy(x, y);
  cout << "(--o1) X: " << x << ", Y: " << y << "\n";

  o1--; // decrement an object
  o1.get_xy(x, y);
  cout << "(o1--) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

6.6 EXERCISE

1.
```
#include <iostream>
#include <cstdlib>
using namespace std;

class dynarray {
  int *p;
  int size;
public:
  dynarray(int s);
  int &put(int i);
  int get(int i);
  dynarray &operator=(dynarray &ob);
};

// Constructor
dynarray::dynarray(int s)
{
  p = new int [s];
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }

  size = s;
}

// Store an element.
int &dynarray::put(int i)
{
  if(i<0 || i>=size) {
    cout << "Bounds error!\n";
    exit(1);
  }

  return p[i];
}

// Get an element.
int dynarray::get(int i)
{
```

```
  if(i<0 || i>=size) {
    cout << "Bounds error!\n";
    exit(1);
  }

  return p[i];
}

// Overload = for dynarray.
dynarray &dynarray::operator=(dynarray &ob)
{
  int i;

  if(size!=ob.size) {
    cout << "Cannot copy arrays of differing sizes!\n";
    exit(1);
  }

  for(i = 0; i<size; i++) p[i] = ob.p[i];
  return *this;
}

int main()
{
  int i;

  dynarray ob1(10), ob2(10), ob3(100);

  ob1.put(3) = 10;
  i = ob1.get(3);
  cout << i << "\n";

  ob2 = ob1;

  i = ob2.get(3);
  cout << i << "\n";

  // generates an error
  ob1 = ob3; //  !!!
  return 0;
```

6.7 EXERCISES

1.
```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *s);
  ~strtype() {
     cout << "Freeing " << (unsigned) p << '\n';
     delete [] p;
   }
  char *get() { return p; }
  strtype &operator=(strtype &ob);
  char &operator[](int i);
};

strtype::strtype(char *s)
{
  int l;

  l = strlen(s)+1;

  p = new char [l];
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }

  len = l;
  strcpy(p, s);
}

// Assign an object.
strtype &strtype::operator=(strtype &ob)
{
  // see if more memory is needed
  if(len < ob.len) { // need to allocate more memory
    delete [] p;
```

```
    p = new char [ob.len];
    if(!p) {
      cout << "Allocation error\n";
      exit(1);
    }
  }
  len = ob.len;
  strcpy(p, ob.p);
  return *this;
}

// Index characters in string.
char &strtype::operator[](int i)
{
  if(i<0 || i>len-1) {
    cout << "\nIndex value of ";
    cout << i << " is out-of-bounds.\n";
    exit(1);
  }

  return p[i];
}

int main()
{
  strtype a("Hello"), b("There");

  cout << a.get() << '\n';
  cout << b.get() << '\n';

  a = b; // now p is not overwritten

  cout << a.get() << '\n';
  cout << b.get() << '\n';

  // access characters using array indexing
  cout << a[0] << a[1] << a[2] << "\n";

  // assign characters using array indexing
  a[0] = 'X';
  a[1] = 'Y';
  a[2] = 'Z';

  cout << a.get() << "\n";
```

```
  return 0;
}
```

2.
```
#include <iostream>
#include <cstdlib>
using namespace std;

class dynarray {
  int *p;
  int size;
public:
  dynarray(int s);
  dynarray &operator=(dynarray &ob);
  int &operator[](int i);
};

// Constructor
dynarray::dynarray(int s)
{
  p = new int [s];
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }

  size = s;
}

// Overload = for dynarray.
dynarray &dynarray::operator=(dynarray &ob)
{
  int i;

  if(size!=ob.size) {
    cout << "Cannot copy arrays of differing sizes!\n
    exit(1);
  }

  for(i = 0; i<size; i++) p[i] = ob.p[i];
  return *this;
}

// Overload []
int &dynarray::operator[](int i)
{
```

```
  if(i<0 || i>size) {
    cout << "\nIndex value of ";
    cout << i << " is out-of-bounds.\n";
    exit(1);
  }
  return p[i];
}

int main()
{
  int i;

  dynarray ob1(10), ob2(10), ob3(100);

  ob1[3] = 10;
  i = ob1[3];
  cout << i << "\n";

  ob2 = ob1;

  i = ob2[3];
  cout << i << "\n";

  // generates an error
  ob1 = ob3; //  arrays differ sizes
  return 0;
}
```

MASTERY SKILLS CHECK: Chapter 6

1.
```
// Overload << and >>.
#include <iostream>
using namespace std;

class coord {
  int x, y; // coordinate values
public:
  coord() { x=0; y=0; }
  coord(int i, int j) { x=i; y=j; }
  void get_xy(int &i, int &j) { i=x; j=y; }
  coord operator<<(int i);
  coord operator>>(int i);
};
```

```
// Overload <<.
coord coord::operator<<(int i)
{
  coord temp;

  temp.x = x << i;
  temp.y = y << i;

  return temp;
}

// Overload >>.
coord coord::operator>>(int i)
{
  coord temp;

  temp.x = x >> i;
  temp.y = y >> i;

  return temp;
}

int main()
{
  coord o1(4, 4), o2;
  int x, y;

  o2 = o1 << 2;  // ob << int
  o2.get_xy(x, y);
  cout << "(o1<<2) X: " << x << ", Y: " << y << "\n";

  o2 = o1 >> 2; // ob >> int
  o2.get_xy(x, y);
  cout << "(o1>>2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
```

```
  {
    x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
     i = x; j = y; k = z;
  }
  three_d operator+(three_d ob2);
  three_d operator-(three_d ob2);
  three_d operator++();
  three_d operator--();
};

three_d three_d::operator+(three_d ob2)
{
  three_d temp;
  temp.x = x + ob2.x;
  temp.y = y + ob2.y;
  temp.z = z + ob2.z;

  return temp;
}

three_d three_d::operator-(three_d ob2)
{
  three_d temp;

  temp.x = x - ob2.x;
  temp.y = y - ob2.y;
  temp.z = z - ob2.z;

  return temp;
}

three_d three_d::operator++()
{
  x++;
  y++;
  z++;

  return *this;
}

three_d three_d::operator--()
```

```
    x--;
    y--;
    z--;

    return *this;
  }

  int main()
  {
    three_d o1(10, 10, 10), o2(2, 3, 4), o3;
    int x, y, z;

    o3 = o1 + o2;
    o3.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    o3 = o1 - o2;
    o3.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    ++o1;
    o1.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    --o1;
    o1.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    return 0;
  }
```

3.
```
  #include <iostream>
  using namespace std;

  class three_d {
    int x, y, z;
  public:
    three_d(int i, int j, int k)
    {
      x = i; y = j; z = k;
```

```
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
     i = x; j = y; k = z;
  }
  three_d operator+(three_d &ob2);
  three_d operator-(three_d &ob2);
  friend three_d operator++(three_d &ob);
  friend three_d operator--(three_d &ob);
};

three_d three_d::operator+(three_d &ob2)
{
  three_d temp;

  temp.x = x + ob2.x;
  temp.y = y + ob2.y;
  temp.z = z + ob2.z;

  return temp;
}

three_d three_d::operator-(three_d &ob2)
{
  three_d temp;

  temp.x = x - ob2.x;
  temp.y = y - ob2.y;
  temp.z = z - ob2.z;

  return temp;
}

three_d operator++(three_d &ob)
{
  ob.x++;
  ob.y++;
  ob.z++;

  return ob;
}

three_d operator--(three_d &ob)
{
```

```
  ob.x--;
  ob.y--;
  ob.z--;

  return ob;
}

int main()
{
  three_d o1(10, 10, 10), o2(2, 3, 4), o3;
  int x, y, z;

  o3 = o1 + o2;
  o3.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  o3 = o1 - o2;
  o3.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  ++o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  --o1;
  o1.get(x, y, z);
  cout << "X: " << x << ", Y: " << y;
  cout << ", Z: " << z << "\n";

  return 0;
}
```

4. A binary member operator function is passed the left operand implicitly via the **this** pointer. A binary friend operator function is passed both operands explicitly. Unary member operator functions have no explicit parameters. A friend unary operator function has one parameter.

5. You will need to overload the = operator when the default bitwise copy is insufficient. For example, you might have a situation in which you want only parts of the data in one object to be assigned to another object.

6. No.

7.
```
#include <iostream>
using namespace std;

class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
    x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
     i = x; j = y; k = z;
  }
  friend three_d operator+(three_d ob, int i);
  friend three_d operator+(int i, three_d ob);
 };

three_d operator+(three_d ob, int i)
{
  three_d temp;
  temp.x = ob.x + i;
  temp.y = ob.y + i;
  temp.z = ob.z + i;

  return temp;
}

three_d operator+(int i, three_d ob)
{
  three_d temp;

  temp.x = ob.x + i;
  temp.y = ob.y + i;
  temp.z = ob.z + i;

  return temp;
}

int main()
{
  three_d o1(10, 10, 10);
```

```
    int x, y, z;

    o1 = o1 + 10;
    o1.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    o1 = -20 + o1;
    o1.get(x, y, z);
    cout << "X: " << x << ", Y: " << y;
    cout << ", Z: " << z << "\n";

    return 0;
}
```

8.
```
#include <iostream>
using namespace std;

class three_d {
  int x, y, z;
public:
  three_d(int i, int j, int k)
  {
    x = i; y = j; z = k;
  }
  three_d() { x=0; y=0; z=0; }
  void get(int &i, int &j, int &k)
  {
     i = x; j = y; k = z;
  }
  int operator==(three_d ob2);
  int operator!=(three_d ob2);
  int operator||(three_d ob2);
};

int three_d::operator==(three_d ob2)
{
  return x==ob2.x && y==ob2.y && z==ob2.z;
}

int three_d::operator!=(three_d ob2)
{
  return x!=ob2.x && y!=ob2.y && z!=ob2.z;
}
```

```
int three_d::operator||(three_d ob2)
{
  return x||ob2.x && y||ob2.y && z||ob2.z;
}

int main()
{
  three_d o1(10, 10, 10), o2(2, 3, 4), o3(0, 0, 0);

  if(o1==o1) cout << "o1 == o1\n";

  if(o1!=o2) cout << "o1 != o2\n";

  if(o3 || o1) cout << "o1 or o3 is true\n";

  return 0;
}
```

9. The [ ] is usually overloaded to allow an array encapsulated within a class to be indexed with the normal array indexing syntax.

CUMULATIVE SKILLS CHECK: Chapter 6

1.
```
/*
   For clarity, no error checking has been used. However
   you should add some if using this code for a real
   application.
*/
#include <iostream>
#include <cstring>
using namespace std;

class strtype {
  char s[80];
public:
  strtype() { *s = '\0'; }
  strtype(char *p) { strcpy(s, p); }
  char *get() { return s; }
  strtype operator+(strtype s2);
  strtype operator=(strtype s2);
  int operator<(strtype s2);
  int operator>(strtype s2);
  int operator==(strtype s2);
```

```
};

strtype strtype::operator+(strtype s2)
{
  strtype temp;

  strcpy(temp.s, s);
  strcat(temp.s, s2.s);

  return temp;
}

strtype strtype::operator=(strtype s2)
{
  strcpy(s, s2.s);

  return *this;
}

int strtype::operator<(strtype s2)
{
  return strcmp(s, s2.s) < 0;
}

int strtype::operator>(strtype s2)
{
  return strcmp(s, s2.s) > 0;
}

int strtype::operator==(strtype s2)
{
  return strcmp(s, s2.s) == 0;
}

int main()
{
  strtype o1("Hello"), o2(" There"), o3;

  o3 = o1 + o2;
  cout << o3.get() << "\n";

  o3 = o1;
  if(o1==o3) cout << "o1 equals o3\n";
```

```
if(o1>o2) cout << "o1 > o2\n";

  if(o1<o2) cout << "o1 < o2\n";

  return 0;
}
```

REVIEW SKILLS CHECK: Chapter 7

1. No. Overloading an operator simply expands the data types upon which it can operate, but no preexisting operations are affected.
2. Yes. You cannot overload an operator relative to one of C++'s built-in types.
3. No, the precedence cannot be changed. No, the number of operands cannot be altered.
4.
```
#include <iostream>
using namespace std;

class array {
  int nums[10];
public:
  array();
  void set(int n[10]);
  void show();
  array operator+(array ob2);
  array operator-(array ob2);
  int operator==(array ob2);
};

array::array()
{
  int i;
  for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
  int i;
```

```
  for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
  int i;

  for(i=0; i<10; i++)
    cout << nums[i] << ' ';

  cout << "\n";
}

array array::operator+(array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = nums[i] + ob2.nums[i];

  return temp;
}

array array::operator-(array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = nums[i] - ob2.nums[i];

  return temp;
}

int array::operator==(array ob2)
{
  int i;

  for(i=0; i<10; i++)
    if(nums[i]!=ob2.nums[i]) return 0;
```

```
  return 1;
}

int main()
{
  array o1, o2, o3;

  int i[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  o1.set(i);
  o2.set(i);

  o3 = o1 + o2;
  o3.show();

  o3 = o1 - o3;
  o3.show();

  if(o1==o2) cout << "o1 equals o2\n";
  else cout << "o1 does not equal o2\n";

  if(o1==o3) cout << "o1 equals o3\n";
  else cout << "o1 does not equal o3\n";

  return 0;
}
```

5.
```
#include <iostream>
using namespace std;

class array {
  int nums[10];
public:
  array();
  void set(int n[10]);
  void show();
  friend array operator+(array ob1, array ob2);
  friend array operator-(array ob1, array ob2);
  friend int operator==(array ob1, array ob2);
};

array::array()
{
  int i;
  for(i=0; i<10; i++) nums[i] = 0;
```

```
}

void array::set(int *n)
{
  int i;

  for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
  int i;

  for(i=0; i<10; i++)
    cout << nums[i] << ' ';

  cout << "\n";
}

array operator+(array ob1, array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = ob1.nums[i] + ob2.nums[i];

  return temp;
}

array operator-(array ob1, array ob2)
{
  int i;
  array temp;

  for(i=0; i<10; i++)
    temp.nums[i] = ob1.nums[i] - ob2.nums[i];

  return temp;
}

int operator==(array ob1, array ob2)
```

```
{
  int i;

  for(i=0; i<10; i++)
    if(ob1.nums[i]!=ob2.nums[i]) return 0;

  return 1;
}

int main()
{
  array o1, o2, o3;

  int i[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  o1.set(i);
  o2.set(i);

  o3 = o1 + o2;
  o3.show();

  o3 = o1 - o3;
  o3.show();

  if(o1==o2) cout << "o1 equals o2\n";
  else cout << "o1 does not equal o2\n";

  if(o1==o3) cout << "o1 equals o3\n";
  else cout << "o1 does not equal o3\n";

  return 0;
}
```

6.
```
#include <iostream>
using namespace std;

class array {
  int nums[10];
public:
  array();
  void set(int n[10]);
  void show();
  array operator++();
```

```
  friend array operator--(array &ob);
};

array::array()
{
  int i;

  for(i=0; i<10; i++) nums[i] = 0;
}

void array::set(int *n)
{
  int i;

  for(i=0; i<10; i++) nums[i] = n[i];
}

void array::show()
{
  int i;

  for(i=0; i<10; i++)
    cout << nums[i] << ' ';

  cout << "\n";
}

// Overload unary op using member function.
array array::operator++()
{
  int i;

  for(i=0; i<10; i++)
    nums[i]++;

  return *this;
}

// Use a friend.
array operator--(array &ob)
{
  int i;

  for(i=0; i<10; i++)
```

```
    ob.nums[i]--;

  return ob;
}

int main()
{
  array o1, o2, o3;

  int i[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  o1.set(i);
  o2.set(i);

  o3 = ++o1;
  o3.show();

  o3 = --o1;
  o3.show();

  return 0;
}
```

7. No. To overload the assignment operator you must use a member function.

7.1 EXERCISES

1. A and C are legal statements.
2. A public member of a base becomes a public member of a derived class when it is inherited as public. When a public member of a base is inherited as private, it becomes a private member of the derived class.

7.2 EXERCISES

. When a protected member of a base class is inherited as public, it becomes a protected member of the derived class. If it is inherited as private, it becomes a private member of the derived class. If it is inherited as protected, it becomes a protected member of the derived class.

2. The protected category is needed to allow a base class to keep certain members private while still allowing a derived class to have access to them.

3. No.

7.3 EXERCISES

1.
```
#include <iostream>
#include <cstring>
using namespace std;

class mybase {
  char str[80];
public:
  mybase(char *s) { strcpy(str, s); }
  char *get() { return str; }
};

class myderived : public mybase {
  int len;
public:
  myderived(char *s) : mybase(s) {
    len = strlen(s);
  }
  int getlen() { return len; }
  void show() { cout << get() << '\n'; }
};

int main()
{
  myderived ob("hello");

  ob.show();
  cout << ob.getlen() << '\n';

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

// A base class for various types of vehicles.
class vehicle {
```

```
    int num_wheels;
    int range;
  public:
    vehicle(int w, int r)
    {
      num_wheels = w; range = r;
    }
    void showv()
    {
      cout << "Wheels: " << num_wheels << '\n';
      cout << "Range: " << range << '\n';
    }
};

class car : public vehicle {
  int passengers;
public:
  car(int p, int w, int r) : vehicle(w, r)
  {
    passengers = p;
  }
  void show()
  {
    showv();
    cout << "Passengers: " << passengers << '\n';
  }
};

class truck : public vehicle {
  int loadlimit;
public:
  truck(int l, int w, int r) : vehicle(w, r)
  {
    loadlimit = l;
  }
  void show()
  {
    showv();
    cout << "loadlimit " << loadlimit << '\n';
  }
};

int main()
{
  car c(5, 4, 500);
```

```
  truck t(30000, 12, 1200);

  cout << "Car: \n";
  c.show();
  cout << "\nTruck:\n";
  t.show();

  return 0;
}
```

7.4 EXERCISES

1.
```
Constructing A
Constructing B
Constructing C
Destructing C
Destructing B
Destructing A
```

2.
```
#include <iostream>
using namespace std;

class A {
  int i;
public:
  A(int a) { i = a; }
};

class B {
  int j;
public:
  B(int a) { j = a; }
};

class C : public A, public B {
  int k;
public:
  C(int c, int b, int a) : A(a), B(b) {
    k = c;
  }
};
```

7.5 EXERCISES

2. A virtual base class is needed when a derived class inherits two (or more) classes, both of which are derived from the same base class. Without virtual base classes, two (or more) copies of the common base class would exist in the final derived class. However, if the original base is virtual, only one copy is present in the final derived class.

MASTERY SKILLS CHECK: Chapter 7

1.
```
#include <iostream>
using namespace std;

class building {
protected:
  int floors;
  int rooms;
  double footage;
};

class house : public building {
  int bedrooms;
  int bathrooms;
public:
  house(int f, int r, double ft, int br, int bth) {
    floors = f;  rooms = r;  footage = ft;
    bedrooms = br; bathrooms = bth;
  }
  void show() {
    cout << "floors: " << floors << '\n';
    cout << "rooms: " << rooms << '\n';
    cout << "square footage: " << footage << '\n';
    cout << "bedrooms: " << bedrooms << '\n';
    cout << "bathrooms: " << bathrooms << '\n';
  }
};

class office : public building {
  int phones;
```

```
  int extinguishers;
public:
  office(int f, int r, double ft, int p, int ext) {
    floors = f;  rooms = r;  footage = ft;
    phones = p; extinguishers = ext;
  }
  void show() {
    cout << "floors: " << floors << '\n';
    cout << "rooms: " << rooms << '\n';
    cout << "square footage: " << footage << '\n';
    cout << "Telephones: " << phones << '\n';
    cout << "fire extinguishers: ";
    cout << extinguishers << '\n';
  }
};

int main()
{
  house h_ob(2, 12, 5000, 6, 4);
  office o_ob(4, 25, 12000, 30, 8);

  cout << "House: \n";
  h_ob.show();

  cout << "\nOffice: \n";
  o_ob.show();

  return 0;
}
```

2. When a base class is inherited as public, the public members of the base become public members of the derived class, and the base's private members remain private to the base. If the base is inherited as private, all members of the base become private members of the derived class.
3. Members declared as **protected** are private to the base class but can be inherited (and accessed) by any derived class. When used as an inheritance access specifier, **protected** causes all public and protected members of the base class to become protected members of the derived class.
4. Constructors are called in order of derivation. Destructors are called in reverse order.

5.
```
#include <iostream>
using namespace std;

class planet {
protected:
  double distance; // miles from the sun
  int revolve;  // in days
public:
  planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
  double circumference; // circumference or orbit
public:
  earth(double d, int r) : planet(d, r) {
    circumference = 2*distance*3.1416;
  }
  void show() {
    cout << "Distance from sun: " << distance << '\n';
    cout << "Days in orbit: " << revolve << '\n';
    cout << "Circumference of orbit: ";
    cout << circumference << '\n';
  }
};

int main()
{
  earth ob(93000000, 365);

  ob.show();

  return 0;
}
```

6. To fix the program, have **motorized** and **road_use** inherit **vehicle** as a virtual base class. Also, refer to Question 1 in the Cumulative Skills Check in this chapter.

CUMULATIVE SKILLS CHECK: Chapter 7

1. Some compilers will not allow you to use a **switch** in an in-line function. If this is the case with your compiler, the functions were automatically made into "regular" functions.

2. The assignment operator is the only operator that is not inherited. The reason for this is easy to understand. Since a derived class will contain members not found in the base class the overloaded = relative to the base has no knowledge of the members added by the derived class and, as such, cannot properly copy those new members.

REVIEW SKILLS CHECK: Chapter 8

1.
```
#include <iostream>
using namespace std;

class airship {
protected:
  int passengers;
  double cargo;
};

class airplane : public airship {
  char engine; // p for propeller, j for jet
  double range;
public:
  airplane(int p, double c, char e, double r)
  {
    passengers = p;
    cargo = c;
    engine = e;
    range = r;
  }
  void show();
};

class balloon : public airship {
  char gas; // h for hydrogen, e for helium
  double altitude;
public:
  balloon(int p, double c, char g, double a)
  {
    passengers = p;
    cargo = c;
    gas = g;
    altitude = a;
  }
```

```
  void show();
};

void airplane::show()
{
  cout << "Passengers: " << passengers << '\n';
  cout << "Cargo capacity: " << cargo << '\n';
  cout << "Engine: ";
  if(engine=='p') cout << "Propeller\n";
  else cout << "Jet\n";
  cout << "Range: " << range << '\n';
}

void balloon::show()
{
  cout << "Passengers: " << passengers << '\n';
  cout << "Cargo capacity: " << cargo << '\n';
  cout << "Gas: ";
  if(gas=='h') cout << "Hydrogen\n";
  else cout << "Helium\n";
  cout << "Altitude: " << altitude << '\n';
}

int main()
{
  balloon b(2, 500.0, 'h', 12000.0);
  airplane b727(100, 40000.0, 'j', 40000.0);

  b.show();
  cout << '\n';
  b727.show();

  return 0;
}
```

2. The **protected** access specifier causes a class member to be private to its class but still accessible by any derived class.
3. The program displays the following output, which indicates when the constructors and destructors are called.

```
Constructing A
Constructing B
Constructing C
Destructing C
```

```
Destructing B
Destructing A
```

4. Constructors are called in the order ABC, destructors in the order CBA.

5.
```
#include <iostream>
using namespace std;

class base {
  int i, j;
public:
  base(int x, int y) { i = x; j = y; }
  void showij() { cout << i << ' ' << j << '\n'; }
};

class derived : public base {
  int k;
public:
  derived(int a, int b, int c) : base(b, c) {
    k = a;
  }
  void show() { cout << k << ' '; showij(); }
};

int main()
{
  derived ob(1, 2, 3);

  ob.show();

  return 0;
}
```

6. The missing words are "general" and "specific."

8.2 EXERCISES

1.
```
#include <iostream>
using namespace std;

int main()
{
```

```
  cout.setf(ios::showpos);

  cout << -10 << ' ' << 10 << '\n';

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::showpoint | ios::uppercase |
            ios::scientific);

  cout << 100.0;

  return 0;
}
```

3.
```
#include <iostream>
using namespace std;

int main()
{
  ios::fmtflags f;

  f = cout.flags(); // store flags

  cout.unsetf(ios::dec);
  cout.setf(ios::showbase | ios::hex);
  cout << 100 << '\n';

  cout.flags(f); // reset flags

  return 0;
}
```

8.3 EXERCISES

1.
```
// Create a table of log10 and log from 2 through 100.
#include <iostream>
#include <cmath>
```

```
using namespace std;

int main()
{
  double x;

  cout.precision(5);
  cout << "          x        log x          ln e\n\n";

  for(x = 2.0; x <= 100.0; x++) {
    cout.width(10);
    cout << x << "   ";
    cout.width(10);
    cout << log10(x) << "   ";
    cout.width(10);
    cout << log(x) << '\n';
  }

  return 0;
}
```

2.
```
#include <iostream>
#include <cstring>
using namespace std;

void center(char *s);

int main()
{
  center("Hi there!");
  center("I like C++.");

  return 0;
}

void center(char *s)
{
  int len;

  len = 40+(strlen(s)/2);

  cout.width(len);
  cout << s << '\n';
}
```

8.4 EXERCISES

1a.
```
// Create a table of log10 and log from 2 to 100.
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  double x;

  cout << setprecision(5);
  cout << "          x        log x        ln e\n\n";

  for(x = 2.0; x <= 100.0; x++) {
    cout << setw(10) << x << "  ";
    cout << setw(10) << log10(x) << "  ";
    cout << setw(10) << log(x) << '\n';
  }

  return 0;
}
```

1b.
```
#include <iostream>
#include <iomanip>
#include <cstring>
using namespace std;

void center(char *s);

int main()
{
  center("Hi there!");
  center("I like C++.");

  return 0;
}

void center(char *s)
{
  int len;
```

```
    len = 40+(strlen(s)/2);

    cout << setw(len) << s << '\n';
  }
```

2. `cout << setiosflags(ios::showbase | ios::hex) << 100;`

3. Setting the **boolalpha** flag on an output stream causes Boolean values to be displayed using the words *true* and *false*. Setting **boolalpha** on an input stream allows you to enter Boolean values using the words *true* and *false*.

8.5 EXERCISES

1.
```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype() {delete [] p;}
  friend ostream &operator<<(ostream &stream, strtype &ob);
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr)+1;
  p = new char [len];
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(p, ptr);
}

ostream &operator<<(ostream &stream, strtype &ob)
{
  stream << ob.p;
```

```
  return stream;
}

int main()
{
  strtype s1("This is a test."), s2("I like C++.");

  cout << s1;
  cout << endl << s2 << endl;

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class planet {
protected:
  double distance; // miles from the sun
  int revolve;     // in days
public:
  planet(double d, int r) { distance = d; revolve = r; }
};

class earth : public planet {
  double circumference; // circumference or orbit
public:
  earth(double d, int r) : planet(d, r) {
    circumference = 2*distance*3.1416;
  }

  friend ostream &operator<<(ostream &stream, earth ob);
};

ostream &operator<<(ostream &stream, earth ob)
{
  stream << "Distance from sun: " << ob.distance << '\n';
  stream << "Days in orbit: " << ob.revolve << '\n';
  stream << "Circumference of orbit: " << ob.circumference;
  stream << '\n';

  return stream;
}
```

```
int main()
{
  earth ob(93000000, 365);

  cout << ob;

  return 0;
}
```

3. An inserter cannot be a member function because the object that generates a call to the inserter is *not* an object of a user-defined class.

8.6 EXERCISES

1.
```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
  int len;
public:
  strtype(char *ptr);
  ~strtype() {delete [] p;}
  friend ostream &operator<<(ostream &stream, strtype &ob);
  friend istream &operator>>(istream &stream, strtype &ob);
};

strtype::strtype(char *ptr)
{
  len = strlen(ptr)+1;
  p = new char [len];
  if(!p) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(p, ptr);
}

ostream &operator<<(ostream &stream, strtype &ob)
{
```

```
  stream << ob.p;

  return stream;
}

istream &operator>>(istream &stream, strtype &ob)
{
  char temp[255];

  stream >> temp;

  if(strlen(temp) >= ob.len) {
    delete [] ob.p;
    ob.len = strlen(temp)+1;
    ob.p = new char [ob.len];
    if(!ob.p) {
      cout << "Allocation error\n";
      exit(1);
    }
  }
  strcpy(ob.p, temp);

  return stream;
}

int main()
{
  strtype s1("This is a test."), s2("I like C++.");

  cout << s1;
  cout << '\n' << s2;

  cout << "\nEnter a string: ";
  cin >> s1;
  cout << s1;

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class factor {
  int num; // number
  int lfact; // lowest factor
public:
```

```
  factor(int i);
  friend ostream &operator<<(ostream &stream, factor ob);
  friend istream &operator>>(istream &stream, factor &ob);
};

factor::factor(int i)
{
  int n;

  num = i;

  for(n=2; n < (i/2); n++)
    if(!(i%n)) break;

  if(n<(i/2)) lfact = n;
  else lfact = 1;
}

istream &operator>>(istream &stream, factor &ob)
{
  stream >> ob.num;

  int n;

  for(n=2; n < (ob.num/2); n++)
    if(!(ob.num%n)) break;
  if(n<(ob.num/2)) ob.lfact = n;
  else ob.lfact = 1;

  return stream;
}

ostream &operator<<(ostream &stream, factor ob)
{
  stream << ob.lfact << " is lowest factor of ";
  stream << ob.num << '\n';

  return stream;
}

int main()
{
  factor o(32);
```

```
  cout << o;

  cin >> o;
  cout << o;

  return 0;
}
```

MASTERY SKILLS CHECK: Chapter 8

1.
```
#include <iostream>
using namespace std;

int main()
{
  cout << 100 << ' ';

  cout.unsetf(ios::dec); // clear dec flag
  cout.setf(ios::hex);
  cout << 100 << ' ';

  cout.unsetf(ios::hex); // clear hex flag
  cout.setf(ios::oct);
  cout << 100 << '\n';

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::left);
  cout.precision(2);
  cout.fill('*');
  cout.width(20);

  cout << 1000.5364 << '\n';

  return 0;
```

3a.
```
#include <iostream>
using namespace std;

int main()
{
  cout << 100 << ' ';

  cout << hex << 100 << ' ';

  cout << oct << 100 << '\n';

  return 0;
}
```

3b.
```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << setiosflags(ios::left);
  cout << setprecision(2);
  cout << setfill('*');
  cout << setw(20);

  cout << 1000.5364 << '\n';

  return 0;
}
```

4.
```
ios::fmtflags f;

f = cout.flags();  // save

// ...

cout.flags(f); // restore
```

5.
```
#include <iostream>
using namespace std;

class pwr {
  int base;
  int exponent;
  double result; // base to the exponent power
```

```
public:
  pwr(int b, int e);
  friend ostream &operator<<(ostream &stream, pwr ob);
  friend istream &operator>>(istream &stream, pwr &ob);
};

pwr::pwr(int b, int e)
{
  base = b;
  exponent = e;

  result = 1;
  for( ; e; e--) result = result * base;
}

ostream &operator<<(ostream &stream, pwr ob)
{
  stream << ob.base << "^" << ob.exponent;
  stream << " is " << ob.result << '\n';

  return stream;
}

istream &operator>>(istream &stream, pwr &ob)
{
  int b, e;

  cout << "Enter base and exponent: ";
  stream >> b >> e;

  pwr temp(b, e); // create temporary

  ob = temp;

  return stream;
}

int main()
{
  pwr ob(10, 2);

  cout << ob;

  cin >> ob;
```

```
  cout << ob;

  return 0;
}
```

6.
```
// This program draws boxes.
#include <iostream>
using namespace std;

class box {
  int len;
public:
  box(int l) { len = l; }
  friend ostream &operator<<(ostream &stream, box ob);
};

// Draw a box.
ostream &operator<<(ostream &stream, box ob)
{
  int i, j;

  for(i=0; i<ob.len; i++) stream << '*';
  stream << '\n';
  for(i=0; i<ob.len-2; i++) {
    stream << '*';
    for(j=0; j<ob.len-2; j++) stream << ' ';
    stream << "*\n";
  }
  for(i=0; i<ob.len; i++) stream << '*';
  stream << '\n';

  return stream;
}

int main()
{
  box b1(4), b2(7);

  cout << b1 << endl << b2;

  return 0;
}
```

CUMULATIVE SKILLS CHECK: Chapter 8

1.
```
#include <iostream>
using namespace std;

#define SIZE 10

// Declare a stack class for characters
class stack {
  char stck[SIZE]; // holds the stack
  int tos; // index of top of stack
public:
  stack();
  void push(char ch); // push character on stack
  char pop(); // pop character from stack
  friend ostream &operator<<(ostream &stream, stack ob);
};

// Initialize the stack
stack::stack()
{
  tos = 0;
}

// Push a character.
void stack::push(char ch)
{
  if(tos==SIZE) {
    cout << "Stack is full\n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Pop a character.
char stack::pop()
{
  if(tos==0) {
    cout << "Stack is empty\n";
    return 0; // return null on empty stack
  }
```

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

ostream &operator<<(ostream &stream, stack ob)
{
  char ch;

  while(ch=ob.pop()) stream << ch;
  stream << endl;

  return stream;
}

int main()
{
  stack s;

  s.push('a');
  s.push('b');
  s.push('c');

  cout << s;
  cout << s;

  return 0;
}
```

2.
```
#include <iostream>
#include <ctime>
using namespace std;

class watch {
  time_t t;
public:
  watch() { t = time(NULL); }
  friend ostream &operator<<(ostream &stream, watch ob);
};

ostream &operator<<(ostream &stream, watch ob)
{
  struct tm *localt;

  localt = localtime(&ob.t);
  stream << asctime(localt) << endl;
```

```
  return stream;
}

int main()
{
  watch w;

  cout << w;

  return 0;
}
```

3.
```
#include <iostream>
using namespace std;

class ft_to_inches {
  double feet;
  double inches;
public:
  void set(double f) {
    feet = f;
    inches = f * 12;
  }
  friend istream &operator>>(istream &stream,
                             ft_to_inches &ob);
  friend ostream &operator<<(ostream &stream,
                             ft_to_inches ob);
};

istream &operator>>(istream &stream, ft_to_inches &ob)
{
  double f;

  cout << "Enter feet: ";
  stream >> f;
  ob.set(f);

  return stream;
}

ostream &operator<<(ostream &stream, ft_to_inches ob)
{
  stream << ob.feet << " feet is " << ob.inches;
  stream << " inches\n";
```

```
  return stream;
}

int main()
{
  ft_to_inches x;

  cin >> x;
  cout << x;

  return 0;
}
```

REVIEW SKILLS CHECK: Chapter 9

1.
```
#include <iostream>
using namespace std;

int main()
{
  cout.width(40);
  cout.fill(':');

  cout << "C++ is fun" << '\n';

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

int main()
{
  cout.precision(4);
  cout << 10.0/3.0 << '\n';

  return 0;
}
```

3.
```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
```

```
{
  cout << setprecision(4) << 10.0/3.0 << '\n';

  return 0;
}
```

4. An inserter is an overloaded **operator<<( )** function that outputs a class's data to an output stream. An extractor is an overloaded **operator>>( )** function that inputs a class's data from an input stream.

5.
```
#include <iostream>
using namespace std;

class date {
  char d[9]; // store date as string: mm/dd/yy
public:
  friend ostream &operator<<(ostream &stream, date ob);
  friend istream &operator>>(istream &stream, date &ob);
};

ostream &operator<<(ostream &stream, date ob)
{
  stream << ob.d << '\n';

  return stream;
}

istream &operator>>(istream &stream, date &ob)
{
  cout << "Enter date (mm/dd/yy): ";
  stream >> ob.d;

  return stream;
}

int main()
{
  date ob;

  cin >> ob;
  cout << ob;

  return 0;
}
```

6. To use a parameteri d manipulato you must include **<iomanip>** in you program.
7. The predefined str ns are **cin**, **cou**, **cerr**, and **clog**.

9.1 EXERCISES

1.
```
// Show time and date.
#include <iostream>
#include <ctime>
using namespace std;

// A time and date output manipulator.
ostream &td(ostream &stream)
{
  struct tm *localt;
  time_t t;

  t = time(NULL);
  localt = localtime(&t);
  stream << asctime(localt) << endl;

  return stream;
}

int main()
{
  cout << td << '\n';

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

// Turn on hex output with uppercase X.
ostream &sethex(ostream &stream)
{
  stream.unsetf(ios::dec | ios::oct);
  stream.setf(ios::hex | ios::uppercase |
              ios::showbase);
```

```
  return stream;
}

// Reset flags.
ostream &reset(ostream &stream)
{
  stream.unsetf(ios::hex | ios::uppercase |
                ios::showbase);
  stream.setf(ios::dec);
  return stream;
}

int main()
{
  cout << sethex << 100 << '\n';
  cout << reset << 100 << '\n';

  return 0;
}
```

3.
```
#include <iostream>
using namespace std;

// Skip 10 characters.
istream &skipchar(istream &stream)
{
  int i;
  char c;

  for(i=0; i<10; i++) stream >> c;

  return stream;
}

int main()
{
  char str[80];

  cout << "Enter some characters: ";
  cin >> skipchar >> str;

  cout << str << '\n';

  return 0;
}
```

9.2 EXERCISES

1.
```
// Copy a text file and display number of chars copied.
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: CPY <input> <output>\n";
    return 1;
  }

  ifstream fin(argv[1]); // open input file
  ofstream fout(argv[2]);  // create output file

  if(!fin) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  if(!fout) {
    cout << "Cannot open output file.\n";
    return 1;
  }

  char ch;
  unsigned count=0;

  fin.unsetf(ios::skipws);  // do not skip spaces
  while(!fin.eof()) {
    fin >> ch;
    if(!fin.eof()) {
      fout << ch;
      count++;
    }
  }

  cout << "Number of bytes copied: " << count << '\n';

  fin.close();
  fout.close();
```

```
  return 0;
}
```

The reason this program might display a result different from that shown when you list the directory is that some character translations might be taking place. Specifically, when a carriage-return/linefeed sequence is read, it is converted into a newline. When output, newlines are counted as one character but converted back into a carriage-return/linefeed sequence again.

2.
```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream pout("phone");

  if(!pout) {
    cout << "Cannot open PHONE file.\n";
    return 1;
  }

  pout << "Isaac Newton  415 555-3423\n";
  pout << "Robert Goddard 213 555-2312\n";
  pout << "Enrico Fermi  202 555-1111\n";

  pout.close();

  return 0;
}
```

3.
```
// Word count.
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: COUNT <input>\n";
```

```
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  int count=0;
  char ch;

  in >> ch; // find first non-space char

  // after first non-space found, do not skip spaces
  in.unsetf(ios::skipws); // do not skip spaces

  while(!in.eof()) {
    in >> ch;
    if(isspace(ch)) {
      count++;
      while(isspace(ch) && !in.eof()) in >> ch;
    }
  }

  cout << "Word count: " << count << '\n';

  in.close();

  return 0;
}
```

4. The **is_open( )** function returns true if the invoking stream is linked to an open file.

9.3 EXERCISES

1a.
```
// Copy a file and display number of chars copied.
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: CPY <input> <output>\n";
    return 1;
  }

  ifstream fin(argv[1], ios::in | ios::binary); // open input file
  ofstream fout(argv[2], ios::out | ios::binary); // create output file

  if(!fin) {
    cout << "Cannot open input file\n";
    return 1;
  }

  if(!fout) {
    cout << "Cannot open output file\n";
    return 1;
  }

  char ch;
  unsigned count=0;

  while(!fin.eof()) {
    fin.get(ch);
    if(!fin.eof()) {
      fout.put(ch);
      count++;
    }
  }

  cout << "Number of bytes copied: " << count << '\n';

  fin.close();
  fout.close();

  return 0;
}
```

1b.

```
// Word count.
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;
```

```
int main(int a: :, char *argv[]
{
  if(argc!=2) {
    cout << "Usa: : COUNT <inpu  n";
    return 1;
  }

  ifstream in(argv 1], ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  int count=0;
  char ch;

  // find first non-space char
  do {
    in.get(ch);
  } while(isspace(ch));

  while(!in.eof()) {
    in.get(ch);
    if(isspace(ch)) {
      count++;
      while(isspace(ch) && !in.eof()) in.get(ch); // find next word
    }
}

  cout << "Word count: " << count << '\n';

  in.close();

  return 0;
}
```

2.
```
// Output account info to a file using an inserter.
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

class account {
  int custnum;
```

```
  char name[80];
  double balance;
public:
  account(int c, char *n, double b)
  {
    custnum = c;
    strcpy(name, n);
    balance = b;
  }
  friend ostream &operator<<(ostream &stream, account ob)
};

ostream &operator<<(ostream &stream, account ob)
{
  stream << ob.custnum << ' ';
  stream << ob.name << ' ' << ob.balance;
  stream << '\n';

  return stream;
}

int main()
{
  account  Rex(1011, "Ralph Rex", 12323.34);
  ofstream out("accounts", ios::out | ios::binary);

  if(!out) {
    cout << "Cannot open output file.\n";
    return 1;
  }

  out << Rex;

  out.close();

  return 0;
}
```

9.4 EXERCISES

1.
```
// Use get() to read a string that contains spaces
#include <iostream>
#include <fstream>
```

```
using namespace std;

int main()
{
  char str[80];

  cout << "Enter your name: ";
  cin.get(str, 79);

  cout << str << '\n';

  return 0;
}
```

The program functions the same whether it uses **get( )** or **getline( )**.

2.
```
// Use getline() to display a text file.
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "usage: PR <filename>\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  char str[255];

  while(!in.eof()) {
    in.getline(str, 254);
    cout << str << '\n';
  }

  in.close();
```

```
    return 0;
}
```

9.5 EXERCISES

1.
```
// Display a file backwards on the screen.
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "usage: REVERSE <filename>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  char ch;
  long i;

  // go to end of file (less eof char)
  in.seekg(0, ios::end);
  i = (long) in.tellg(); // see how many bytes in file
  i -= 2; // backup before eof

  for( ;i>=0; i--) {
    in.seekg(i, ios::beg);
    in.get(ch);
    cout << ch;
  }

  in.close();

  return 0;
}
```

2.
```
// Swap characters in a file.
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "usage: SWAP <filename>\n";
    return 1;
  }

  // open file for input/output
  fstream io(argv[1], ios::in | ios::out | ios::binary);
  if(!io) {
    cout << "Cannot open file.\n";
    return 1;
  }

  char ch1, ch2;
  long i;

  for(i=0 ; !io.eof(); i+=2) {
    io.seekg(i, ios::beg);
    io.get(ch1);
    if(io.eof()) continue;
    io.get(ch2);
    if(io.eof()) continue;
    io.seekg(i, ios::beg);
    io.put(ch2);
    io.put(ch1);
  }

  io.close();

  return 0;
}
```

9.6 EXERCISE

1a.
```
/* Display a file backwards on the screen,
   plus error checking. */
#include <iostream>
#include <fstream>
```

```
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "usage: REVERSE <filename>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file\n";
    return 1;
  }

  char ch;
  long i;

  // go to end of file (less eof char)
  in.seekg(0, ios::end);
  if(!in.good()) return 1;
  i = (long) in.tellg(); // see how many bytes in file
  if(!in.good()) return 1;
  i -= 2; // backup before eof

  for( ;i>=0; i--) {
    in.seekg(i, ios::beg);
    if(!in.good()) return 1;
    in.get(ch);
    if(!in.good()) return 1;
    cout << ch;
  }

  in.close();
  if(!in.good()) return 1;

  return 0;
}
```

1b.

```
// Swap characters in a file with error checking.
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "usage: SWAP <filename>\n";
    return 1;
  }

  // open file for input/output
  fstream io(argv[1], ios::in | ios::out | ios::binary);

  if(!io) {
    cout << "Cannot open file.\n";
    return 1;
  }

  char ch1, ch2;
  long i;

  for(i=0 ;!io.eof(); i+=2) {
    io.seekg(i, ios::beg);
    if(!io.good()) return 1;
    io.get(ch1);
    if(io.eof()) continue;
    io.get(ch2);
    if(!io.good()) return 1;
    if(io.eof()) continue;
    io.seekg(i, ios::beg);
    if(!io.good()) return 1;
    io.put(ch2);
    if(!io.good()) return 1;
    io.put(ch1);
    if(!io.good()) return 1;
  }

  io.close();
  if(!io.good()) return 1;

  return 0;
}
```

MASTERY SKILLS CHECK: Chapter 9

1.
```
#include <iostream>
using namespace std;

ostream &tabs(ostream &stream)
{
  stream << '\t' << '\t' << '\t' ;
  stream.width(20);

  return stream;
}

int main()
{
  cout << tabs << "Testing\n";

  return 0;
}
```

2.
```
#include <iostream>
#include <cctype>
using namespace std;

istream &findalpha(istream &stream)
{
  char ch;

  do {
    stream.get(ch);
  } while(!isalpha(ch));
  return stream;
}

int main()
{
  char str[80];

  cin >> findalpha >> str;
  cout << str << '\n';

  return 0;
}
```

3.
```
// Copy a file and reverse case of letters.
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Usage: COPYREV <source> <target>\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  ofstream out(argv[2]);

  if(!out) {
    cout << "Cannot open output file.\n";
    return 1;
  }

  while(!in.eof()) {
    ch = in.get();
    if(!in.eof()) {
      if(islower(ch)) ch = toupper(ch);
      else ch = tolower(ch);
      out.put(ch);
    }
  };

  in.close();
  out.close();

  return 0;
}
```

4.
```
// Count letters.
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int alpha[26];

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Usage: COUNT <source>\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  // init alpha[]
  int i;
  for(i=0; i<26; i++) alpha[i] = 0;

  while(!in.eof()) {
    ch = in.get();
    if(isalpha(ch)) { // if letter found, count it
      ch = toupper(ch); // normalize
      alpha[ch-'A']++; // 'A'-'A' == 0, 'B'-'A' == 1, etc.
    }
  };

  // display count
  for(i=0; i<26; i++) {
    cout << (char) ('A'+ i) << ": " << alpha[i] << '\n';
  }

  in.close();

  return 0;
```

5a.
```
/* Copy a file and reverse case of letters
   with error checking. */
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Usage: COPYREV <source> <target>\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  ofstream out(argv[2]);

  if(!out) {
    cout << "Cannot open output file";
    return 1;
  }

  while(!in.eof()) {
    ch = in.get();
    if(!in.good() && !in.eof()) return 1;
    if(!in.eof()) {
      if(islower(ch)) ch = toupper(ch);
      else ch = tolower(ch);
      out.put(ch);
      if(!out.good()) return 1;
    }
  };

  in.close();
  out.close();
  if(!in.good() && !out.good()) return 1;
```

```
  return 0;
}
```

5b.

```
// Count letters with error checking.
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int alpha[26];

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Usage: COUNT <source>\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  // init alpha[]
  int i;
  for(i=0; i<26; i++) alpha[i] = 0;

  while(!in.eof()) {
    ch = in.get();
    if(!in.good() && !in.eof()) return 1;
    if(isalpha(ch)) { // if letter found, count it
      ch = toupper(ch); // normalize
      alpha[ch-'A']++; // 'A'-'A' == 0, 'B'-'A' == 1, etc.
    }
  };

  // display count
  for(i=0; i<26; i++) {
    cout << (char) ('A'+ i) << ": " << alpha[i] << '\n';
  }
```

```
  in.close();
  if(!in.good()) return 1;

  return 0;
}
```

6. To set the **get** pointer, use **seekg( )**. To set the **put** pointer, use **seekp( )**.

CUMULATIVE SKILLS CHECK: Chapter 9

1.
```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

#define SIZE 40

class inventory {
  char item[SIZE]; // name of item
  int onhand; // number on hand
  double cost;  // cost of item
public:
  inventory(char *i, int o, double c)
  {
    strcpy(item, i);
    onhand = o;
    cost = c;
  }
  void store(fstream &stream);
  void retrieve(fstream &stream);
  friend ostream &operator<<(ostream &stream, inventory ob);
  friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
  stream << ob.item << ": " << ob.onhand;
  stream << " on hand at $" << ob.cost << '\n';

  return stream;
}

istream &operator>>(istream &stream, inventory &ob)
```

```
  cout << "Enter item name: ";
  stream >> ob.item;
  cout << "Enter number on hand: ";
  stream >> ob.onhand;
  cout << "Enter cost: ";
  stream >> ob.cost;

  return stream;
}

void inventory::store(fstream &stream)
{
  stream.write(item, SIZE);
  stream.write((char *) &onhand, sizeof(int));
  stream.write((char *) &cost, sizeof(double));
}

void inventory::retrieve(fstream &stream)
{
  stream.read(item, SIZE);
  stream.read((char *) &onhand, sizeof(int));
  stream.read((char *) &cost, sizeof(double));
}

int main()
{

  fstream inv("inv", ios::out | ios::binary);
  int i;

  inventory pliers("pliers", 12, 4.95);
  inventory hammers("hammers", 5, 9.45);
  inventory wrenches("wrenches", 22, 13.90);
  inventory temp("", 0, 0.0);

  if(!inv) {
    cout << "Cannot open file for output.\n";
    return 1;
  }
  // write to file
  pliers.store(inv);
  hammers.store(inv);
  wrenches.store(inv);
```

```
  inv.close();

  // open for input
  inv.open("inv", ios::in | ios::binary);

  if(!inv) {
    cout << "Cannot open file for input.\n";
    return 1;
  }

  do {
    cout << "Record # (-1 to quit): ";
    cin >> i;
    if(i == -1) break;
    inv.seekg(i*(SIZE+sizeof(int)+sizeof(double)), ios::beg);
    temp.retrieve(inv);
    cout << temp;
  } while(inv.good());

  inv.close();

  return 0;
}
```

REVIEW SKILLS CHECK: Chapter 10

1.
```
#include <iostream>
using namespace std;

ostream &setsci(ostream &stream)
{
  stream.setf(ios::scientific | ios::uppercase);

  return stream;
}

int main()
{
  double f = 123.23;

  cout << setsci << f;
  cout << '\n';
```

```
  return 0;
}
```

2.
```
// Copy and convert tabs to spaces.
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: CPY <in> <out>\n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  ofstream out(argv[2]);
  if(!out) {
    cout << "Cannot open output file.\n";
    return 1;
  }

  char ch;
  int i = 8;

  while(!in.eof()) {
    in.get(ch);
    if(ch=='\t') for( ; i>0; i--) out.put(' ');
    else out.put(ch);
    if(i == -1 || ch=='\n') i = 8;
    i--;
  }

  in.close();
  out.close();

  return 0;
}
```

3.
```
// Search file.
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: SEARCH <file> <word>\n";
    return 1;
  }

  ifstream in(argv[1]);
  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  char str[255];
  int count=0;

  while(!in.eof()) {
    in >> str;
    if(!strcmp(str, argv[2])) count++;
  }

  cout << argv[2] << " found " << count;
  cout << " number of times.\n";

  in.close();

  return 0;
}
```

4. The statement is

```
out.seekp(234, ios::beg);
```

5. The functions are **rdstate( )**, **good( )**, **eof( )**, **fail( )**, and **bad( )**.

6. The C++ I/O system can be customized to operate on classes that you create.

10.2 EXERCISES

1.
```
#include <iostream>
using namespace std;

class num {
public:
  int i;
  num(int x) { i = x; }
  virtual void shownum() { cout << i << '\n'; }
};

class outhex : public num {
public:
  outhex(int n) : num(n) {}
  void shownum() { cout << hex << i << '\n'; }
};

class outoct : public num {
public:
  outoct(int n) : num(n) {}
  void shownum() { cout << oct << i << '\n'; }
};

int main()
{
  outoct o(10);
  outhex h(20);

  o.shownum();
  h.shownum();

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class dist {
public:
  double d;
  dist(double f) { d = f; }
  virtual void trav_time()
  {
```

```
    cout << "Travel time at 60 mph: ";
    cout << d / 60 << '\n';
  }
};

class metric : public dist {
public:
  metric(double f) : dist(f) {}
  void trav_time()
  {
    cout << "Travel time at 100 kph: ";
    cout << d / 100 << '\n';
  }
};

int main()
{
  dist *p, mph(88.0);
  metric kph(88);

  p = &mph;
  p->trav_time();

  p = &kph;
  p->trav_time();

  return 0;
}
```

10.3 EXERCISES

2. By definition, an abstract class contains at least one pure virtual function. This means that no body for that function exists relative to that class. Thus, there is no way that an object can be created because the class definition is not complete.
3. When **func( )** is called relative to **derived1**, it is the **func( )** inside **base** that is used. The reason this works is that virtual functions are hierarchical.

10.4 EXERCISE

1.
```
// Demonstrate virtual functions.
#include <iostream>
#include <cstdlib>
using namespace std;

class list {
public:
  list *head;  // pointer to start of list
  list *tail;  // pointer to end of list
  list *next;  // pointer to next item
  int num; // value to be stored

  list() { head = tail = next = NULL; }
  virtual void store(int i) = 0;
  virtual int retrieve() = 0;
};

// Create a queue-type list.
class queue : public list {
public:
  void store(int i);
  int retrieve();
};

void queue::store(int i)
{
  list *item;

  item = new queue;
  if(!item) {
    cout << "Allocation error.\n";
    exit(1);
  }
  item->num = i;

  // put on end of list
  if(tail) tail->next = item;
  tail = item;
  item->next = NULL;
  if(!head) head = tail;
}
```

```
int queue::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "List empty.\n";
    return 0;
  }

  // remove from start of list
  i = head->num;
  p = head;
  head = head->next;
  delete p;

  return i;
}

// Create a stack-type list.
class stack : public list {
public:
  void store(int i);
  int retrieve();
};

void stack::store(int i)
{
  list *item;

  item = new stack;
  if(!item) {
    cout << "Allocation error.\n";
    exit(1);
  }
  item->num = i;

  // put on front of list for stack-like operation
  if(head) item->next = head;
  head = item;
  if(!tail) tail = head;
}

int stack::retrieve()
{
```

```
  int i;
  list *p;

  if(!head) {
    cout << "L  . empty.\n";
    return 0;
  }

  // remove fro  start of list
  i = head->nur .
  p = head;
  head = head-> ext;
  delete p;

  return i;
}

// Create a sorted list.
class sorted : public list {
public:
  void store(int i);
  int retrieve();
};

void sorted::store(int i)
{
  list *item;
  list *p, *p2;

  item = new sorted;
  if(!item) {
    cout << "Allocation error.\n";
    exit(1);

  }
  item->num = i;

  // find where to put next item
  p = head;
  p2 = NULL;
  while(p) { // goes in middle
    if(p->num > i) {
      item->next = p;
      if(p2) p2->next = item;  // not 1st element
      if(p==head) head = item; // new 1st element
```

```
        break;
      }
      p2 = p;
      p = p->next;
    }
    if(!p) { // goes on end
      if(tail) tail->next = item;
      tail = item;
      item->next = NULL;
    }
    if(!head) // is first element
      head = item;
}

int sorted::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "List empty.\n";
    return 0;
  }

  // remove from start of list
  i = head->num;
  p = head;
  head = head->next;
  delete p;

  return i;
}

int main()
{
  list *p;

  // demonstrate queue
  queue q_ob;
  p = &q_ob; // point to queue

  p->store(1);
  p->store(2);
  p->store(3);
```

```
  cout << "Queue: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '\n';

  // demonstrate stack
  stack s_ob;
  p = &s_ob; // point to stack

  p->store(1);
  p->store(2);
  p->store(3);

  cout << "Stack: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '\n';

  // demonstrate sorted list
  sorted sorted_ob;
  p = &sorted_ob;

  p->store(4);
  p->store(1);
  p->store(3);
  p->store(9);
  p->store(5);

  cout << "Sorted: ";
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();
  cout << p->retrieve();

  cout << '\n';


  return 0;
}
```

MASTERY SKILLS CHECK: Chapter 10

1. A virtual function is essentially a placeholder function that is declared in a base class and that is redefined by a class derived from that base. The process of redefinition is called overriding.
2. Nonmember functions and constructor functions cannot be made virtual.
3. A virtual function supports run-time polymorphism through the use of base class pointers. When a base class pointer points to an object of a derived class containing a virtual function, the specific function called is determined by the type of object being pointed to.
4. A pure virtual function is one that contains no definition relative to the base class.
5. An abstract class is a base class that contains at least one pure virtual function. A polymorphic class is one that contains at least one virtual function.
6. The fragment is incorrect because the redefinition of a virtual function must have the same return type and type and number of parameters as the original function. In this case, the redefinition of **f( )** differs in the number of its parameters.
7. Yes.

CUMULATIVE SKILLS CHECK: Chapter 10

1.
```
// Demonstrate virtual functions.
#include <iostream>
#include <cstdlib>
using namespace std;

class list {
public:
  list *head;  // pointer to start of list
  list *tail;  // pointer to end of list
  list *next;  // pointer to next item
  int num; // value to be stored

  list() { head = tail = next = NULL; }
  virtual void store(int i) = 0;
  virtual int retrieve() = 0;
```

```
// Create a queue-type list.
class queue : public list {
public:
  void store(int i);
  int retrieve();
  queue operator+(int i) { store(i); return *this; }
  int operator--(int unused) { return retrieve(); }
};

void queue::store(int i)
{
  list *item;

  item = new queue;
  if(!item) {
    cout << "Allocation error.\n";
    exit(1);
  }
  item->num = i;

  // put on end of list
  if(tail) tail->next = item;
  tail = item;
  item->next = NULL;
  if(!head) head = tail;
}

int queue::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "List empty.\n";
    return 0;
  }

  // remove from start of list
  i = head->num;
  p = head;
  head = head->next;
  delete p;
```

```
  return i;
}

// Create a stack-type list.
class stack : public list {
public:
  void store(int i);
  int retrieve();
  stack operator+(int i) { store(i); return *this; }
  int operator--(int unused) { return retrieve(); }
};

void stack::store(int i)
{
  list *item;

  item = new stack;
  if(!item) {
    cout << "Allocation error.\n";
    exit(1);
  }
  item->num = i;

  // put on front of list for stack-like operation
  if(head) item->next = head;
  head = item;
  if(!tail) tail = head;
}

int stack::retrieve()
{
  int i;
  list *p;

  if(!head) {
    cout << "List empty.\n";
    return 0;
  }

  // remove from start of list
  i = head->num;
  p = head;
  head = head->next;
  delete p;
```

```
  return i;
}

int main()
{
  // demonstrate queue
  queue q_ob;

  q_ob + 1;
  q_ob + 2;
  q_ob + 3;

  cout << "Queue: ";
  cout << q_ob--;
  cout << q_ob--;
  cout << q_ob--;

  cout << '\n';

  // demonstrate stack
  stack s_ob;

  s_ob + 1;
  s_ob + 2;
  s_ob + 3;

  cout << "Stack: ";
  cout << s_ob--;
  cout << s_ob--;
  cout << s_ob--;

  cout << '\n';

  return 0;
}
```

2. Virtual functions differ from overloaded functions in that overloaded functions *must* differ in either the number of parameters or the type of parameters. An overridden virtual function must have exactly the same prototype (that is, the same return type and the same type and number of parameters) as the original function.

REVIEW SKILLS CHECK: Chapter 11

1. A virtual function is a function that is declared as **virtual** by a base class and then overridden by a derived class.
2. A pure virtual function is one that has no body defined within the base class. This means that the function must be overridden by a derived class. A base class that contains at least one pure virtual function is called an abstract class.
3. The missing words are "virtual" and "base".
4. If a derived class does not override a non-pure virtual function, the derived class will use the base class's version of the virtual function.
5. The main advantage of run-time polymorphism is flexibility. **The main disadvantage is loss of ex**ecution speed.

11.1 EXERCISES

2.
```
#include <iostream>
using namespace std;

template <class X> X min(X a, X b)
{
  if(a<=b) return a;
  else return b;
}

int main()
{
  cout << min(12.2, 2.0);
  cout << endl;
  cout << min(3, 4);
  cout << endl;
  cout << min('c', 'a');
  return 0;
}
```

3.
```
#include <iostream>
#include <cstring>
using namespace std;

template <class X> int find(X object, X *list, int size
```

```
  int i;

  for(i=0; i<size; i++)
    if(object == list[i]) return i;
  return -1;
}

int main()
{
  int a[] = {1, 2, 3, 4};
  char *c = "this is a test";
  double d[] = {1.1, 2.2, 3.3};

  cout << find(3, a, 4);
  cout << endl;
  cout << find('a', c, (int) strlen(c));
  cout << endl;
  cout << find(0.0, d, 3);

  return 0;
}
```

4. Generic functions are valuable because they allow you to define a general algorithm that can be applied to various types of data. (That is, specific versions of the algorithm need not be explicitly created by you.) Generic functions further help implement the concept of "one interface, multiple methods," which is a common theme in C++ programming.

11.2 EXERCISES

2.
```
// Create a generic queue.
#include <iostream>
using namespace std;

#define SIZE 100

template <class Qtype> class q_type {
  Qtype queue[SIZE]; // holds the queue
  int head, tail; // indices of head and tail
public:
  q_type() { head = tail = 0; }
```

```
  void q(Qtype num); // store
  Qtype deq();  // retrieve
};

// Put value on queue.
template <class Qtype> void q_type<Qtype>::q(Qtype num)
{
  if(tail+1==head || (tail+1==SIZE && !head)) {
    cout << "Queue is full.\n";
    return;
  }
  tail++;
  if(tail==SIZE) tail = 0; // cycle around
  queue[tail] = num;
}

// Remove value from queue.
template <class Qtype> Qtype q_type<Qtype>::deq()
{
  if(head == tail) {
    cout << "Queue is empty.\n";
    return 0;  // or some other error indicator
  }
  head++;
  if(head==SIZE) head = 0; // cycle around
  return queue[head];
}

int main()
{
  q_type<int> q1;
  q_type<char> q2;
  int i;

  for(i=1; i<=10; i++) {
    q1.q(i);
    q2.q(i-1+'A');
  }

  for(i=1; i<=10; i++) {
    cout << "Dequeue 1: " << q1.deq() << "\n";
    cout << "Dequeue 2: " << q2.deq() << "\n";
  }
```

```
  return 0;
}
```

3.
```
#include <iostream>
using namespace std;

template <class X> class input {
  X data;
public:
  input(char *s, X min, X max);
  // ...
};

template <class X>
input<X>::input(char *s, X min, X max)
```

11.4 EXERCISES

2. There is no corresponding **catch** statement for the **throw**.
3. One way to fix the problem is to create a **catch(int)** handler. Another way to fix it is to catch all exceptions with a **catch(...)** handler.
4. **catch(...)** catches all exceptions.
5.
```
#include <iostream>
#include <cstdlib>
using namespace std;

double divide(double a, double b)
{
  try {
    if(!b) throw(b);
  }
  catch(double) {
    cout << "Cannot divide by zero.\n";
    exit(1);
  }
  return a/b;
}

int main()
{
  cout << divide(10.0, 2.5) << endl;
  cout << divide(10.0, 0.0);

  return 0;
```

11.5 EXERCISES

1. By default, **new** throws an exception when an allocation error occurs. The **nothrow** version of **new** returns a null pointer if memory cannot be allocated.
2.
```
p = new(nothrow) int;

if(!p) {
  cout << "Allocation error.\n";
  // ...
```

```
try {
  p = new int;
} catch(bad_alloc ba) {
  cout << "Allocation error.\n";
  // ...
}
```

MASTERY SKILLS CHECK: Chapter 11

1.
```
#include <iostream>
#include <cstring>
using namespace std;

// A generic mode-finding function.
template <class X> X mode(X *data, int size)
{
  register int t, w;
  X md, oldmd;
  int count, oldcount;

  oldmd = 0;
  oldcount = 0;
  for(t=0; t<size; t++) {
    md = data[t];
    count = 1;
    for(w = t+1; w < size; w++)
      if(md==data[w]) count++;
    if(count > oldcount) {
      oldmd = md;
      oldcount = count;
    }
  }
  return oldmd;
}

int main()
{
  int i[] = { 1, 2, 3, 4, 2, 3, 2, 2, 1, 5};
  char *p = "this is a test";
```

```
  cout << "mode of i: " << mode(i, 10) << endl;
  cout << "mode of p: " << mode(p, (int) strlen(p));

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

template <class X> X sum(X *data, int size)
{
  int i;
  X result = 0;

  for(i=0; i<size; i++) result += data[i];

  return result;
}

int main()
{
  int i[] = {1, 2, 3, 4};
  double d[] = {1.1, 2.2, 3.3, 4.4};

  cout << sum(i, 4) << endl;
  cout << sum(d, 4) << endl;

  return 0;
}
```

3.
```
#include <iostream>
using namespace std;

// A generic bubble sort.
template <class X> void bubble(X *data, int size)
{
  register int a, b;
  X t;

  for(a=1; a < size; a++)
    for(b=size-1; b >= a; b--)
      if(data[b-1] > data[b]) {
        t = data[b-1];
        data[b-1] = data[b];
        data[b] = t;
```

```
int main()
{
  int i[] = {3, 2, 5, 6, 1, 8, 9, 3, 6, 9};
  double d[] = {1.2, 5.5, 2.2, 3.3};
  int j;

  bubble(i, 10); \\ sort ints
  bubble(d, 4);  \\ sort doubles

  for(j=0; j<10; j++) cout << i[j] << ' ';
  cout << endl;

  for(j=0; j<4; j++) cout << d[j] << ' ';
  cout << endl;

  return 0;
}
```

4. `/* This function demonstrates a generic stack that holds two values. */`

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

#define SIZE 10

\\ Create a generic stack class
template <class StackType> class stack {
  StackType stk[SIZE][2]; \\ holds the stack
  int tos; \\ index of top of stack

public:
  void init() { tos = 0; }
  void push(StackType ob, StackType ob2);
  StackType pop(StackType &ob2);
};

\\ Push object.
template <class StackType>
void stack<StackType>::push(StackType ob, StackType ob2)
{
  if(tos==SIZE) {
    cout << "Stack is full./n";
```

```
    return;
  }
  stck[tos][0] = ob;
  stck[tos][1] = ob2;
  tos++;
}

// Pop objects.
template <class StackType>
StackType stack<StackType>::pop(StackType &ob2)
{
  if(tos==0) {
    cout << "Stack is empty.\n";
    return 0; // return null on empty stack
  }
  tos--;
  ob2 = stck[tos][1];
  return stck[tos][0];
}

int main()
{
  // Demonstrate character stacks.
  stack<char> s1, s2;  // create two stacks
  int i;
  char ch;

  // initialize the stacks
  s1.init();
  s2.init();

  s1.push('a', 'b');
  s2.push('x', 'z');
  s1.push('b', 'd');
  s2.push('y', 'e');
  s1.push('c', 'a');
  s2.push('z', 'x');

  for(i=0; i<3; i++) {
    cout << "Pop s1: " << s1.pop(ch);
    cout << ' ' << ch << "\n";
  }
  for(i=0; i<3; i++) {
    cout << "Pop s2: " << s2.pop(ch);
    cout << ' ' << ch << "\n";
```

```
}

// demonstrate double stacks
stack<double> ds1, ds2;  // create two stacks
double d;

// initialize the stacks
ds1.init();
ds2.init();

ds1.push(1.1, 2.0);
ds2.push(2.2, 3.0);
ds1.push(3.3, 4.0);
ds2.push(4.4, 5.0);
ds1.push(5.5, 6.0);
ds2.push(6.6, 7.0);

for(i=0; i<3; i++) {
  cout << "Pop ds1: " << ds1.pop(d);
  cout << ' '<< d << "\n";
}

for(i=0; i<3; i++) {
  cout << "Pop ds2: " << ds2.pop(d);
  cout << ' '<< d << "\n";
}

return 0;
}
```

5. The general forms of **try**, **catch**, and **throw** are shown here:

```
try {
 // try block
 throw exp;
}
catch(type arg) {
 // ...
}
```

6.
```
/* This function demonstrates a generic stack
   that includes exception handling. */
#include <iostream>
```

```
using namespace std;

#define SIZE 10

// Create a generic stack class
template <class StackType> class stack {
  StackType stck[SIZE]; // holds the stack
  int tos; // index of top of stack

public:
  void init() { tos = 0; } // initialize stack
  void push(StackType ch); // push object on stack
  StackType pop(); // pop object from stack
};

// Push an object.
template <class StackType>
void stack<StackType>::push(StackType ob)
{
  try {
    if(tos==SIZE) throw SIZE;
  }
  catch(int) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = ob;
  tos++;
}

// Pop an object.
template <class StackType>
StackType stack<StackType>::pop()
{
  try {
    if(tos==0) throw 0;
  }
  catch(int) {
    cout << "Stack is empty.\n";
    return 0; // return null on empty stack
  }
  tos--;
  return stck[tos];
}
```

```
int main()
{
  // Demonstrate character stacks.
  stack<char> s1, s2;  // create two stacks
  int i;
  // initialize the stacks
  s1.init();
  s2.init();

  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
  for(i=0; i<4; i++) cout << "Pop s2: " << s2.pop() << "\n";

  // demonstrate double stacks
  stack<double> ds1, ds2;  // create two stacks

  // initialize the stacks
  ds1.init();
  ds2.init();

  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);

  for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";
  for(i=0; i<4; i++) cout << "Pop ds2: " << ds2.pop() << "\n";

  return 0;
}
```

8. If **new** throws an exception when an allocation error occurs, you can be sure that the error will be handled one way or another—even if only by abnormal program termination. In contrast, an allocation failure that is reported by **new**'s return of a null pointer can be overlooked if you forget to check for this

possibility. The trouble is that when your program attempts to use the null pointer, it might work for a while, then behave erratically, and finally crash in unpredictable (and unduplicatable) ways. This is a very difficult type of bug to diagnose.

REVIEW SKILLS CHECK: Chapter 12

1. In C++, a generic function defines a general set of operations that will be applied to various types of data. It is implemented with the keyword **template**. Its general form is shown here:

```
template <class Ttype> ret-type func-name(param-list)
{
  // ...
}
```

2. In C++, a generic class defines all operations that relate to that class, but the actual data is specified as a parameter when an object of that class is created. Its general form is shown here:

```
template <class Ttype> class class-name
{
  // ...
}
```

3.
```
#include <iostream>
using namespace std;

// Return a to the b.
template <class X> X gexp(X a, X b)
{
   X i, result=1;

  for(i=0; i<b; i++) result *= a;

  return result;
}

int main()
{
  cout << gexp(2, 3) << endl;
```

```
  cout << gexp(10.0, 2.0);

  return 0;
}
```

4.
```
#include <iostream>
#include <fstream>
using namespace std;

template <class CoordType> class coord {
  CoordType x, y;
public:
  coord(CoordType i, CoordType j) { x = i; y = j; }
  void show() { cout << x << ", " << y << endl; }
};

int main()
{
  coord<int> o1(1, 2), o2(3, 4);

  o1.show();
  o2.show();

  coord<double> o3(0.0, 0.23), o4(10.19, 3.098);

  o3.show();
  o4.show();

  return 0;
}
```

5. **try**, **catch**, and **throw** work together like this: Put all statements that you wish to monitor for exceptions within a **try** block. If an exception occurs, throw that exception using **throw** and handle it with a corresponding **catch** statement.
6. No.
7. **terminate( )** is called when an exception is thrown for which there is no corresponding **catch** statement. **unexpected( )** is called when an attempt is made to throw an exception out of a function that is not specified in the function's **throws** clause.
8. **catch(...)**

12.1 EXERCISES

1. RTTI is necessary to C++ because it is not always possible to know at compile time what type of object is being pointed to by a base class pointer or referenced by a base class reference.
2. Here is the output produced when **BaseClass** is no longer a polymorphic class.

```
Typeid of i is int
p is pointing to an object of type class BaseClass
p is pointing to an object of type class BaseClass
p is pointing to an object of type class BaseClass
```

3. Yes.
4. `if(typeid(*p) == typeid(D2)) ...`
5. False. Although the same template class is used, the type of data used by each version is different.

12.2 EXERCISES

1. The **dynamic_cast** operator determines the validity of a polymorphic cast.
2.

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

class B {
  virtual void f() {}
};

class D1: public B {
  void f() {}
};

class D2: public B {
  void f() {}
};

int main()
{
  B *p;
```

```
  D2 ob;

  p = dynamic_cast<D2 *> (&ob);

  if(p) cout << "Cast OK";
  else cout << "Cast Fails";

  return 0;
}
```

3.
```
int main()
{
  int i;
  Shape *p;

  for(i=0; i<10; i++) {
    p = generator(); // get next object

    cout << typeid(*p).name() << endl;

    // draw object only if it is not a NullShape
    if(!dynamic_cast<NullShape *> (p))
      p->example();
  }

  return 0;
}
```

4. No. **Bp** and **Dp** are pointers to two fundamentally different types of objects.

.3 EXERCISES

1. The new casting operators provide safer, more explicit ways tc perform casts.

2.
```
#include <iostream>
using namespace std;

void f(const double &i)
{
  double &v = const_cast<double &> (i);

  v = 100.0;
```

```
int main()
{
  double x = 98.6;

  cout << x << endl;
  f(x);
  cout << x << endl;

  return 0;
}
```

3. Since **const_cast** overrides the **const** specifier, it allows unexpected modifications to objects.

MASTERY SKILLS CHECK: Chapter 12

1. The **typeid** operator returns a reference to an object of the class **type_info** that contains type information.
2. You must include **<typeinfo>** in order to use **typeid**.
3. The new casting operators are shown here.

Operator	Purpose
dynamic_cast	Performs a polymorphic cast
reinterpret_cast	Converts one type of pointer into another
static_cast	Performs a "normal" cast
const_cast	Casts away **const**-ness

4.
```
#include <iostream>
#include <typeinfo>
using namespace std;

class A {
  virtual void f() {}
};

class B : public A {
};
```

```
class C: public B {
};

int main()
{
  A *p, a_ob;
  B b_ob;
  C c_ob;
  int i;

  cout << "Enter 0 for A objects, ";
  cout << "1 for B objects or ";
  cout << "2 for C objects.\n";

  cin >> i;

  if(i==1) p = &b_ob;
  else if(i==2) p = &c_ob;
  else p = &a_ob;

  if(typeid(*p) == typeid(A))
    cout << "A object";
  if(typeid(*p) == typeid(B))
    cout << "B object";
  if(typeid(*p) == typeid(C))
    cout << "C object";

  return 0;
}
```

5. The **dynamic_cast** operator can replace **typeid** in situations in which **typeid** is being used to determine the validity of a polymorphic cast.
6. The **typeid** operator returns a reference to a **type_info** object.

CUMULATIVE SKILLS CHECK: Chapter 12

1. Here is a version of **generator( )** that uses exception handling to check for an allocation failure.

```
// Use exception handling to check for allocation failure.
Shape *generator()
```

```
  try {
    switch(rand() % 4) {
      case 0:
        return new Line;
      case 1:
        return new Rectangle;
      case 2:
        return new Triangle;
      case 3:
        return new NullShape;
    }
  } catch (bad_alloc ba) {
    return NULL;
  }
  return NULL;
}
```

2. Here is **generator( )** recoded to use **new(nothrow)**.

```
// Use new(nothrow).
Shape *generator()
{
  Shape *temp;

  switch(rand() % 4) {
    case 0:
      temp = new(nothrow) Line;
      break;
    case 1:
      temp = new(nothrow) Rectangle;
      break;
    case 2:
      temp = new(nothrow) Triangle;
      break;
    case 3:
      temp = new(nothrow) NullShape;
  }

  if(temp) return temp;
  else return NULL;
}
```

REVIEW SKILLS CHECK: Chapter 13

1. In addition to the C-like cast, the casting operators are

Operator	Purpose
dynamic_cast	Performs a polymorphic cast
reinterpret_cast	Converts one type of pointer into another
static_cast	Performs a "normal" cast
const_cast	Casts away **const**-ness

2. **type_info** is a class that encapsulates information about a data type. A reference to a **type_info** object is returned by the **typeid** operator.
3. **typeid**
4.
```
if(typeid(Derived) == typeid(*p))
  cout << "p points to derived object\n";
else
  cout << "p points to a Base object\n";
```
5. The missing word is "derived."
6. No.

13.1 EXERCISES

1.
```
/* Convert spaces to !s without the
   "using namespace std" statement. */
#include <iostream>
#include <fstream>

int main(int argc, char *argv[])
{
  if(argc!=3) {
    std::cout << "Usage: CONVERT <input> <output>\n";
    return 1;
  }

  std::ifstream fin(argv[1]); // open input file
  std::ofstream fout(argv[2]);  // create output file

  if(!fout) {
```

```
    std::cout << "Cannot open output file.\n";
    return 1;
  }
  if(!fin) {
    std::cout << "Cannot open input file.\n";
    return 1;
  }

  char ch;

  fin.unsetf(std::ios::skipws);  // do not skip spaces
  while(!fin.eof()) {
    fin >> ch;
    if(ch==' ') ch = '|';
    if(!fin.eof()) fout << ch;
  }

  fin.close();
  fout.close();

  return 0;
}
```

2. A unnamed namespace restricts the scope of the identifiers declared within it to its file.
3. This form of **using**,

   using *name::member*;

   brings only the specified member into the current namespace. The form

   using namespace *name*;

   brings the entire namespace into view.
4. Since the entire C++ standard library, including the streams **cin** and **cout**, is declared within the **std** namespace, for convenience most programs have brought the **std** namespace into view. This allows the standard library names to be used directly without qualification by **std::**. For many programs, an alternative would be to simply qualify all references to the standard library with **std::**. Another alternative is to create **using** statements for only **std::cin** or **std::cout**.

5. By putting library code into its own namespace you reduce the possibility of name conflicts.

2 EXERCISES

1.
```
// Convert string type to integer.
#include <iostream>
#include <cstring>
using namespace std;

class strtype {
  char str[80];
  int len;
public:
  strtype(char *s) { strcpy(str, s); len = strlen(s); }
  operator char *() { return str; }
  operator int() { return len; }
};

int main()
{
  strtype s("Conversion functions are convenient.");
  char *p;
  int l;

  l = s; // convert s to integer - which is length of string
  p = s; // convert s to char * - which is pointer to string

  cout << "The string:\n";
  cout << p << "\nis " << l << " chars long.\n";

  return 0;
}
```

2.
```
#include <iostream>
using namespace std;

int p(int base, int exp);

class pwr {
  int base;
  int exp;
public:
  pwr(int b, int e) { base = b; exp = e; }
```

```
  operator int() { return p(base, exp); }
};

// Return base to the exp power.
int p(int base, int exp)
{
  int temp;

  for(temp=1; exp; exp--) temp = temp * base;

  return temp;
}

int main()
{
  pwr o1(2, 3), o2(3, 3);
  int result;

  result = o1;
  cout << result << '\n';

  result = o2;
  cout << result << '\n';

  // can use directly in a cout statement like this:
  cout << o1+100 << '\n';

  return 0;
}
```

13.3 EXERCISES

1.
```
// A shared resource example that traces output.
#include <iostream>
#include <cstring>
using namespace std;

class output {
  static char outbuf[255]; // this is the shared resource
  static int inuse; // buffer available if 0; in use otherwise
  static int oindex; // index of outbuf
  char str[80];
  int i; // index of next char in str
  int who; // identifies the object, must be > 0
```

```
public:
  output(int w, char *s) { strcpy(str, s); i = 0; who = w; }

  /* This function returns -1 if waiting for buffer,
     it returns 0 if it is done outputting, and
     it returns who if it is still using the buffer.
  */
  int putbuf()
  {
    if(!str[i]) { // done outputting
      inuse = 0; // release buffer
      return 0; // signal termination
    }
    if(!inuse) inuse = who; // get buffer
    if(inuse != who) {
      cout << "Process " << who << " Currently blocked\n";
      return -1; // in use by someone else
    }
    if(str[i]) { // still chars to output
      outbuf[oindex] = str[i];
      cout << "Process " << who << " sending char\n";
      i++; oindex++;
      outbuf[oindex] = '\0';  // always keep null-terminated
      return 1;
    }
    return 0;
  }
  void show() { cout << outbuf << '\n'; }
};

char output::outbuf[255]; // this is the shared resource
int output::inuse = 0; // buffer available if 0; in use otherwise
int output::oindex = 0; // index of outbuf

int main()
{
  output o1(1, "This is a test"), o2(2, " of statics");

  while(o1.putbuf() | o2.putbuf()) ; // output chars

  o1.show();

  return 0;
```

2.
```
#include <iostream>
#include <new>
using namespace std;

class test {
  static int count;
public:
  test() { count++; }
  ~test() { count--; }
  int getcount() { return count; }
};

int test::count = 0;

int main()
{
  test o1, o2, o3;

  cout << o1.getcount() << " objects in existence\n";

  test *p;

  /* Watch for allocation errors using both
     old-style and new-style error handling. */
  try {
    p = new test; // allocate an object
    if(!p) { // old-style
      cout << "Allocation error\n";
      return 1;
    }
  } catch(bad_alloc ba) { // new-style
      cout << "Allocation error\n";
      return 1;
  }

  cout << o1.getcount();
  cout << " objects in existence after allocation\n";

  // delete object
  delete p;

  cout << o1.getcount();
  cout << " objects in existence after deletion\n";
```

```
  return 0;
}
```

13.4 EXERCISES

1. To fix the program, simply make **current** mutable so that it can be modified by **counting( )**, which is a **const** member function. The solution is shown here:

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

class CountDown {
  int incr;
  int target;
  mutable int current; // make current mutable
public:
  CountDown(int delay, int i=1) {
    target = delay;
    incr = i;
    current = 0;
  }

  bool counting() const {
    current += incr;
    if(current >= target) {
      cout << "\a";
      return false;
    }
    cout << current << " ";
    return true;
  }
};

int main()
{
  CountDown ob(100, 2);

  while(ob.counting()) ;

  return 0;
}
```

2. No. If it were possible for a **const** member function to call a non-**const** member function, the non-**const** function could be used to modify the invoking object.

13.5 EXERCISES

1. Yes.
2. Yes, because C++ defines an automatic conversion from **int** to **double**.
3. One potential problem with implicit constructor conversions is that you might forget that such a conversion is taking place. For example, an implicit conversion that takes place in an assignment statement looks a lot like an overloaded assignment operator. However, their actions are not necessarily the same. When you are creating classes that others will be using, it might be wise to prevent implicit constructor conversions to avoid misconceptions on the part of the users of your classes.

13.7 EXERCISES

1.
```
/* This version displays the number of chars
   written to buf.
*/
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char buf[255];

  ostrstream ostr(buf, sizeof buf);

  ostr << "Array-based I/O uses streams just like ";
  ostr << "'normal' I/O\n" << 100;
  ostr << ' ' << 123.23 << '\n';

  // you can use manipulators, too
  ostr << hex << 100 << ' ';
  // or format flags
  ostr.setf(ios::scientific);
```

```
    ostr << dec << 123.23;
    ostr << endl << ends; // ensure that buffer is
                          // null-terminated

    // show resultant string
    cout << buf;

    cout << ostr.pcount();

    return 0;
}
```

2.
```
/* Use array-based I/O to copy contents of one
   array into another.
*/
#include <iostream>
#include <strstream>
using namespace std;

char inbuf[] = "This is a test of C++ array-based I/O";
char outbuf[255];

int main()
{
  istrstream istr(inbuf);
  ostrstream ostr(outbuf, sizeof outbuf);

  char ch;

  while(!istr.eof()) {
    istr.get(ch);
    if(!istr.eof()) ostr.put(ch);
  }
  ostr.put('\0'); // null terminate

  cout << "Input: " << inbuf << '\n';
  cout << "Output: " << outbuf << '\n';

  return 0;
}
```

3.
```
// Convert string to float.
#include <iostream>
#include <strstream>
```

```
using namespace std;

int main()
{
  float f;
  char s[] = "1234.564";  // float represented as string

  istrstream istr(s);

  // Convert to internal representation the easy way!
  istr >> f;

  cout << "Converted form: " << f << '\n';

  return 0;
}
```

MASTERY SKILLS CHECK: Chapter 13

1. Unlike a normal member variable, for which each object has its own copy, only one copy of a **static** member variable exists, and it is shared by all objects of that class.
2. To use array-based I/O, include **<strstream>**.
3. No.
4. `extern "C" int counter();`
5. A conversion function simply converts an object into a value compatible with another type. Conversion functions are typically used to convert objects into values compatible with the built-in data types.
6. The **explicit** keyword applies only to constructors. It prevents implicit constructor conversions.
7. A **const** member function cannot modify the object that invokes it.
8. A namespace, which is declared with the **namespace** keyword, defines a declarative region.

9. The keyword **mutable** allows a **const** member function to modify a data member of its class.

CUMULATIVE SKILLS CHECK: Chapter 13

1. Yes. In cases in which the implicit conversion performs the same action that would be performed by an overloaded assignment operator for the type of the constructor's parameter, there is no need to overload the assignment operator.
2. Yes.
3. New libraries can be contained within their own namespaces, thus preventing name conflicts with other code. This benefit applies even to old code that is being updated with new libraries.

REVIEW SKILLS CHECK: Chapter 14

1. Namespaces were added to C++ to localize identifier names in order to prevent name collisions. The problem of name collisions was becoming severe because of the growth of third-party class libraries.
2. To specify a member function as **const**, follow the function's parameter list with the keyword **const**. Here's an example:

```
int f(int a) const;
```

3. False. **mutable** allows a member variable to be changed by a **const** member function.
4.
```
class X {
  int a, b;
public:
  X(int i, int j) { a = i, b = j; }
  operator int() { return a+b; }
};
```

5. True.
6. No. The **explicit** specifier prevents the automatic conversion from **int** to **Demo**.

14.1 EXERCISES

1. A container is an object that holds other objects. An algorithm is a routine that acts upon the contents of containers. An iterator is similar to a pointer.
2. Binary and unary.
3. The five types of iterators are random access, bidirectional, forward, input, and output.

14.3 EXERCISES

2. Any object stored in a vector must provide a default constructor.
3.
```
// Store Coord objects in a vector.
#include <iostream>
#include <vector>
using namespace std;

class Coord {
public:
  int x, y;
  Coord() { x = y = 0; }
  Coord(int a, int b) { x = a; y = b; }
};

bool operator<(Coord a, Coord b)
{
  return (a.x+a.y) < (b.x+b.y);
}

bool operator==(Coord a, Coord b)
{
  return (a.x+a.y) == (b.x+b.y);
}

int main()
{
  vector<Coord> v;
  int i;

  for(i=0; i<10; i++)
    v.push_back(Coord(i, i));
```

```
for(i=0; i<v.size(); i++)
  cout << v[i].x << "," << v[i].y << " ";

cout << endl;

for(i=0; i<v.size(); i++)
  v[i].x = v[i].x * 2;

for(i=0; i<v.size(); i++)
  cout << v[i].x << "," << v[i].y << " ";

return 0;
```

14.4 EXERCISES

2.
```
// List basics.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> lst; // create an empty list
  int i;

  for(i=0; i<10; i++) lst.push_back('A'+i);

  cout << "Size = " << lst.size() << endl;

  list<char>::iterator p;

  cout << "Contents: ";
  for(i=0; i<lst.size(); i++) {
    p = lst.begin();
    cout << *p;
    lst.pop_front();
    lst.push_back(*p); // put element on end of list
  }

  cout << endl;

  if(!lst.empty())
```

```
  cout << "List is not empty.\n";

  return 0;
}
```

This program displays the following output.

```
Size = 10
Contents: ABCDEFGHIJ
List is not empty.
```

In this approach, elements are removed from the front but put on the back. Thus, the list is not emptied. The loop that displays the list is controlled by obtaining the size of the list using **size( )**.

3.
```
// Merge two projects lists.
#include <iostream>
#include <list>
#include <cstring>
using namespace std;

class Project {
public:
  char name[40];
  int days_to_completion;
  Project() {
    strcpy(name, "");
    days_to_completion = 0;
  }
  Project(char *n, int d) {
    strcpy(name, n);
    days_to_completion = d;
  }

  void add_days(int i) {
    days_to_completion += i;
  }

  void sub_days(int i) {
    days_to_completion -= i;
  }

  bool completed() { return !days_to_completion; }

  void report() {
    cout << name << ": ";
```

```
    cout << days_to_completion;
    cout << " days left.\n";
  }
};

bool operator<(const Project &a, const Project &b)
{
  return a.days_to_completion < b.days_to_completion;
}

bool operator>(const Project &a, const Project &b)
{
  return a.days_to_completion > b.days_to_completion;
}

bool operator==(const Project &a, const Project &b)
{
  return a.days_to_completion == b.days_to_completion;
}

bool operator!=(const Project &a, const Project &b)
{
  return a.days_to_completion != b.days_to_completion;
}

int main()
{
  list<Project> proj;
  list<Project> proj2;

  proj.push_back(Project("Compiler", 35));
  proj.push_back(Project("Spreadsheet", 190));
  proj.push_back(Project("STL Implementation", 1000));

  proj2.push_back(Project("Database", 780));
  proj2.push_back(Project("Mail Merge", 50));
  proj2.push_back(Project("COM Objects", 300));

  proj.sort();
  proj2.sort();

  proj.merge(proj2); // merge projects

  list<Project>::iterator p = proj.begin();
```

```
  /* display projects */
  while(p != proj.end()) {
    p->report();
    p++;
  }

  return 0;
}
```

14.5 EXERCISES

2.
```
// A map of names and phone numbers.
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class name {
  char str[20];
public:
  name() { strcpy(str, ""); }
  name(char *s) { strcpy(str, s); }
  char *get() { return str; }
};

// must define less than relative to name objects
bool operator<(name a, name b)
{
   return strcmp(a.get(), b.get()) < 0;
}

class phonenum {
  char str[20];
public:
  phonenum() { strcmp(str, ""); }
  phonenum(char *s) { strcpy(str, s); }
  char *get() { return str; }
};


int main()
{
  map<name, phonenum> m;
```

```
  // put names and phone numbers into map
  m.insert(pair<name,
    phonenum>(name("Joe"), phonenum("555-4323")));
  m.insert(pair<name,
    phonenum>(name("Tom"), phonenum("555-9784")));
  m.insert(pair<name,
    phonenum>(name("Larry"), phonenum("212 555-9372")));
  m.insert(pair<name,
    phonenum>(name("Tod"), phonenum("01 11 232-4232")));

  // given a name, find phone number
  char str[80];
  cout << "Enter name: ";
  cin >> str;

  map<name, phonenum>::iterator p;

  p = m.find(name(str));
  if(p != m.end())
    cout << "Phone Number: " <<  p->second.get();
  else
    cout << "Name not in map.\n";

  return 0;
}
```

3. Yes.

14.6 EXERCISES

1.
```
// Sort a vector using the sort algorithm.
#include <iostream>
#include <vector>
#include <cstdlib>
#include <algorithm>
using namespace std;

int main()
{
  vector<char> v;
  int i;

  // create a vector of random characters
  for(i=0; i<10; i++)
```

```
    v.push_back('A'+ (rand()%26));

  cout << "Original contents: ";
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";

  cout << endl << endl;

  // sort the vector
  sort(v.begin(), v.end());

  cout << "Sorted contents: ";
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";

  return 0;
}
```

2.
```
// Merge two lists using the merge algorithm.
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
  list<char> lst1, lst2, lst3(20);
  int i;

  for(i=0; i<10; i+=2) lst1.push_back('A'+i);
  for(i=1; i<11; i+=2) lst2.push_back('A'+i);

  cout << "Contents of lst1: ";
  list<char>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  cout << "Contents of lst2: ";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p;
    p++;
```

```
  cout << endl << endl;

  // now, merge the two lists
  merge(lst1.begin(), lst1.end(),
        lst2.begin(), lst2.end(),
        lst3.begin());

  cout << "Merged list:\n";
  p = lst3.begin();
  while(p != lst3.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

14.7 EXERCISES

1.
```
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{
  list<string> str;

  str.push_back(string("one"));
  str.push_back(string("two"));
  str.push_back(string("three"));
  str.push_back(string("four"));
  str.push_back(string("five"));
  str.push_back(string("six"));
  str.push_back(string("seven"));
  str.push_back(string("eight"));
  str.push_back(string("nine"));
  str.push_back(string("ten"));

  str.sort(); // sort the vector

  list<string>::iterator p = str.begin();
  while(p != str.end()) {
```

```
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

2.
```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
  string str;

  cout << "Enter a string: ";
  cin >> str;

  int i = count(str.begin(),'str.end(), 'e');

  cout << i << " of them are e.\n";

  return 0;
}
```

3.
```
#include <iostream>
#include <string>
#include <algorithm>
#include <cctype>
using namespace std;

int main()
{
  string str;

  cout << "Enter a string: ";
  cin >> str;

  int i = count_if(str.begin(), str.end(), islower);

  cout << i << " of them are lowercase.\n";

  return 0;
}
```

4. **string** is a specialization of **basic_string**.

MASTERY SKILLS CHECK: Chapter 14

1. The STL provides off-the-shelf, debugged versions of many common data structures and algorithms. Because the STL classes are templatized, they can be used with any type of data.
2. A container is an object that holds other objects. An iterator is similar to a pointer. An algorithm acts on the contents of containers.
3.
```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

int main()
{
  vector<int> v(10);
  list<int> lst;
  int i;

  for(i=0; i<10; i++) v[i] = i;

  for(i=0; i<10; i++)
    if(!(v[i]%2)) lst.push_back(v[i]);

  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```
4. The **string** type allows strings to be manipulated with operators. However, working with **string** objects is not as efficient as working with null-terminated character arrays.
5. A predicate is a function that returns either true or false.


C++