Listing 2.40 Returning class objects by reference

```
/*Beginning of Distance.h*/
/*Header file containing the definition of the Distance
class*/
class Distance
{
/*definition of the class Distance*/
};
Distance& larger(Distance&, Distance&);
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
Distance& larger(Distance& dd1, Distance& dd2)
{
   float i,j;
   i=dd1.getFeet()*12+dd1.getInches();
   j=dd2.getFeet()*12+dd2.getInches();
   if(i>j)
      return dd1;
   else
      return dd2;
}
/*
definitions of the rest of the functions of class Distance
*/
/*End of Distance.cpp*/

/*Beginning of Distmain.cpp*/
#include<iostream.h>
#include"Distance.h"
void main()
{
   Distance d1,d2;
   d1.setFeet(5);
   d1.setInches(7.5);
   d2.setFeet(5);
   d2.setInches(6.25);
   Distance& d3=larger(d1,d2);
   d3.setFeet(0);
   d3.setInches(0.0);
   cout<<d1.getFeet()<<" "<<d1.getInches()<<endl;
   cout<<d2.getFeet()<<" "<<d2.getInches()<<endl;
}
/*End of Distmain.cpp*/
```

Output

```
0 0.0
5 6.25
```

2.4 Objects and Arrays

Let us understand how arrays of objects and arrays inside objects are handled in C++.

2.4.1 Arrays of Objects

We can create arrays of objects. The following program shows how.

Listing 2.41 Array of objects

```
/*Beginning of DistArray.cpp*/
#include"Distance.h"
#include<iostream.h>
#define SIZE 3

void main()
{
   Distance dArray[SIZE];
   int a;
   float b;
   for(int i=0;i<SIZE;i++)
   {
      cout<<"Enter the feet : ";
      cin>>a;
      dArray[i].setFeet(a);
      cout<<"Enter the inches : ";
      cin>>b;
      dArray[i].setInches(b);
   }
   for(int i=0;i<SIZE;i++)
   {
         cout <<dArray[i].getFeet()<<" "
              <<dArray[i].getInches()<<endl;
   }
}

/*End of DistArray.cpp*/
```

Output

```
Enter the feet : 1<enter>
Enter the inches : 1.1<enter>
Enter the feet : 2<enter>
Enter the inches : 2.2<enter>
Enter the feet : 3<enter>
Enter the inches : 3.3<enter>
1 1.1
2 2.2
3 3.3
```

2.4.2 Arrays Inside Objects

An array can be declared inside a class. Such an array becomes a member of all objects of the class. It can be manipulated/accessed by all member functions of the class. The class definition shown in Listing 2.42 illustrates this.

Listing 2.42 Arrays inside objects

```
#define SIZE 3
/*A class to duplicate the behaviour of an integer array*/
class A
{
      int iArray[SIZE];
   public:
      void setElement(unsigned int,int);
      int getElement(unsigned int);
};
/*function to write the value passed as second parameter at the position passed
as first parameter*/
void A::setElement(unsigned int p,int v)
{
   if(p>=SIZE)
      return;            //better to throw an exception
   iArray[p]=v;
}
/*function to read the value from the position passed as parameter*/
int A::getElement(unsigned int p)
{
   if(p>=SIZE)
      return -1;         //better to throw an exception
   return iArray[p];
}
```

The class definition is self-explanatory. However, the comments indicate that it is better to throw exceptions rather than terminate the function. What are exceptions? How are they thrown? What are the benefits of using them? All these questions are answered in the chapter on Exception Handling.

2.5 Namespaces

Namespaces enable the C++ programmer to prevent pollution of the global namespace that leads to name clashes.

The term 'global namespace' refers to the entire source code. It also includes all the directly and indirectly included header files. By default, the name of each class is visible in the entire source code, that is, in the global namespace. This can lead to problems.

Suppose a class with the same name is defined in two header files.

```
/*Beginning of A1.h*/
class A
{
};
/*End of A1.h*/

/*Beginning of A2.h*/
class A //a class with an existing name
{
};
/*End of A2.h*/
```

Now, let us include both these header files in a program and see what happens if we declare an object of the class. See Listing 2.43.

Listing 2.43 Referring to a globally declared class can lead to ambiguity error

```
/*Beginning of multiDef01.cpp*/
#include"A1.h"
#include"A2.h"
void main()
{
  A AObj;                         //ERROR: Ambiguity error due to multiple
                                  //definitions of A
}
/*End of multiDef01.cpp*/
```

The scenario in Listing 2.43 is quite likely in large programs. The global visibility of the definition of class A makes the inclusion of the two header files mutually exclusive. Consequently, this also makes use of the two definitions of class A *mutually exclusive*.

How can this problem be overcome? How can we ensure that an application is able to use both definitions of class A simultaneously? Enclosing the two definitions of the class in separate namespaces overcomes this problem.

```
/*Beginning of A1.h*/
namespace A1                      //beginning of a namespace A1
{
  class A
  {
  };
}                                 //end of a namespace A1
/*End of A1.h*/

/*Beginning of A2.h*/
namespace A2                      //beginning of a namespace A2
{
  class A
  {
  };
}                                 //end of a namespace A2
/*End of A2.h*/
```

Now, the two definitions of the class are enveloped in two different namespaces. The corresponding namespace, followed by the scope resolution operator, must be prefixed to the name of the class while referring to it anywhere in the source code. Thus, the ambiguity encountered in the above listing can be overcome. A revised definition of the `main()` function from Listing 2.43 illustrates this (Listing 2.44).

Listing 2.44 Enclosing classes in namespaces prevents pollution of the global namespace

```
/*Beginning of multiDef02.cpp*/
#include"A1.h"
#include"A2.h"
void main()
{
  A1::A AObj1;                    //OK: AObj1 is an object of the class
                                  //defined in A1.h
  A2::A AObj2;                    //OK: AObj2 is an object of the class
                                  //defined in A2.h
}
/*End of multiDef02.cpp*/
```

Qualifying the name of the class with that of the namespace can be cumbersome. The `using` directive enables us to make the class definition inside a namespace visible so that qualifying the name of the referred class by the name of the namespace is no longer required. Listing 2.45 shows how this is done.

Listing 2.45 The `using` directive makes qualifying of referred class names by names of enclosing namespaces unnecessary

```
/*Beginning of using.cpp*/
#include"A1.h"
#include"A2.h"
void main()
{
   using namespace A1;
   A AObj1;                          //OK: AObj1 is an object of the class
                                     //defined in A1.h
A2::A AObj2;                         //OK: AObj2 is an object of the class
                                     //defined in A2.h
}
/*Beginning of using.cpp*/
```

However, we must note that the `using` directive brings back the global namespace pollution that the namespaces mechanism was supposed to remove in the first place! The last line in the above listing compiles only because the class name was qualified by the name of the namespace.

Some namespaces have long names. Qualifying the name of a class that is enclosed within such a namespace, with the name of the namespace, is cumbersome. See Listing 2.46.

Listing 2.46 Cumbersome long names for namespace

```
/*Beginning of longName01.cpp*/
namespace a_very_very_long_name
{
   class A
   {
   };
}

void main()
{
   a_very_very_long_name::A A1;    //cumbersome long name
}
/*End of longName01.cpp*/
```

Assigning a suitably short alias to such a long namespace name solves the problem as illustrated in Listing 2.47.

Listing 2.47 Providing an alias for a namespace

```
/*Beginning of longName02.cpp*/
namespace a_very_very_long_name
{
   class A
   {
   };
}
namespace x = a_very_very_long_name;   //declaring an
                                       //alias
void main()
{
   x::A A1;                        //convenient short name
}
/*End of longName02.cpp*/
```

Aliases provide an incidental benefit also. Suppose an alias has been used at a number of places in the source code. Changing the alias declaration so that it stands as an alias for a different namespace will make each reference of the enclosed class refer to a completely different class. Suppose an alias X refers to a namespace 'N1'.

```
namespace X = N1;                  //declaring an alias
```

Further, suppose that this alias has been used extensively in the source code.

```
X::A AObj;       //AObj is an object of class A that is
                 //enclosed in namespace N1.
AObj.f1();       //f1() is a member function of the above
                 //class.
```

If the declaration of alias X is modified as follows ('N2' is also a namespace)

```
namespace X = N2;                  //modifying the alias
```

then, all existing qualifications of referred class names that use X would now refer to class A that is contained in namespace 'N2'. Of course, the lines having such references would compile only if *both* of the namespaces, 'N1' and 'N2', contain a class named A, and if these two classes have the *same* interface.

For keeping the explanations simple, classes that have been given as examples in the rest of this book are not enclosed in namespaces.

2.6 Nested Inner Classes

A class can be defined inside another class. Such a class is known as a *nested class*. The class that contains the nested class is known as the *enclosing class*. Nested classes can be defined in the private, protected, or public portions of the enclosing class (protected access specifier is explained in the chapter on inheritance).

In Listing 2.48, class B is defined in the private section of class A.

Listing 2.48 Nested classes

```
/*Beginning of nestPrivate.h*/
class A
{
   class B
   {
      /*
         definition of class B
      */
   };
   /*
      definition of class A
   */
};
/*End of nestPrivate.h*/
```

In Listing 2.49, class B is defined in the public section of class A.

Listing 2.49 A public nested class

```
/*Beginning of nestPublic.h*/
class A
{
  public:
  class B
  {
    /*
      definition of class B
    */
  };
  /*
    definition of class A
  */
};
/*End of nestPublic.h*/
```

A nested class is created if it does not have any relevance outside its enclosing class. By defining the class as a nested class, we avoid a *name collision*. In Listings 2.48 and 2.49, even if there is a class B defined as a global class, its name will *not* clash with the nested class B.

The size of objects of an enclosing class is not affected by the presence of nested classes. See Listing 2.50.

Listing 2.50 Size of objects of the enclosing class

```
/*Beginning of nestSize.cpp*/
#include<iostream.h>

class A
{
    int x;
  public:
    class B
    {
                           int y;
    };
};

void main()
{
  cout<<sizeof(int)<<endl;
  cout<<sizeof(A)<<endl;
}
/*End of nestSize.cpp*/
```

Output

```
4
4
```

How are the member functions of a nested class defined? Member functions of a nested class can be defined outside the definition of the enclosing class. This is done by prefixing the function name with the name of the enclosing class followed by the scope resolution operator. This, in turn, is followed by the name of the nested class followed again by the scope resolution operator. This is illustrated by Listing 2.51.

Listing 2.51 Defining member functions of nested classes

```
/*Beginning of nestClassDef.h*/
class A
{
  public:
  class B
  {
    public:
      void BTest();              //prototype only
  };
  /*
    definition of class A
  */
};
/*End of nestClassDef.h*/

/*Beginning of nestClassDef.cpp*/
#include"nestClassDef.h"
void A::B::BTest()
{
  //definition of A::B::BTest() function
}

/*
  definitions of the rest of the functions of class B
*/
/*End of nestClassDef.cpp*/
```

A nested class may be only prototyped within its enclosing class and defined later. Again, the name of the enclosing class followed by the scope resolution operator is required. See Listing 2.52.

Listing 2.52 Defining a nested class outside the enclosing class

```
/*Beginning of nestClassDef.h*/
class A
{
  class B;                       //prototype only
};

class A::B
{
  /*
    definition of the class B
  */
};
/*End of nestClassDef.h*/
```

Objects of the nested class are defined outside the member functions of the enclosing class in much the same way (by using the name of the enclosing class followed by the scope resolution operator).

```
A::B B1;
```

However, the above line will compile only if class B is defined within the `public` section of class A. Otherwise, a compile-time error will result.

An object of the nested class can be used in any of the member functions of the enclosing class without the scope resolution operator. Moreover, an object of the nested class can be a member of the enclosing class. In either case, only the public members of the object can be accessed unless the enclosing class is a friend of the nested class. See Listing 2.53.

Listing 2.53 Declaring objects of the nested class in the member functions of the enclosing class

```
/*Beginning of nestClassObj.h*/
class A
{
    class B
    {
      public:
                                  void BTest();     //prototype only
    };
    B B1;
  public:
    void ATest();
};
/*End of nestClassObj.h*/

/*Beginning of nestClassObj.cpp*/
#include"nestClassObj.h"

void A::ATest()
{
  B1.BTest();
  B B2;
  B2.BTest();
}
/*End of nestClassObj.cpp*/
```

Member functions of the nested class can access the non-static public members of the enclosing class through an object, a pointer, or a reference only. An illustrative example follows in Listing 2.54.

Listing 2.54 Accessing non-static members of the enclosing class in member functions of the nested class.

```
/*Beginning of enclClassObj.h*/
class A
{
  public:
    void ATest();
    class B
    {
      public:
                                  void BTest(A&);
                                  void BTest1();
    };
};
/*End of enclClassObj.h*/
```

```
/*Beginning of enclClassObj.cpp*/
#include"enclClassObj.h"

void A::B::BTest(A& ARef)
{
  ARef.ATest();                    //OK
}

void A::B::BTest1()
{
  ATest();                         //ERROR!!
}
/*End of enclClassObj.cpp*/
```

It can be observed that an error is produced when a direct access is made to a member of the enclosing class through a function of the nested class. This is as it should be. After all, *creation of an object of the nested class does not cause an object of the enclosing class to be created*. The classes are nested to merely control the visibility. Since 'A::B::BTest()' function will be called with respect to an object of class B, a direct access to a member of the enclosing class A can be made through an object of that class only.

By default, the enclosing class and the nested class do not have any access rights to each other's private data members. They can do so only if they are friends to each other.

Summary

Classes have both member data and member functions. Member functions can be given *exclusive rights* to access data members. Member functions and member data can be private, protected, or public. The `struct` keyword has been redefined in C++. Apart from member data, structures in C++ can have member functions also. In a class, members are *private* by default. In a structure, members are *public* by default.

The scope resolution operator is used to separate the class definition from the definitions of the member functions. The class definition can be placed in a header file. Member functions, with the aid of scope resolution operator, can be placed in a separate implementation file.

The `this` pointer is implicitly inserted by the compiler, as a leading formal argument, in the prototype and in the definition of each member function of each class. When a member function is called with respect to an object, the compiler inserts the address of the calling object as a leading parameter to the function call. Consequently, the `this` pointer, which exists as the implicit leading formal argument in all member functions, always points at the object with respect to which the member function has been called.

Access to member data and member functions from within member functions is resolved by the `this` pointer. The `this` pointer is a *constant pointer in case of non-constant member functions* and a *constant pointer to a constant in case of constant member functions*.

If the operand on its right is a data member, then the object-to-member access operator (.) behaves just as it does in C language. However, if it is a member function of a class whereas an object of the same class is its left-hand side operand, then the compiler simply passes the *address of the object* as an implicit leading parameter to the function call.

Similarly, if the operand on its right is a data member, then the pointer-to-member access operator (->) behaves just as it does in C language. However, if it is a member function of a class whereas a pointer to an object of the same class is its left-hand side operand, then the compiler simply passes *the value of the pointer* as an implicit leading parameter to the function call. Member functions can call each other. Calls are resolved through the `this` pointer. Member functions can be overloaded. Default values can be given to the formal arguments of member functions.

Programs having inline functions tend to run faster than equivalent programs with non-inline functions. A function is declared inline either by defining it inside a class or by declaring it inside a class and defining it outside with the keyword `inline`. This feature should be used sparingly. Otherwise, the increased size of the executable can slow it down.

If required, member functions can be declared as constant functions to prevent even an inadvertent change in the data members. A function can be declared as a constant function by suffixing its prototype and the header of its definition by the keyword `const`.

A mutable data member is never constant. It is modifiable inside constant functions also. A friend function is a non-member function that has a special right to access private data members of objects of the class of which it is a friend. This does not really negate the philosophy of OOPS. A friend function still needs to be declared inside the class of which it is a friend. The advantage that a friend function provides is that it is not called with respect to an object.

A global non-member function can be declared as a friend to a class. Member function of one class can be declared as a friend function of another. An entire class can be declared as a friend of another too. A class or a function is declared friend to a desired class by prototyping it in the class and prefixing the prototype with the keyword `friend`.

Only *one* copy of a static data member exists for the entire class. This is in contrast to non-static data members that exist separately in *each* object. Static data members are used to keep data that relates to the entire set of objects that exist at any given point during the program's execution. A data member is declared as a static member of a class by prefixing its declaration in the class by the keyword `static`.

Static member functions can access *static data members only*. They can be called without declaring any objects. A member function is declared as a static member of a class by prefixing its declaration in the class by the keyword `static`.

Objects can appear as local variables inside functions. They can also be passed by value or by reference to functions. Finally, they can be returned by value or by reference from functions.

Arrays of objects can be created. Arrays can be created inside classes also. One class can be defined inside another class. Such a class is known as a *nested class*. The class that contains the nested class is known as the *enclosing class*. Nested classes can be defined in the private, protected, or public portions of the enclosing class.

Namespaces enable the C++ programmer to prevent pollution of the global namespace. They help prevent name classes.

Key Terms

class
private access specifier
public access specifier
objects
scope resolution operator
the `this` pointer
data abstraction
arrow operator
overloaded member functions
default values for formal arguments of member functions
inline member functions
constant member functions
mutable data members
friend non-member functions
friend classes
friend member functions
friends as bridges
static member data
static member functions
namespaces
nested classes

Exercises

1. How does the class construct enable data security?
2. What is the use of the scope resolution operator?
3. What is the `this` pointer? Where and why does the compiler insert it implicitly?
4. What is data abstraction? How is it implemented in C++?
5. Which operator is used to access a class member with respect to a pointer?

6. What is the difference between a mutable data member and a static data member?
7. Describe the two ways in which a member function can be declared as an inline function.
8. How can a global non-member function be declared as a friend to a class?
9. What is the use of declaring a class as a friend of another?
10. Explain why friend functions do not contradict the principles of OOPS.
11. Explain why static data members should be explicitly declared outside the class.
12. Why should static data members be defined in the implementation files only?
13. What is the use of static member functions?
14. How do namespaces help in preventing pollution of the global namespace?
15. What is a nested class? What is its use?
16. How are the member functions of a nested class defined outside the definition of the enclosing class?
17. State true or false.
    (a) Structures in C++ can have member functions also.
    (b) Structure members are private by default.
    (c) The `this` pointer is always a constant pointer.
    (d) Member functions cannot be overloaded.
    (e) Default values can be given to the formal arguments of member functions.
    (f) Only constant member function can be called for constant objects.
    (h) The keyword `friend` should appear in the prototype as well as the definition of the function that is being declared as a friend.
    (i) A friend function can be prototyped in only the public section of the class.
    (j) Friendship is not transitive.
    (k) A static data member can be of the same type as the class of which it is a member.
    (l) The size of objects of an enclosing class is affected by the presence of nested classes.
    (m) An object of the nested class can be used in any of the member functions of the enclosing class without the scope resolution operator.
    (n) An object of the nested class cannot be a member of the enclosing class.
    (o) Public members of the nested class's object which have been declared in a function of the enclosing class can always be accessed.
18. Your compiler should provide a structure and associated functions to fetch the current system date. Suppose the name of the structure is `date_d` and the name of the associated functions to fetch the current system date is `getSysDate()`.

    Create a class with a name that is similar to the above structure. This class should contain a variable of the above structure as its private data member. Introduce a member function in the class that calls the associated function of the date structure. Thus, create a wrapper class and make an available structure safe to use.

```
class date_D    //a wrapper class
{
      date_d d;
   public:
      void getSysDate();
};

void date_D::getSysDate()
{
getSysDate(&d); //calling the associ-
                   ated function from
                //the member function
}
```

    Also, write a small test program to test the above class.
19. Create a class named `Distance_mks`. This class should be similar to the class `Distance`, except for the following differences:
    - The data members of this new class would be `iMeters` (type integer; for representing the meters portion of a distance) and `fCentimeters` (type float; for representing the centimeters portion of a distance) instead of `iFeet` and `fInches`.
    - Suitably designed member functions to work upon the new data members should replace the ones that we have seen for the class `Distance`. The member functions should ensure that the `fCentimeters` of no object should ever exceed 100.

4 Constructors and Destructors

OVERVIEW

We are already aware of the need to include a member function in our class that initializes the data members of its class to desired default values and gets called automatically for each object that has just got created. Constructors fulfill this need and the first portion of this chapter deals with constructors. Various types of constructors are described in the middle portion of this chapter.

There is also the need to include a member function in our class that gets called automatically for each object that is going out of scope. Destructors fulfill this need and the penultimate portion of this chapter deals with destructors.

Along with the class construct and the access specifiers, constructors and destructors complete the requirements needed to created new data type—safe and efficient data types. This is discussed in the last portion of this chapter.

4.1 Constructors

The constructor gets called automatically for each object that has just got created. It appears as member function of each class, whether it is defined or not. It has the same name as that of the class. It may or may not take parameters. It does not return anything (not even void). The prototype of a constructor is

```
<class name> (<parameter list>);
```

The need for a function that guarantees initialization of member data of a class was felt in Chapter 2. Constructors fulfill this need. Domain constraints on the values of data members can also be implemented via constructors. For example, we want the value of data member `finches` of each object of the class `Distance` to be between 0.0 and 12.0 at all times within the lifetime of the object. But this condition may get violated in case an object has just got created. However, introducing a suitable constructor to the class `Distance` can enforce this condition.

The compiler embeds a call to the constructor for each object when it is created. Suppose a class A has been declared as follows:

```
/*Beginning of A.h*/
class A
{
    int x;
  public:
    void setx(const int=0);
    int getx();
};
/*End of A.h*/
```

Consider the statement that declares an object of a class A in Listing 4.1.

Listing 4.1 Constructor gets called automatically for each object when it is created

```
/*Beginning of AMain.cpp*/
#include"A.h"
void main()
{
  A A1;                           //object declared … constructor called
}
/*End of AMain.cpp*/
```

The statement in the function `main()` in Listing 4.1 is transformed into the following statements.

```
A A1;           //memory allocated for the object (4 bytes)
A1.A();         //constructor called implicitly by compiler
```

The second statement above is then transformed to

```
A(&A1); //see Chapter 2
```

Similarly, the constructor is called for each object that is created dynamically in the heap by the `new` operator.

```
A * APtr;
APtr = new A;   //constructor called implicitly by compiler
```

The second statement above is transformed into the following two statements.

```
APtr = new A;   //memory allocated
APtr->A();      //constructor called implicitly by compiler
```

The second statement above is then transformed into

```
A(APtr);        //see Chapter 2
```

The foregoing explanations make one thing very clear. Unlike their name, constructors do not actually allocate memory for objects. They are member functions that are called for each object immediately after memory has been allocated for the object.

The constructor is called in this manner separately for each object that is created. But did we prototype and define a public function with the name 'A()' inside the class A? The answer is 'no'. Then how did the above function call get resolved? The compiler prototypes and defines the constructor for us. But what statements does the definition of such a constructor have? The answer is 'nothing'.

Before

```
class A
{
     . . . .
     . . . .
  public:
     . . . .
     . . . .
     //no constructor
};
```

After

```
class A
{
     . . . .
     . . . .
  public:
     A();      //prototype inserted implicitly by compiler
     . . . .
     . . . .
};

A::A()
{
  //empty definition inserted implicitly by compiler
}
```

As we can see, the name of the constructor is the same as the name of the class. Also, the constructor does not return anything. The compiler defines the constructor in order to resolve the call to the constructor that it compulsorily places for the object being created.

For reasons that we will discuss later, it is forbidden to call the constructor explicitly for an existing object as follows.

```
A1.A(); //not legal C++ code!
```

4.1.1 Zero-argument Constructor

We can and should define our own constructors if the need arises. If we do so, the compiler does not define the constructor. However, it still embeds implicit calls to the constructor as before.

The constructor is a non-static member function. It is called for an object. It, therefore, takes the `this` pointer as a leading formal argument just like other non-static member functions. Correspondingly, the address of the invoking object is passed as a leading parameter to the constructor call. This means that the members of the invoking object can be accessed from within the definition of the constructor.

Let us add our own constructor to class A defined in Listing 4.1 and verify whether the constructor is actually called implicitly by the compiler or not. See Listing 4.2.

Listing 4.2 Constructor gets called for each object when the object is created

```
/*Beginning of A.h*/
class A
{
  int x;
  public:
  A();                               //our own constructor
  void setx(const int=0);
  int getx();
};
/*End of A.h*/

/*Beginning of A.cpp*/
#include"A.h"
#include<iostream.h>
A::A()                               //our own constructor
```

```
{
   cout<<"Constructor of class A called\n";
}
/*
definitions of the rest of the functions of class A
*/
/*End of A.cpp*/

/*Beginning of AMain.cpp*/
#include<iostream.h>
#include"A.h"
void main()
{
   A A1;
   cout<<"End of program\n";
}
/*End of AMain.cpp*/
```

Output

```
Constructor of class A called
End of program
```

Let us now define our own constructor for the class `Distance`. What should the constructor do to the invoking object? We would like it to set the values of the `iFeet` and `fInches` data members of the invoking object to 0 and 0.0, respectively. Accordingly, let us add the prototype of the function within the class definition in the header file and its definition in the library source code. See Listing 4.3.

Listing 4.3 A user-defined constructor to implement domain constraints on the data members of a class

```
/*Beginning of Distance.h*/
class Distance
{
   public:
      Distance();                    //our own constructor
      /*
         rest of the class Distance
      */
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include"Distance.h"
Distance::Distance()                 //our own constructor
{
   iFeet=0;
   fInches=0.0;
}
/*
   definitions of the rest of the functions of class
   Distance
*/
/*End of Distance.cpp*/

/*Beginning of DistTest.cpp*/
#include<iostream.h>
```

```
#include"Distance.h"
void main()
{
  Distance d1;                   //constructor called
  cout<<d1.getFeet()<<" "<<d1.getInches();
}
/*End of DistTest.cpp*/
```

Output

0 0.0

Now, due to the presence of the constructor within the class `Distance`, there is a guaranteed initialization of the data of all objects of the class `Distance`. Our objective of keeping the `fInches` portion of all objects of the class `Distance` within 12.0 is now fulfilled.

The constructor that we have defined in Listing 4.2 does not take any arguments and is called the zero-argument constructor. The constructor provided by default by the compiler also does not take any arguments. Therefore, the terms 'zero-argument constructor' and 'default constructor' are used interchangeably.

Now, let us start the study of a class that will enable us to abstract character arrays and overcome many of the drawbacks that exist in them. This class will be our running example for explaining most of the concepts of this book. We will define it incrementally. Our purpose is to ultimately define a class that can be used instead of character arrays.

Let us call the class `String`. It will have two data members. Both these data members will be private. The first data member will be a character pointer. It will point at a dynamically allocated block of memory that contains the actual character array. The other data member will be a long unsigned integer that will contain the length of this character array.

```
/*Beginning of String.h*/
class String
{
     char * cStr;              //character pointer to point at
                               //the character array

     long unsigned int len;    //to hold the length of the
                               //character array

     /*
       rest of the class String
     */

};
/*End of String.h*/
```

Suppose 's1' is an object of the class `String` and the string 'abc' has been assigned to it. Diagrammatically this situation can be depicted in Figure 4.1.

The address of the first byte of the memory block containing the string is 101. This value is stored in the 'cStr' portion of 's1'. The address of 's1' is 27.

Also, we would religiously implement the following two conditions on all objects of the class `String`.

- 'cStr' should either point at a dynamically allocated block of memory exclusively allocated for it (that is, no other pointer should point at the block of memory being pointed at by 'cStr') or 'cStr' should be NULL.
- There should be no memory leaks.

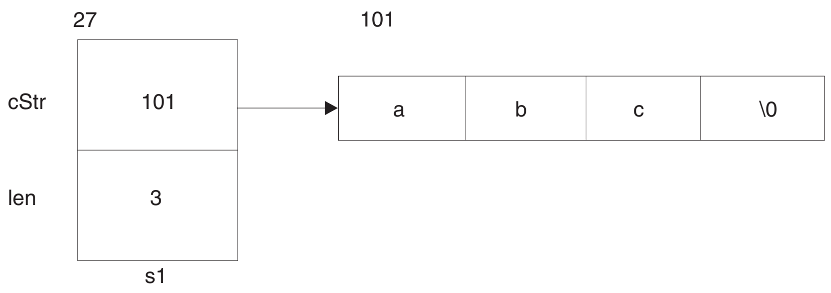


Figure 4.1 Memory layout of an object of the class `String`

Obviously, when an object of the class `String` is created, the 'cStr' portion of the object should be initially set to NULL (and 'len' should be set to 0). Accordingly, the prototype and the definition of the constructor are as shown in Listing 4.4.

Listing 4.4 A user-defined constructor

```
/*Beginning of String.h*/
class String
{
    char * cStr;
    long unsigned int len;
  public:
    String();                   //prototype of the constructor
    /*
      rest of the class String
    */
};
/*End of String.h*/

/*Beginning of String.cpp*/
#include"String.h"
String::String()                //definition of the constructor
{                               //When an object is created …
  cStr=NULL;                    //…nullify its pointer and…
  len=0;                        //…set the length as zero.
}
/*
  definitions of the rest of the functions of class String
*/
/*End of String.cpp*/
```

4.1.2 Parameterized Constructors

Constructors take arguments and can, therefore, be overloaded. Suppose, for the class `Distance`, the library programmer decides that while creating an object, the application programmer should be able to pass some initial values for the data members contained in the object. For this, he/she can create a parameterized constructor as shown in Listing 4.5.

Listing 4.5 A user-defined parameterized constructor—called by creating an object in the stack

```
/*Beginning of Distance.h*/
class Distance
{
  public:
    Distance();                //prototypes provided by the
                               //library programmer
    Distance(int,float);       //prototype of the parameterized
                               //constructor
    /*
      rest of the class Distance
    */
};
/*End of Distance.h*/

/*Beginning of Distance.cpp*/
#include“Distance.h”
Distance::Distance()
{
  iFeet=0;
  fInches=0.0;
}
Distance::Distance(int p, float q)
{
  iFeet=p;
  setInches(q);
}

/*
  definitions of the rest of the functions of class
Distance
*/
/*End of Distance.cpp*/

/*Beginning of DistTest1.cpp*/
#include<iostream.h>
#include“Distance.h”
void main()
{
  Distance d1(1,1.1);             //parameterized constructor called
  cout<<d1.getFeet()<<“ ”<<d1.getInches();
}
/*End of DistTest1.cpp*/
```

Output

```
1 1.1
```

Listing 4.5 demonstrates a user-defined parameterized costructor being called by creating an object in the stack while Listing 4.6 demonstrates a user-defined parameterized constructor being called in the heap.

Listing 4.6 A user-defined parameterized constructor—called by creating an object in the heap

```
/*Beginning of DistTest2.cpp*/
#include<iostream.h>
#include“Distance.h”
```

```
void main()
{
   Distance * dPtr;
   dPtr = new Distance(1,1.1);      // parameterized
                                    //constructor called Output
   cout<<dPtr->getFeet()<<" "<<dPtr->getInches();
}
/*End of DistTest2.cpp*/
```

Output

1 1.1

The first line of the function `main()` in Listing 4.5 and the second line of the `main()` function in Listing 4.6 show the syntax for passing values to the parameterized constructor. The parameterized constructor is prototyped and defined just like any other member function except for the fact that it does not return any value.

We must remember that if the parameterized constructor is provided and the zero-argument constructor is not provided, the compiler will not provide the default constructor. In such a case, the following statement will not compile.

```
Distance d1;                        //ERROR: No matching constructor
```

Just like in other member functions, the formal arguments of the parameterized constructor can be assigned default values. But in that case, the zero-argument constructor should be provided. Otherwise, an ambiguity error will arise when we attempt to create an object without passing any values for the constructor. See Listing 4.7.

Listing 4.7 Default values given to parameters of a parameterized constructor make the zero-argument constructor unnecessary

```
/*Beginning of Distance.h*/
class Distance
{
   public:
      //Distance();zero-argument constructor commented out
      Distance(int=0,float=0.0);    //default values given
      /*
         rest of the class Distance
      */
};
/*End of Distance.h*/
```

If we write,

```
Distance d1;
```

an ambiguity error arises if the zero-argument constructor is also defined. This is because both the zero-argument constructor and the parameterized constructor can resolve this statement.

Let us now create a parameterized constructor for the class `String`. We will also assign a default value for the argument of the parameterized constructor. The constructor would handle the following statements.

```
String s1("abc");
OR
```

```
char * cPtr = "abc";
String s1(cPtr);
OR
char cArr[10] = "abc";
String s1(cArr);
```

In each of these statements, we are essentially passing the base address of the memory block in which the string itself is stored to the constructor.

In the first case, base address of the memory block of four bytes in which the string "abc" is stored is passed as a parameter to the constructor. But the constructor of the class `String` should be defined in such a manner that 's1.cStr' is made to point at the base of a different memory block of four bytes in the heap area that has been exclusively allocated for the purpose. Only the contents of the memory block, whose base address is passed to the constructor, should be copied into the memory block at which 's1.cStr' points. Finally, 's1.len' should be set to 3. The formal argument of the parameterized constructor for the class `String` will obviously be a character pointer because the address of a memory block containing a string has to be passed to it. Let us call this pointer 'p'. Then, after the statements `String s1 ("abc");` executes, the scenario shown in Figure 4.2 should emerge.

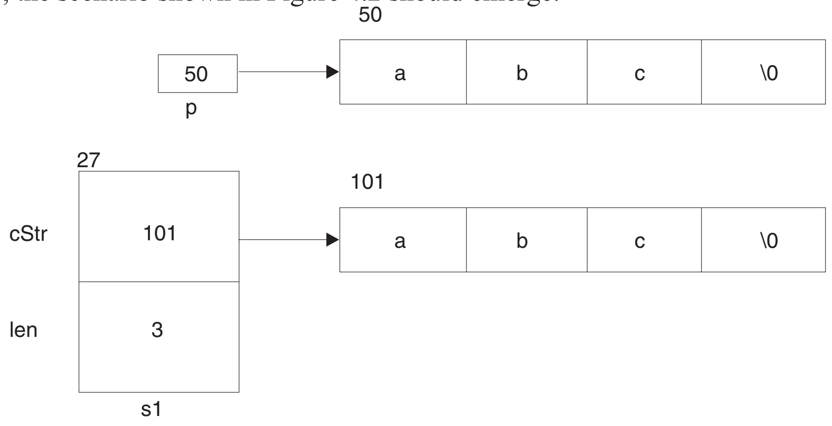


Figure 4.2 Assigning a string to an object of the class `String`

In Figure 4.2, 'p' is the formal argument of the constructor. The address of the memory block that contains the passed string is 50. This address is passed to the constructor and stored in 'p'. Therefore, the value of 'p' is 50. But the constructor should execute in such a manner that a different block that is sufficiently long to hold the string at which 'p' is pointing should also be allocated dynamically in the heap area (see Figure 4.2). This memory block extends from byte numbers 101 to 104. The base address of this block of memory is then stored in the pointer embedded in 's1'. The string is copied from the memory block at which 'p' points to the memory block at which 's1.cStr' points. Finally, 's1.len' is appropriately set to 3.

In the second case

```
char * cPtr = "abc";
String s1(cPtr);
```

the value of 'cPtr' is passed as a parameter to the constructor. This value is stored in 'p'. Thus, both 'p' and 'cPtr' point at the same place. As in the previous case, the constructor of the class `String` should be defined in such a manner that 's1.cStr' should be made to point

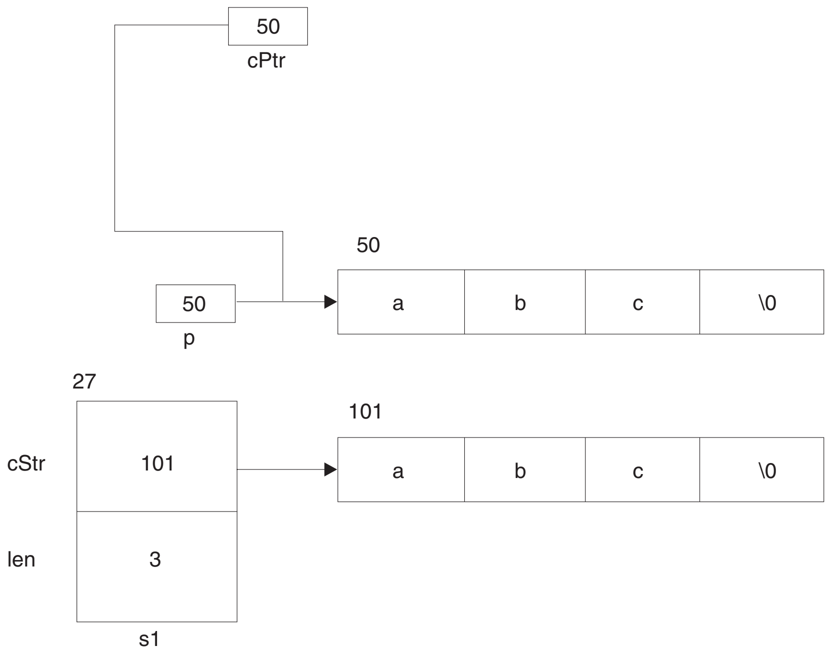


Figure 4.3 Assigning a string to an object of the class `String`

at the base of a different memory block of four bytes that has been exclusively allocated for the purpose. Only the contents of the memory block whose base address is passed to the constructor should be copied into the memory block at which ‘s1.cStr’ points.

In Figure 4.3, ‘cPtr’ points at the memory block containing the string. In other words, the value of ‘cPtr’ is the address of the memory block containing the string.

The third case

```
char cArr[10] = “abc”;
String s1(cArr);
```

is very similar to the second. In this, we are passing the name of the array as a parameter to the constructor. But we know that the name of an array is itself a fixed pointer that contains the base address of the memory block containing the actual contents of the array. This can be seen in Figure 4.4.

Let us now define the constructor that produces these effects. We must realize that ‘p’ (the formal argument of the constructor) should be as follows:

```
const char * const
```

First, it should be a constant pointer because throughout the execution of the constructor, it should continue to point at the same memory block. Second, it should be a pointer to a constant because even inadvertently, the library programmer should not dereference it to change the contents of the memory block at which it is pointing. Additionally, we would like to specify a default value for ‘p’ (NULL) so that there is no need to separately define a zero-argument constructor.

The definition of the class `String` along with the prototype of the constructor and its definition are shown in Listing 4.8.

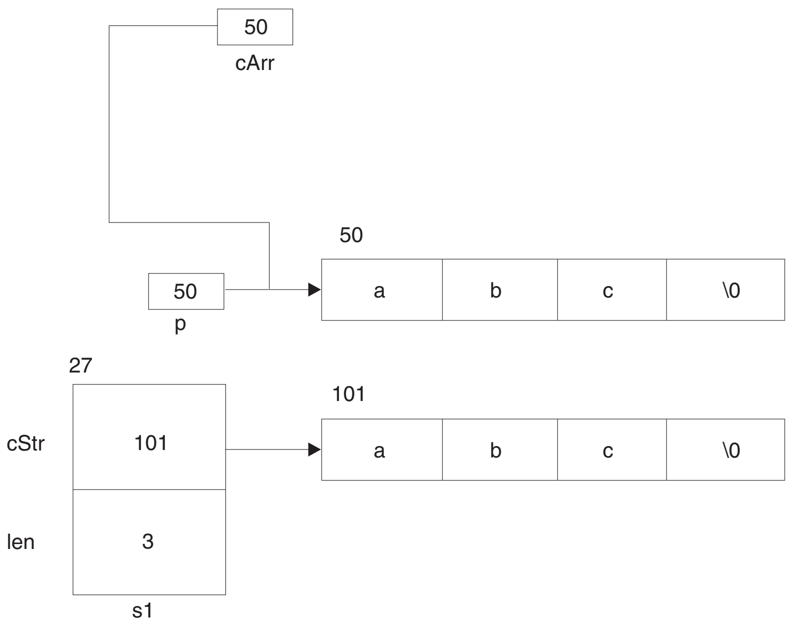


Figure 4.4 Assigning an array to an object of the class `String`

Listing 4.8 A user-defined parameterized constructor for acquiring memory outside the object

```
/*Beginning of String.h*/
class String
{
      char * cStr;
      long unsigned int len;
   public:
      /*no zero-argument constructor*/
      String(const char * const p = NULL);
      const char * getString();
      /*
         rest of the class String
      */
};
/*End of String.h*/

/*Beginning of String.cpp*/
#include"String.h"
#include<string.h>
String::String(const char * const p)
{
   if(p==NULL)        //if default value passed…
   {
      cStr=NULL;      //…nullify
      len=0;
   }
   else               //…otherwise…
   {
      len=strlen(p);
```

```
        cStr=new char[len+1];          //…dynamically allocate a
                                       //separate memory block
        strcpy(cStr,p);                //…and copy into it
      }
    }
    const char * String::getString()
    {
      return cStr;
    }

    /*
      definitions of the rest of the functions of class String
    */
    /*End of String.cpp*/
    /*Beginning of StringMain.cpp*/
    #include"String.h"
    #include<iostream.h>
    void main()
    {
      String s1("abc");                //pass a string to the
                                       //parameterized constructor
      cout<<s1.getString()<<endl;      //display the string
    }
    /*End of StringMain.cpp*/
```

Output

```
abc
```

Another function called `getString()` has also been introduced to the class `String`. It will enable us to display the string itself. The function returns a `const char *` so that only a pointer to a constant can be equated to a call to this function.

```
    const char * p = s1.getString();
```

Such a pointer will effectively point at the same memory block at which the invoking object's pointer points. As a result of the above statement both 'p' and 's1.cStr' would end up pointing at the same place. Yet it will not be able to change the values contained in the memory block since it is a pointer to a constant. We must note that for securing data that is outside the object itself, extra efforts are required on the part of the library programmer.

We can reprogram the above `main()` function and verify that the newly defined constructor is capable of producing the effects depicted in Figures 4.2, 4.3, and 4.4.

4.1.3 Explicit Constructors

Note that the first statement of the `main()` function in Listing 4.8 calls the constructor of the class `String`. Now, look at the following statement.

```
    String s1 = "abc";
```

The above statement also calls the constructor of the class `String`. The above statement compiles because there is a constructor in the class `String` that takes a string as a parameter. This constructor *implicitly* converts the string "abc" into an object of the class `String`. It is as if the above statement was written as follows (note the cast):

```
    String s1 = (String)"abc";
```

But, we did not provide a cast in the statement that we wrote. Then how did the conversion take place? As mentioned earlier, it is the constructor that is carrying out the conversion for us.

However, if the constructor is declared as an explicit constructor, statements like the one above will not compile. Explicit constructors do not allow implicit conversions like the one that occurred in the above example.

Constructors are declared explicit by prefixing their declarations with the `explicit` keyword. Let us first look at the syntax for declaring an explicit constructor (see Listing 4.9). We will then look at a program that will illustrate the situation under which we can get the error if a constructor has been declared as an explicit constructor.

Listing 4.9 The explicit constructor

```
/*Beginning of String.h*/
class String
{
        char * cStr;
        long unsigned int len;
    public:
        /*no zero-argument constructor*/
        /*
           The next statement declares an explicit constructor.
           Note the explicit keyword.
        */
        explicit String(const char * const p = NULL);
        const char * getString();
        /*
           rest of the class String
        */
};
/*End of String.h*/
```

Let us look at Listing 4.10, which illustrates the error we can get when a constructor is declared as an `explicit constructor`.

Listing 4.10 Error caused by the explicit constructor

```
/*Beginning of StringMain.cpp*/
#include<iostream.h>
#include"String.h"
void main()
{
    String s1("abc");              //ok: explicit constructor called
    String s2 = "def";             //error: will not compile due to
                                   //the explicit constructor
}
/*End of StringMain.cpp*/
```

Note that the error in the above program will go away if the statement is written as follows:

```
String s2 = (String)"def";         //ok
```

It is obvious that the `explicit` constructor is preventing an implicit conversion of string into an object of the class `String` and is forcing the application programmer to do explicit conversion.

Further note that we need to mention the `explicit` keyword in the declaration of the constructor only. It is not necessary to prefix the definition of the constructor with the `explicit` keyword.

Explicit constructors can prove to be useful for the programmer if he is creating a class for which an implicit conversion by the constructor is undesirable.

4.1.4 Copy Constructor

The copy constructor is a special type of parameterized constructor. As its name implies, it copies one object to another. It is called when an object is created and equated to an existing object at the same time. The copy constructor is called for the object being created. The pre-existing object is passed as a parameter to it. The copy constructor member-wise copies the object passed as a parameter to it into the object for which it is called.

If we do not define the copy constructor for a class, the compiler defines it for us. But in either case, a call is embedded to it under the following three circumstances.

- When an object is created and simultaneously equated to another existing object, the copy constructor is called for the object being created. The object to which this object was equated is passed as a parameter to the copy constructor.

  ```
  A A1;                             //zero-argument/default constructor called
  A A2=A1;                          //copy constructor called
  ```

  or

  ```
  A A2(A1);                         //copy constructor called
  ```

  or

  ```
  A * APtr = new A(A1);             //copy constructor called
  ```

  Here, the copy constructor is called for 'A2' and for 'Aptr' while 'A1' is passed as a parameter to the copy constructor in both cases.

- When an object is created as a non-reference formal argument of a function. The copy constructor is called for the argument object. The object passed as a parameter to the function is passed as a parameter to the copy constructor.

  ```
  void abc(A);
  A A1;                             //zero-argument/default constructor called
  abc(A1);                          //copy constructor called

  void abc(A A2)
  {
     /*
        definition of abc()
     */
  }
  ```

  Here again the copy constructor is called for 'A2' while 'A1' is passed as a parameter to the copy constructor.

- When an object is created and simultaneously equated to a call to a function that returns an object. The copy constructor is called for the object that is equated to the function call. The object returned from the function is passed as a parameter to the constructor.

  ```
  A abc()
  {
  ```

```
    A A1;        //zero-argument/default constructor called
    /*
      remaining definition of abc()
    */
    return A1;
  }
  A A2=abc();    //copy constructor called
```

Once more, the copy constructor is called for ‘A2’ while ‘A1’ is passed as a parameter to the copy constructor.

The prototype and the definition of the default copy constructor defined by the compiler are as follows.

```
class A
{
  public:
    A(A&);       //the default copy constructor
};

A::A(A& AOBj)   //the default copy constructor
{
*this=AObj;     //copies the passed object into the invoking
                //object
}
```

As is obvious, the default copy constructor does exactly what it is supposed to do—it copies. The statement

```
A A2=A1;
```

is converted as follows:

```
A A2;           //memory allocated for A2
A2.A(A1);       //copy constructor is called for A2 and A1 is
                //passed as a parameter to it
```

This last statement is then transformed to

```
A(&A2,A1);      //see the section on ‘this’ pointer in Chapter 2
```

When the above statement executes, ‘AObj’ (the formal argument in the copy constructor) becomes a reference to ‘A1’, whereas the `this` pointer points at ‘A2’ (the invoking object). Similarly, the other statements where the object is created as a formal argument or is returned from a function can also be explained.

But why does the compiler create the formal argument of the default copy constructor as a reference object? And when the compiler does define a copy constructor in the expected way, then why should we define one on our own? Both these questions are answered now.

First, let us find out why objects are passed by reference to the copy constructor. Suppose the formal argument (‘AObj’) of the copy constructor is not a reference. Now, suppose the following statement executes.

```
A A2=A1;
```

The copy constructor will be called for ‘A2’ and ‘A1’ will be passed as a parameter to it. Then the copy constructor will be called for ‘AObj’ and ‘A1’ will be passed as a parameter to it. This is because ‘AObj’ is a non-reference formal argument of the copy constructor. Thus, an endless chain of calls to the copy constructor will be initiated. However, if the formal argument of the copy constructor is a reference, then no constructor (not even the copy constructor) will be called for it. This is because a reference to an object is not a separate object. No separate memory is allocated for it. Therefore, a call to a constructor is not embedded for it.

Now we come to a crucial question. Why should we define our own copy constructor? After all, the default copy constructor (which is provided free of cost by the complier) does a pretty decent job. First, recollect the conditions we decided to implement for all objects of the class `String`. Suppose an object of the class `String` is created and at the same time equated to another object of the class. For example,

```
String s1("abc");
String s2=s1;  //copy constructor is called for s2 and s1
               //is passed as a parameter to it
```

Since we have not defined the copy constructor for the class `String`, the compiler has done it for us. What does this default copy constructor do in the above case? It simply copies the values of ‘s1’ to ‘s2’! This means that the value of ‘s2.cStr’ becomes equal to ‘s1.cStr’. Thus, both the pointers point at the same place! This is certainly a violation of our conditions. The behaviour of the default copy constructor is undesirable in this case. To overcome this problem of the default copy constructor, we must define our own copy constructor.

From within the copy constructor of the class `String`, a separate memory block must be first allocated dynamically in the heap. This memory block must be equal in length to that of the string at which the pointer of the object passed as a parameter (‘s1’ in this case) points. The pointer of the invoking object (‘s2’ in this case) must then be made to point at this newly allocated memory block. The value of ‘len’ variable of the invoking object should also be set appropriately. However, if the pointer in the object passed as a parameter is NULL, then the value of the pointer and ‘len’ variable of the invoking object must be set to NULL and zero, respectively.

Accordingly, the prototype and the definition of the copy constructor of the class `String` appear as shown in Listing 4.11.

Listing 4.11 A user-defined copy constructor

```
/*Beginning of String.h*/
#include<iostream.h>
class String
{
  char * cStr;
  long unsigned int len;

  public:
  String(const String&);          //our own copy constructor
  /*
    rest of the class String
  */
    explicit String(const char * const p = NULL);
const char * getString();
};
/*End of String.h*/

/*Beginning of String.cpp*/
#include"String.h"
#include<string.h>
String::String(const String& ss)  //our own copy constructor
{
  if(ss.cStr==NULL)               //if passed object's pointer is NULL…
  {
      cStr=NULL;                  //… then nullify the invoking object's
                                  //pointer too
      len=0;
    }
    else                          //otherwise…
    {
      len=ss.len;
      cStr = new char[len+1];     //…dynamically allocate a
                                  //separate memory block
      strcpy(cStr,ss.cStr);       //…and copy into it
    }
  }
  String::String(const char * const p)
  {
    if(p==NULL)                   //if default value passed…
    {
      cStr=NULL;                  //…nullify
      len=0;
    }
    else                          //…otherwise…
    {
      len=strlen(p);
      cStr=new char[len+1];       //…dynamically allocate a
                                  //separate memory block
      strcpy(cStr,p);             //…and copy into it
    }
  }
  const char * String::getString()
  {
    return cStr;
  }
  /*End of String.cpp*/

  /*Beginning of StringMain.cpp*/
  #include"String.h"
  #include<iostream.h>
  void main()
  {
    String s1("abc");
    String s2=s1;
    cout<<s1.getString()<<endl;
    cout<<s2.getString()<<endl;
  }
  /*End of StringMain.cpp*/
```

Output

```
abc
abc
```

In the copy constructor (Listing 4.11), the formal argument is a constant. It has to be a reference in order to prevent an endless chain of calls to itself. But at the same time the library programmer would certainly want to prevent even an inadvertent change in the values of the object that gets passed to the copy constructor. He/she would like the compiler to report a compile-time error if he/she inadvertently writes statements like the following.

```
ss.cStr=NULL;   //pointer of parameter object modified!
ss.len++;       //len variable of the parameter object
                //modified!
```

4.2 Destructors

The destructor gets called for each object that is about to go out of scope. It appears as a member function of each class whether we define it or not. It has the same name as that of the class but prefixed with a tilde sign. It does not take parameters. It does not return anything (not even void). The prototype of a destructor is

```
~ <class name> ();
```

The need for a function that guarantees deinitialization of member data of a class and frees up the resources acquired by the object during its lifetime will be explained soon. Destructors fulfill this need.

The compiler embeds a call to the destructor for every object when it is destroyed. Let us have one more look at the `main()` function of Listing 4.1.

```
void main()
{
  A A1;
} //A1 goes out of scope here
```

'A1' goes out of scope just before the `main()` function terminates. At this point, the compiler embeds a call to the destructor for 'A1'. It embeds the following statement.

```
A1.~A();        //destructor called … not legal C++ code
```

An explicit call to the destructor for an existing object is forbidden. The above statement is then transformed into

```
~A(&A1);                          //see chapter 2
```

The destructor will also be called for an object that has been dynamically created in the heap just before the `delete` operator is applied on the pointer pointing at it.

```
A * APtr;
APtr = new A;                     //object created … constructor called
. . . .
. . . .
delete APtr;                      //object destroyed … destructor called
```

The last statement is transformed into

```
APtr->~A();                       //destructor called for *APtr
delete APtr;                      //memory for *APtr released
```

First, the destructor is called for the object that is going out of scope. Thereafter, the memory occupied by the object itself is deallocated. The second last statement above is transformed into

```
~A(APtr);               //see the section on 'this' pointer in Chapter 2
```

Unlike its name, the destructor does not 'destroy' or deallocate memory that an object occupies. It is merely a member function that is called for each object just before the object goes out of scope (gets destroyed).

As can be readily observed, the compiler embeds a call to the destructor for each and every object that is going out of scope. But we did not prototype and define the destructor inside the class. Then how was the above call to the destructor resolved? The compiler prototypes and defines the destructor for us. But what statements does the definition of such a destructor have? The answer is 'nothing'. An example of a compiler-defined destructor follows.

Before

```
class A
{
     . . . .
     . . . .
  public:
     . . . .
     . . . .
     //no destructor
};
```

After

```
class A
{
     . . . .
     . . . .
  public:
     ~A();     //prototype inserted implicitly by compiler
     . . . .
     . . . .
};

A::~A()
{
  //empty definition inserted implicitly by compiler
}
```

Let us add our own destructor to the class A defined in Listing 4.2 and verify whether the destructor is actually called implicitly by the compiler or not. See Listing 4.12.

Listing 4.12 Destructor gets called for each object when the object is destroyed

```
/*Beginning of A.h*/
class A
{
     int x;
  public:
     A();
     void setx(const int=0);
     int getx();
     ~A();                        //our own destructor
};
/*End of A.h*/

/*Beginning of A.cpp*/
#include"A.h"
#include<iostream.h>
A::A()
{
  cout<<"Constructor of class A called\n";
}

A::~A()                           //our own destructor
{
  cout<<"Destructor of class A called\n";
}

/*
  definitions of the rest of the functions of class A
*/
```

```
/*End of A.cpp*/

/*Beginning of AMain.cpp*/
#include"A.h"
#include<iostream.h>
void main()
{
   A A1;
   cout<<"End of program\n";
}
/*End of AMain.cpp*/
```

Output

```
Constructor of class A called
End of program
Destructor of class A called
```

As we can see, the name of the destructor is the same as the name of the class but prefixed with a tilde sign. Moreover, the destructor does not return anything. The compiler defines the destructor in order to resolve the call to the destructor that it compulsorily places for the object going out of scope.

Destructors do not take any arguments. Therefore, they cannot be overloaded.

Why should we define our own destructor? We must remember that the destructor is also a member function. It is called for objects. Therefore, it can access the data members of the object for which it has been called.

Let us think of a relevant definition for the destructor of the class `Distance`. What would we like it to do for us? What should it do to the data members of the object that is going out of scope? Should it set them to zero?

```
Distance::~Distance()
{
   iFeet=0;
   fInches=0.0;
}
```

But what is the use? The object is anyway going out of scope immediately after the destructor executes.

But we must define the destructor for classes whose objects, during their lifetime, acquire resources that are outside the objects themselves. Let us take the example of the class `String`. We consider the following code block.

```
{
   . . . .
   . . . .
   String s1("abc");
   . . . .
   . . . .
}
```

The memory that was allocated to 's1' itself gets deallocated when this block finishes execution. But 's1.cStr' was pointing at a memory block that was dynamically allocated in the heap area. This memory block was outside the memory block occupied by 's1' itself. After 's1' gets destroyed, this memory block remains allocated as a locked up lost resource. The only pointer that was pointing at it ('s1.cStr') is no longer available. This is memory leak. It should be prevented. We should deallocate the memory block at which the pointer inside any object of the class `String` is pointing exactly when the object goes out of scope. This means that we must call the `delete` operator for the pointer inside the class `String` and place this statement inside the destructor. See Listing 4.13.

Listing 4.13 A user-defined destructor

```
/*Beginning of String.h*/
class String
{
    char * cStr;
    long unsigned int len;
  public:
    ~String();                  //our own destructor
    /*
      rest of the class String
    */
};
/*End of String.h*/

/*Beginning of String.cpp*/
#include"String.h"
#include<string.h>
String::~String()               //our own destructor
{
  if(cStr!=NULL)                //if memory exists
    delete[] cStr;              //… destroy it
}

/*
  definitions of the rest of the functions of class String
*/
/*End of String.cpp*/
```

4.3 Philosophy of OOPS

Now, let us digress and appreciate the basic philosophy of OOPS. One of the aims in OOPS is to abolish the use of fundamental data types. Classes can contain huge amounts of functionality (member functions) that free the application programmer from the worry of taking precautions against bugs.

The class `String` is one such data type. By adding some more relevant functions, we can conveniently use objects of the class `String`. Consider adding the following function to the class `String`.

```
void String::addChar(char);     //function to add a character
                                //to the string
```

As its name suggests, this function will append a character to the string at which the pointer inside the invoking object points.

```
String s1("abc");
```

As a result of this statement, the pointer inside 's1' points at a memory block of four bytes (last one containing NULL). Now, if we write

```
s1.addChar('d');                //add a character to the string
```

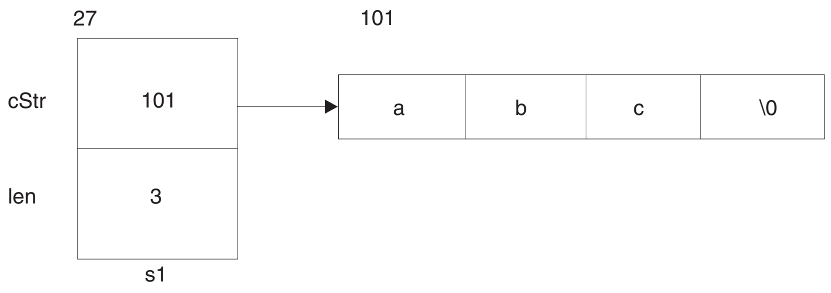
the following things should happen.

- Another block of five bytes should get allocated.
- The string contained in the memory block at which ‘s1.cStr’ is currently pointing should get copied into this new memory block.
- The character ‘d’ should get appended to the string.
- The null character should get further appended to the string.
- ‘s1.cStr’ should be made to point at this new memory block.
- The memory block at which ‘s1.cStr’ was pointing previously should be deallocated (to prevent memory leaks).

Figure 4.5 shows adding a character to a stretchable string in the object-oriented way.

Before

```
String s1(“abc”);
```



After

```
s1.addChar(‘d’);
```

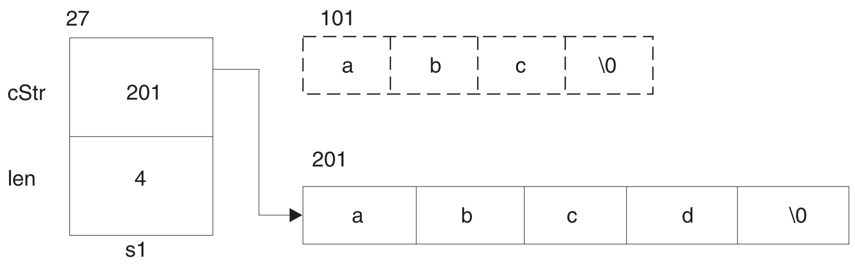


Figure 4.5 Adding a character to a stretchable string—the object-oriented way

One possible way of using this function is by using a loop to obtain a string from the user, which can be of any length. While writing the program, the application programmer need not predict the length of the string the user will enter. The following code can be used for adding a character to a stretchable string in the object-oriented way.

```
while(1)        //potentially infinite loop
{
   ch=getche();
   if(ch==‘\n’)                    //if user finishes entering the string
      break;                       //… break the loop
   s1.addChar(ch);                 //…else append the character to it
}
```

As the user keeps adding characters to the string, the allocated memory keeps getting stretched in a manner that is transparent to the application programmer. Such an effect is simply unthinkable with character arrays.

We would also like to add a function that will replace the string associated with an object with the string that we pass to it. We let this function be

```
void String::setString(const char * const);
```

Suppose the following statements are executed.

```
String s1("abc");
s1.setString("def");                //replace "abc" by "def"
```

Then the following events should take place when the second statement executes ('s1.cStr' is already pointing at a memory block that contains the string `abc` and is not NULL).

- A block of four bytes should be dynamically allocated to accommodate the string "def".
- The string `def` should get written in that memory block with the null character appended.
- `s1.cStr` should be made to point at this new block of memory.
- The block of memory at which `s1.cStr` was previously pointing should be deallocated to prevent memory leak.

The formal argument of the `String::setString()` function is a `const char * const`. The reasons for this have already been discussed under the section on parameterized constructor. We may think that the definition of this function will be the same as that of the constructor. But this is not so. When the constructor starts executing, `cStr` may or may not be NULL (it may contain junk value). But if it is not NULL, it does not mean that it is pointing at a dynamically allocated block of memory. But when the `String::setString()` function starts executing, if `cStr` is not NULL, then it is definitely pointing at a dynamically allocated block of memory. Statements to check this condition and to deallocate the memory block and to nullify `cStr` and to set 'len' to zero should be inserted at the beginning of the `String::setString()` function. Otherwise a memory leak will occur. Defining the `String::addChar()` and `String::setString()` functions is left as an exercise.

Let us think of more such relevant functions that can be added to the class `String`. There can be a function that will change the value of a character at a particular position in the string at which the pointer of the invoking object points. Moreover, there can be a function that reads the value from a particular position in the string at which the pointer of the invoking object points. These functions can have built-in checks to prevent values from being written to or read from bytes that are beyond the memory block allocated. Again, such a check is not built into character arrays. The application programmer has to put in extra efforts on his/her own to prevent the program from exceeding the bounds of the array.

After we have added all such functions to the class `String`, we will get a new data type that will be safe, efficient, and convenient to use.

Suitably defined constructors and destructors have a vital role to play in the creation of such data types. Together they ensure that

- There are no memory leaks (the destructor frees up unwanted memory).
- There are no run-time errors (no two calls to the destructor try to free up the same block of memory).
- Data is never in an invalid state and domain constraints on the values of data members are never violated.

After such data types have been defined, new data types can be created that extend the definitions of existing data types. They contain the definition of the existing data types and at the same time add more specialized features on their own. This facility of defining new data types by making use of existing data types is known as inheritance. Chapter 5 deals with this feature of OOPS and its implementation in C++.

Summary

Constructors can be used to guarantee a proper initialization of data members of a class. Domain constraints on values of data members can be implemented via constructors.

Constructors are member functions and have the same name as that of the class itself. The compiler creates a zero-argument constructor and a copy constructor if we do not define them. Constructors take parameters and, therefore, can be overloaded. They do not return anything (not even void). The compiler implicitly embeds a call to the constructor for each object that is being created. An explicit call to the constructor for an existing object is forbidden.

If necessary, destructors can be used to guarantee a proper clean up when an object goes out of scope. Destructors are member functions and have the same name as that of the class itself but with the tilde sign prefixed. The compiler creates a destructor if we do not define one. Destructors do not take parameters and, therefore, cannot be overloaded. They do not return anything (not even void). The compiler implicitly embeds a call to the destructor for each object that is going out of scope (being destroyed). An explicit call to the destructor for an existing object is forbidden.

Key Terms

constructors
- called automatically for each object that has just got created
- defined by default
- has the same name as that of the class
- does not return anything

zero-argument constructor
parameterized constructors
copy constructor
destructors

Exercises

1. What are constructors? When are they called? What is their utility?
2. Why should the formal argument of a copy constructor be a reference object?
3. What are destructors? When are they called? What is their utility?
4. Is a destructor necessary for the following class?

```
class Time
{
      int hours, minutes, seconds;
   public:
      /*
         rest of the class Time … but no
         more data members
      */
};
```

5. Define a suitable parameterized constructor with default values for the class `Time` given in question 4.
6. Four member functions are provided by default by the compiler for each class that we define. We have studied three of them in this chapter. Name them.
7. State true or false.
   (a) Memory occupied by an object is allocated by the constructor of its class.
   (b) Constructors can be used to acquire memory outside the objects.
   (c) Constructors can be overloaded.
   (d) A constructor can have a return statement in its definition.
   (e) Memory occupied by an object is deallocated by the destructor of its class.

(f) Destructors can be used to release memory that has been acquired outside the objects.
(g) Destructors can be overloaded.
(h) A destructor can have a return statement in its definition.

8. The copy constructor has been explicitly defined for the class `String` so that no two objects of the class `String` end up sharing the same resource, that is, end up with their contained pointers pointing at the same block of dynamically allocated memory. In this case, two such blocks may contain two copies of the same data as a result of the copy constructor, which is perfectly acceptable. However, there are situations where no two objects should share even copies of the same data. If A is a class for whose objects this restriction needs to be applied, then we should ensure that a statement like the second one below should not compile.

```
A A1;
A A2 = A1;
```

How can this objective be achieved? (*Hint:* Member functions are not always public and the copy constructor is a member function.)