In the definition (Listing 2.4), the member functions have not been placed under any access modifier. Therefore, they are public members by default.

On the other hand, *class members are private by default.* This is the only difference between the `class` keyword and the `struct` keyword.

Thus, the structure 'Distance' can be redefined by using the `class` keyword as shown in Listing 2.5.

```
class Distance
{
    int iFeet;              //private by default
    float fInches;          //private by default
  public:
    void setFeet(int x)
    {
      iFeet=x;
    }
    int getFeet()
    {
      return iFeet;
    }
    void setInches(float y)
    {
      fInches=y;
    }
    float getInches()
    {
      return fInches;
    }
};
```

Listing 2.5 Class members are private by default

The `struct` keyword has been retained to maintain backward compatibility with C language. A header file created in C might contain the definition of a structure, and structures in C will have member data only. A C++ compiler will easily compile a source code that has included the above header file since the new definition of the `struct` keyword allows, not mandates, the inclusion of member functions in structures.

Functions in a C language source code access member data of structures. A C++ compiler will easily compile such a source code since the C++ compiler treats members of structures as public members by default.

Objects

Variables of classes are known as objects.

An object of a class occupies the same amount of memory as a variable of a structure that has the same data members. This is illustrated by the following.

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

struct A
{
  char a;
  int b;
  float c;
};

class B   //a class with the same data members
{
  char a;
  int b;
  float c;
};

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

Output

```
9
9
```

Listing 2.6 Size of a class object is equal to that of a structure variable with identical data members

Introducing member functions does not influence the size of objects. The reason for this will become apparent when we study the 'this' pointer. Moreover, making data members private or public does not influence the size of objects. The access modifiers merely control the accessibility of the members.

The Scope Resolution Operator

It is possible and usually necessary for the library programmer to define the member functions *outside* their respective classes. The scope resolution operator makes this possible. The following example illustrates the use of the scope resolution operator (::).

```
/*Beginning of scopeResolution.cpp*/
class Distance
{
    int iFeet;
    float fInches;
  public:
    void setFeet(int);          //prototype only
    int getFeet();              //prototype only
    void setInches(float);      //prototype only
    float getInches();          //prototype only
};

void Distance::setFeet(int x)       //definition
{
  iFeet=x;
}

int Distance::getFeet()             //definition
{
  return iFeet;
}

void Distance::setInches(float y)   //definition
{
  fInches=y;
}

float Distance::getInches()         //definition
{
  return fInches;
}
/*End of scopeResolution.cpp*/
```

Listing 2.7 The scope resolution operator

We can observe that the member functions have been only prototyped within the class; they have been *defined* outside. The scope resolution operator signifies the class to which they belong. The class name is specified on the left-hand side of the scope resolution operator. The name of the function being defined is on the right-hand side.

Creating Libraries using the Scope Resolution Operator

As in C language, creating a new data type in C++ using classes is also a three-step process that is executed by the library programmer.

Step 1: Place the class definition in a header file.

```
/*Beginning of Distance.h*/
/*Header file containing the definition of the Distance
class*/

class Distance
{
    int iFeet;
    float fInches;
  public:
    void setFeet(int);       //prototype only
    int getFeet();           //prototype only
    void setInches(float);   //prototype only
    float getInches();       //prototype only
};
/*End of Distance.h*/
```

Step 2: Place the definitions of the member functions in a C++ source file (the library source code). A file that contains definitions of the member functions of a class is known as the implementation file of that class. Compile this implementation file and put in a library.

```
/*Beginning of Distlib.cpp*/
/*Implementation file for the class Distance*/
#include"Distance.h"

void Distance::setFeet(int x)          //definition
{
  iFeet=x;
}
int Distance::getFeet()                //definition
{
  return iFeet;
}

void Distance::setInches(float y)      //definition
{
  fInches=y;
}

float Distance::getInches()       //definition
{
  return fInches;
}
/*End of Distlib.cpp*/
```

Step 3: Provide the header file and the library, in whatever media, to other programmers who want to use this new data type.

Using Classes in Application Programs

The steps followed by programmers for using this new data type are:

Step 1: Include the header file provided by the library programmer in their source code.

```
/*Beginning of Distmain.cpp*/
#include"Distance.h"

void main()
{
  . . . .
  . . . .
}
/*End of Distmain.cpp*/
```

Step 2: Declare variables of the new data type in their source code.

```
/*Beginning of Distmain.cpp*/
#include"Distance.h"

void main()
{
  Distance d1,d2;
  . . . .
  . . . .
}
/*End of Distmain.cpp*/
```

Step 3: Embed calls to the associated functions by passing these variables in their source code.

```
/*Beginning of Distmain.cpp*/
/*A sample driver program for creating and using objects
of the class Distance*/
#include<iostream.h>
#include"Distance.h"

void main()
{
  Distance d1,d2;
  d1.setFeet(2);
  d1.setInches(2.2);
  d2.setFeet(3);
  d2.setInches(3.3);
  cout<<d1.getFeet()<<" "<<d1.getInches()<<endl;
  cout<<d2.getFeet()<<" "<<d2.getInches()<<endl;
}
/*End of Distmain.cpp*/
```

Listing 2.8 Using classes in application programs

Step 4: Compile the source code to get the object file.

Step 5: Link the object file with the library provided by the library programmer to get the executable or another library.

Output of Listing 2.8
2 2.2
3 3.3

Implementation files are compiled and converted into static and dynamic libraries in the usual manner.

Again, we notice that there is no obvious connection between the member data being accessed within the member function and the object that is invoking the function.

The 'this' Pointer

The facility to create and call member functions of class objects is provided by the C++ compiler. You have already seen how this facility is to be used. However, how does the compiler support this facility? The compiler does this by using a unique pointer known as the 'this' pointer. A thorough understanding of the 'this' pointer is vital for understanding many concepts in C++.

The 'this' pointer is always a constant pointer. The 'this' pointer always points at the object with respect to which the function was called. An explanation that follows shortly explains why and how it functions.

After the compiler has ascertained that no attempt has been made to access the private members of an object by non-member functions, it converts the C++ code into an ordinary C language code as follows:

1. It converts the class into a structure with only data members as follows.

   Before

```
class Distance
{
     int iFeet;
     float fInches;
   public:
     void setFeet(int);            //prototype only
     int getFeet();                //prototype only
     void setInches(float);        //prototype only
     float getInches();            //prototype only
};
```

After

```
struct Distance
{
  int iFeet;
  float fInches;
};
```

2. It puts a declaration of the 'this' pointer as a leading formal argument in the prototypes of all member functions as follows.

Before

```
void setFeet(int);
```

After

```
void setFeet(Distance * const, int);
```

Before

```
int getFeet();
```

After

```
int getFeet(Distance * const);
```

Before

```
void setInches(float);
```

After

```
void setInches(Distance * const, float);
```

Before

```
float getInches();
```

After

```
float getInches(Distance * const);
```

3. It puts the definition of the 'this' pointer as a leading formal argument in the definitions of all member functions as follows. It also modifies all the statements to access object members by accessing them through the 'this' pointer using the pointer-to-member access operator (->).

Before

```
void Distance::setFeet(int x)
{
  iFeet=x;
}
```

After

```
void setFeet(Distance * const this, int x)
{
  this->iFeet=x;
}
```

Before

```
int Distance::getFeet()
{
  return iFeet;
}
```

After

```
int getFeet(Distance * const this)
{
  return this->iFeet;
}
```

Before

```
void Distance::setInches(float y)
{
  fInches=y;
}
```

After

```
void setInches(Distance * const this, float y)
{
  this->fInches=y;
}
```

Before

```
float Distance::getInches()
{
  return fInches;
}
```

After

```
float getInches(Distance * const this)
{
  return this->fInches;
}
```

We must understand how the scope resolution operator works. The scope resolution operator is also an operator. Just like any other operator, it operates upon its operands. The scope resolution operator is a *binary* operator, that is, it takes two operands. The operand on its left is the name of a pre-defined class. On its right is a member function of that class. Based upon this information, the scope resolution operator inserts a constant operator of the correct type as a leading formal argument to the function on its right. For example, if the class name is 'Distance', as in the above case, the compiler inserts a pointer of type 'Distance * const' as a leading formal argument to the function on its right.

4. It passes the address of invoking object as a leading parameter to each call to the member functions as follows.

Before

```
d1.setFeet(1);
```

After

```
setFeet(&d1,1);
```

Before

```
d1.setInches(1.1);
```

After

```
setInches(&d1,1.1);
```

Before

```
cout<<d1.getFeet()<<endl;
```

After

```
cout<<getFeet(&d1)<<endl;
```

Before

```
cout<<d1.getInches()<<endl;
```

After

```
cout<<getInches(&d1)<<endl;
```

In the case of C++, the dot operator's definition has been extended. It not only takes data members as in C but also member functions as its right-hand side operand. If the operand on its right is a data member, then the dot operator behaves just like it does in C language. However, if the operand on its right is a member function, then the dot operator causes the address of the object on its left to be passed as an implicit leading parameter to the function call.

Clearly, members of the *invoking object* are referred to when they are accessed without any qualifiers in member functions. It should also be obvious that multiple copies of member data exist (one inside each object) but only one copy exists for each member function.

It is evident that the 'this' pointer should continue to point at the same object—the object with respect to which the member function has been called—throughout its lifetime. For this reason, the compiler creates it as a constant pointer.

The accessibility of the implicit object is the same as that of the other objects passed as parameters in the function call and the local objects inside that function. The following example illustrates this. A new function—'add()'—has been added to the existing definition of the 'Distance' class.

```
/*Beginning of Distance.h*/
class Distance
{
    /*
      rest of the class Distance
```

```
    */
    Distance add(Distance);
};
/*End of Distance.h*/

/*Beginning of Distlib.cpp*/
#include"Distance.h"

Distance Distance::add(Distance dd)
{
  Distance temp;
  temp.iFeet=iFeet+dd.iFeet;        //legal to access both
                                    //temp.iFeet and
                                    //dd.iFeet
  temp.fInches=fInches+dd.fInches; //ditto.
  return temp;
}

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

/*Beginning of Distmain.cpp*/
#include<iostream.h>
#include"Distance.h"

void main()
{
  Distance d1,d2,d3;
  d1.setFeet(1);
  d1.setInches(1.1);
  d2.setFeet(2);
  d2.setInches(2.2);
  d3=d1.add(d2);
  cout<<d3.getFeet()<<"'-"<<d3.getInches()<<"''\n";
}
/*End of Distmain.cpp*/
```

Output

3'-3.3'

Listing 2.9 Accessing data members of local objects inside member functions and of objects that are passed as parameters

The definition of 'Distance :: add()' function, after the previously described conversion by the compiler is carried out, will appear as follows.

```
Distance add(Distance * const this, Distance dd)
{
  Distance temp;
  temp.iFeet=this->iFeet+dd.iFeet;
  temp.fInches=this->fInches+dd.fInches;
  return temp;
}
```

When this function is called from the 'main()' function with respect to 'd1', the 'this' pointer points at 'd1'. Thus, it is the private data member of 'd1' that is being accessed in the second and third lines of the 'add()' function.

So, now we can

- Declare a class
- Define member data and member functions
- Make members private and public
- Declare objects and call member functions with respect to objects

What advantages does all this lead to? The advantage that library programmers can now derive from this arrangement is epitomized in the following observation:

'An executable file will not be created from a source code in which private data members of an object have been accessed by non-member functions.'

Once again, the importance of compile-time errors over run-time errors is emphasized. Suppose, an if block exists in a function that is not intended by the library programmer to access the data members of a structure. This if block contains a bug (say 'd1.month' has been assigned the value 13, where 'd1' is a variable of the structure 'date').

A pure C compiler would not recognize this statement as an invalid access. During testing, the if condition of this if block might never become true. The bug would remain undetected; the executable will get created with bugs. Thus, *creating bug-free executables is difficult and unreliable in C.* This is due to the absence of language constructs that enforce data security.

On the other hand, a C++ compiler that also detects invalid access of private data members would immediately throw an error during compile time itself and prevent the creation of the executable. Thus, *creating bug-free executables is easier and more reliable in C++ than in C.* This is due to the presence of language constructs that enforce data security.

Data Abstraction

The class construct provides facilities to implement data abstraction. Data abstraction is an important concept and should be understood properly. Let us take up the example of the LCD projector from the previous chapter. It has member data (light and fan) as well

as member functions (switches that operate the light and the fan). This real-world object hides its internal operations from the outside world. It, thus, obviates the need for the user to know the possible pitfalls that might be encountered during its operation. During its operation, the LCD projector never reaches an invalid state. Moreover, the LCD projector does not start in an invalid state.

Data abstraction is a virtue by which an object hides its internal operations from the rest of the program. It makes it unnecessary for the client programs to know how the data is internally arranged in the object. Thus, it obviates the need for the client programs to write precautionary code upon creating and while using objects.

Now, in order to understand this concept, let us take an example in C++. The library programmer, who has designed the 'Distance' class, wants to ensure that the 'fInches' portion of an object of the class should never exceed 12. If a value larger than 12 is specified by an application programmer while calling the 'Distance::setInches()' function, the logic incorporated within the definition of the function should automatically increment the value of 'iFeet' and decrement the value of 'fInches' by suitable amounts. A modified definition of the 'Distance::setInches()' function is as follows.

```
void Distance::setInches(float y)
{
  fInches=y;
  if(fInches>=12)
  {
    iFeet+=fInches/12;
    fInches-=((int)fInches/12)*12;
  }
}
```

Here, we notice that an application programmer need not send values less than 12 while calling the 'Distance::setInches()' function. The default logic within the 'Distance::setInches()' function does the necessary adjustments. This is an example of data abstraction.

The above restriction may not appear mandatory. However, very soon we will create classes where similar restrictions will be absolutely necessary (and also complicated).

Similarly, the definition of the 'Distance::add()' function should also be modified as follows by the library programmer. Here it can assumed that the value of 'fInches' portion of neither the invoking object nor the object appearing as formal argument ('dd') can be greater than 12.

```
Distance Distance::add(Distance dd)
{
  Distance temp;
  temp.iFeet=iFeet+dd.iFeet;
  temp.setInches(fInches+dd.fInches);
  return temp;
}
```

Now, if we write

```
d1.setFeet(1);
d1.setInches(9.5);
d2.setFeet(2);
d2.setInches(5.5);
d3=d1.add(d2);
```

Listing 2.10 Enforcing restrictions on the data members of a class

then the value of 'd3.fInches' will become 3 (not 15) and the value of 'd3.iFeet' will become 4 (not 3).

It has already been mentioned that real-world objects never attain an invalid state. They also do not start in an invalid state. Does C++ enable the library programmer to implement this feature in class objects?

Let us continue with our earlier example—the 'Distance' class. Recollect that it is the library programmer's intention to ensure that the value of 'fInches' portion of none of the objects of the class 'Distance' should exceed 12. Now, let us see Listing 2.11.

```
/*Beginning of DistJunk.cpp*/
#include<iostream.h>
#include"Distance.h"

void main()
{
  Distance d1;
  cout<<d1.getFeet()<<" "<<d1.getInches()<<endl;
}
/*End of DistJunk.cpp*/
```

Output

297 34.56

Listing 2.11 Object gets created with improper values

As you can see, the value of 'fInches' of 'd1' is larger than 12! This happened because the value of both 'iFeet' and 'fInches' automatically got set to junk values when 'd1' was allocated memory and the junk value is larger than 12 for 'd1.fInches'. Thus, the objective of the library programmer to keep the value of 'fInches' less than 12 has not yet been achieved.

It would be unrealistic to expect that an application programmer will explicitly initialize each object that is declared.

```
Distance d1;
d1.setFeet(0);       //initialization
d1.setInches(0.0);   //initialization
```

Obviously, the library programmer would like to add a function to the 'Distance' class that gets called automatically whenever an object is created and sets the values of the data members of the object properly. Such a function is the *constructor*. The concept of constructor and a related function, the *destructor*, is discussed in one of the later chapters.

But we may say that even if 'Distance' was an ordinary structure and 'setInches()' function was a non-member function just as in C, data abstraction would still be in place. Nevertheless, in the case of C, the library programmer cannot force calls to only those functions that have been defined. He/she cannot prevent calls to those functions that he/she has not defined. *Data abstraction is effective due to data hiding only* (recall the case of the overhead projector systems discussed earlier).

On the other side of the coin, in C language, life becomes difficult for an application programmer also. If a certain member of a structure variable acquires an invalid or a wrong value, he/she has to hunt through the entire source code to detect the bug. This problem rapidly gains significance as the code length increases. In actual practice, it is common to have code of more than 25,000 lines.

Let us now sum up as follows:

'Perfect definitions of the member functions are guaranteed to achieve their objective because of data hiding.'

This is the essence of the object-oriented programming system. Real-world objects have not only working parts but also an exclusive interface to these inner-working parts. A perfect interface is guaranteed to work because of its exclusive rights.

Explicit Address Manipulation

An application programmer can manipulate the member data of any object by explicit address manipulation. The following code illustrates the point.

```
/*Beginning of. DistAddrManip.cpp*/
#include"Distance.h"
#include<iostream.h>
```

```
void main()
{
  Distance d1;
  d1.setFeet(256);
  d1.setInches(2.2);
  char * p=(char *)&d1;  //explicit address manipulation
  *p=1;                  //undesirable but unpreventable
  cout<<d1.getFeet()<<" "<<d1.getInches()<<endl;
}
/*End of DistAddrManip.cpp*/
```

Output

257 2.2

Listing 2.12 Explicit address manipulation

However, such explicit address manipulation by an application programmer cannot be prevented. It is left as an exercise for the readers to explain the output of the above program (Listing 2.12).

The Arrow Operator

Member functions can be called with respect to an object through a pointer pointing at the object. The arrow operator (->) does this. An illustrative example follows.

```
/*Beginning of PointerToMember.cpp*/
#include<iostream.h>
#include"Distance.h"

void main()
{
  Distance d1;           //object
  Distance * dPtr;       //pointer
  dPtr=&d1;              //pointer initialized
  /*Same as d1.setFeet(1) and d1.setInches(1.1)*/
  dPtr->setFeet(1);      //calling member functions
  dPtr->setInches(1.1);  //through pointers
  /*Same as d1.getFeet() and d1.getInches()*/
  cout<<dPtr->getFeet()<<" "<<dPtr->getInches()<<endl;
}
/*End of PointerToMember.cpp*/
```

Output

1 1.1

Listing 2.13 Accessing members through pointers

It is interesting to note that just like the dot (.) operator, *the definition of the arrow (->) operator has also been extended in C++*. It takes not only data members on its right as in C, but also member functions as its right-hand side operand. If the operand on its right is a data member, then the arrow operator behaves just as it does in C language. However, if it is a member function of a class where a pointer of the same class type 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. Thus, the statement

```
dPtr->setFeet(1);
```

after conversion becomes

```
setFeet(dPtr,1);
```

Now, the value of 'dPtr' is copied into the 'this' pointer. Therefore, the 'this' pointer also points at the same object at which 'dPtr' points.

Calling One Member Function from Another

One member function can be called from another. An illustrative example follows.

```
/*Beginning of NestedCall.cpp*/
class A
{
    int x;
  public:
    void setx(int);
    void setxindirect(int);
};

void A::setx(int p)
{
  x=p;
}

void A::setxindirect(int q)
{
  setx(q);
}

void main()
{
  A A1;
  A1.setxindirect(1);
}
/*End of NestedCall.cpp*/
```

Listing 2.14 Calling one member function from another

It is relatively simple to explain the above program. The call to the 'A::setxindirect()' function changes from

```
A1.setxindirect(1);
```

to

```
setxindirect(&A1,1);
```

The definition of the 'A::setxindirect()' function changes from

```
void A::setxindirect(int q)
{
  setx(q);
}
```

to

```
void setxindirect(A * const this, int q)
{
  this->setx(q);     //calling function through a pointer
}
```

which, in turn, changes to

```
void setxindirect(A * const this, int q)
{
  setx(this,q);      //action of arrow operator
}
```

2.2 Member Functions and Member Data

Let us study the various kinds of member functions and member data that classes in C++ have.

Overloaded Member Functions

Member functions can be overloaded just like non-member functions. The following example illustrates the point.

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

class A
{
  public:
```

```
    void show();
    void show(int); //function show() overloaded!!
};

void A::show()
{
  cout<<"Hello\n";
}

void A::show(int x)
{
  for(int i=0;i<x;i++)
    cout<<"Hello\n";
}

void main()
{
  A A1;
  A1.show();          //first definition called
  A1.show(3);         //second definition called
}
/*End of memFuncOverload.cpp*/
```

Output

Hello
Hello
Hello
Hello

Listing 2.15 Overloaded member functions

Function overloading enables us to have two functions of the same name and same signature in two different classes. The following class definitions illustrate the point.

```
class A
{
  public:
    void show();
};
class B
{
  public:
    void show();
};
```

Listing 2.16 Facility of overloading functions permits member functions of two different classes to have the same name

A function of the same name 'show()' is defined in both the classes—'A' and 'B'. The signature also appears to be the same. But with our knowledge of the 'this' pointer, we know that the signatures are *actually* different. The function prototypes in the respective classes are actually as follows.

```
void show(A * const);
void show(B * const);
```

Without the facility of function overloading, it would not be possible for us to have two functions of the same name in different classes. Without the facility of function overloading, choice of names for member functions would become more and more restricted. Later, we will find that function overloading enables function overriding that, in turn, enables dynamic polymorphism.

Default Values for Formal Arguments of Member Functions

We already know that default values can be assigned to arguments of non-member functions. Default values can be specified for formal arguments of member functions also. An illustrative example follows.

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

class A
{
  public:
    void show(int=1);
};

void A::show(int p)
{
  for(int i=0;i<p;i++)
    cout<<"Hello\n";
}

void main()
{
  A A1;
  A1.show();          //default value taken
  A1.show(3);         //default value overridden
}
/*End of memFuncDefault.cpp*/
```

Output

```
Hello
Hello
Hello
Hello
```

Listing 2.17 Giving default values to arguments of member functions

Again, it has to be kept in mind that a member function should be overloaded with care if default values are specified for some or all of its formal arguments. For example, the compiler will report an *ambiguity error* when it finds the second prototype for the 'show()' function of class 'A' in the following listing.

```
class A
{
  public:
    void show();
    void show(int=0);     //ambiguity error
};
```

Listing 2.18 Giving default values to arguments of overloaded member functions can lead to ambiguity errors

Reasons for such ambiguity errors have already been explained in the section on function overloading in Chapter 1. As in the case of non-member functions, if default values are specified for more than one formal argument, they must be specified from the right to the left. Similarly, default values must be specified in the function prototypes and not in function definitions. Further, default values can be specified for a formal argument of any type.

Inline Member Functions

Member functions are made inline by either of the following two methods.

- By defining the function within the class itself (as in Listing 2.5)
- By only prototyping and not defining the function within the class. The function is defined outside the class by using the scope resolution operator. The definition is prefixed by the `inline` keyword. As in non-member functions, the definition of the inline function must appear before it is called. Hence, the function should be defined in the same header file in which its class is defined. The following listing illustrates this.

```
/*Beginning of memInline.cpp*/
class A
{
  public:
    void show();
};

inline void A::show()  //definition in header file itself
{
  //definition of A::show() function
}
/*End of memInline.cpp*/
```

Listing 2.19 Inline member functions

Constant Member Functions

Let us consider this situation. The library programmer desires that one of the member functions of his/her class should not be able to change the value of member data. This function should be able to merely read the values contained in the data members, but not change them. However, he/she fears that while defining the function he/she might accidentally write the code to do so. In order to prevent this, he/she seeks the compiler's help. If he/she declares the function as a *constant function*, and thereafter attempts to change the value of a data member through the function, the compiler throws an error.

Let us consider the class 'Distance'. The 'Distance::getFeet()', 'Distance::getInches()' and the 'Distance::add()' functions should obviously be constant functions. They should not change the values of 'iFeet' or 'fInches' members of the invoking object even by accident.

Member functions are specified as constants by *suffixing the prototype and the function definition header with the* `const` *keyword.* The modified prototypes and definitions of the member functions of the class 'Distance' are as follows.

```
/*Beginning of Distance.h*/
/*Header file containing the definition of the Distance
class*/
class Distance
{
    int iFeet;
    float fInches;
  public:
    void setFeet(int);
    int getFeet() const;            //constant function
    void setInches(float);
```

```
    float getInches() const;        //constant function
    Distance add(Distance) const;   //constant function
};
/*End of Distance.h*/

/*Beginning of Distlib.cpp*/
/*Implementation file for the class Distance*/
#include"Distance.h"

void Distance::setFeet(int x)
{
  iFeet=x;
}
int Distance::getFeet() const       //constant function
{
  iFeet++;           //ERROR!!
  return iFeet;
}

void Distance::setInches(float y)
{
  fInches=y;
}

float Distance::getInches() const  //constant function
{
  fInches=0.0;       //ERROR!!
  return fInches;
}

Distance Distance::add(Distance dd) const     //constant
                                              //function
{
  Distance temp;
  temp.iFeet=iFeet+dd.iFeet;
  temp.setInches(fInches+dd.fInches);
  iFeet++;           //ERROR!!
  return temp;
}
/*End of Distlib.cpp*/
```

Listing 2.20 Constant member functionsc

For constant member functions, the memory occupied by the invoking object is a read-only memory. How does the compiler manage this? For constant member functions, the 'this' pointer becomes 'a constant pointer to a constant' instead of only 'a constant pointer'. For example, the 'this' pointer is of type `const` 'Distance *' `const` for the

'Distance::getFeet()', 'Distance::getInches()' and 'Distance::add()' functions. For the other member functions of the class 'Distance', the 'this' pointer is of type 'Distance *' const.

Clearly, only constant member functions can be called with respect to constant objects. Non-constant member functions cannot be called with respect to constant objects. However, constant as well as non-constant functions can be called with respect to non-constant objects.

Mutable Data Members

A mutable data member is *never* constant. It can be modified inside constant functions also. Prefixing the declaration of a data member with the keyword `mutable` makes it `mutable`. The following listing illustrates this.

```
/*Beginning of mutable.h*/
class A
{
    int x;          //non-mutable data member
    mutable int y;  //mutable data member

  public:

    void abc() const      //a constant member function
    {
      x++;      //ERROR: cannot modify a non-constant data
                //member in a constant member function
      y++;      //OK: can modify a mutable data member in a
                //constant member function
    }

    void def()            //a non-constant member function
    {
      x++;      //OK: can modify a non-constant data member
                //in a non-constant member function
      y++;      //OK: can modify a mutable data member in a
                //non-constant member function
    }
};
/*End of mutable.h*/
```

Listing 2.21 Mutable data members

We frequently need a data member that can be modified even for constant objects. Suppose, there is a member function that saves the data of the invoking object in a disk file.

Obviously, this function should be declared as a constant to prevent even an inadvertent change to data members of the invoking object. If we need to maintain a flag inside each object that tells us whether the object has already been saved or not, such a flag should be modified within the above constant member function. Therefore, this data member should be declared a mutable data member.

Friends

A class can have global non-member functions and member functions of other classes as friends. Such functions can directly access the private data members of objects of the class.

Friend Non-member Functions

A friend function is a non-member function that has special rights to access private data members of any object of the class of whom it is a friend. In this section, we will study only those friend functions that are not member functions of some other class.

A friend function is prototyped within the definition of the class of which it is intended to be a friend. The prototype is prefixed with the keyword `friend`. Since it is a non-member function, it is defined without using the scope resolution operator. Moreover, it is not called with respect to an object. An illustrative example follows:

```
/*Beginning of friend.cpp*/
class A
{
    int x;
  public:
    friend void abc(A&); //prototype of the friend function
};

void abc(A& AObj)    //definition of the friend function
{
  AObj.x++;      //accessing private members of the object
}

void main()
{
  A A1;
  abc(A1);
}
/*End of friend.cpp*/
```

Listing 2.22 Friend functions

A few points about the friend functions that we must keep in mind are as follows:

- `friend` keyword should appear in the prototype only and not in the definition.
- Since it is a non-member function of the class of which it is a friend, it can be prototyped in either the private or the public section of the class.
- A friend function takes one extra parameter as compared to a member function that performs the same task. This is because it cannot be called with respect to any object. Instead, the object itself appears as an explicit parameter in the function call.
- We need not and should not use the scope resolution operator while defining a friend function.

There are situations where a function that needs to access the private data members of the objects of a class cannot be called with respect to an object of the class. In such situations, the function must be declared as a friend. We will encounter one such situation in the chapter on 'Operating Overloading'.

Friend functions do not contradict the principles of OOPS. Since it is necessary to prototype the friend function inside the class itself, the list of functions that can access the private members of a class's object remains well defined and restricted. The benefits provided by data hiding are not compromised by friend functions.

Friend Classes

A class can be a friend of another class. *Member functions of a friend class can access private data members of objects of the class of which it is a friend.* If class B is to be made a friend of class A, then the statement

```
friend class B;
```

should be written within the definition of class A. The following example (Listing 2.23) illustrates this.

```
class A
{
  friend class B;    //declaring B as a friend of A
  /*
    rest of the class A
  */
};
```

Listing 2.23 Declaring friend classes

It does not matter whether the statement declaring class B as a friend is mentioned within the private or the public section of class A. Now, member functions of class B can access the private data members of objects of class A. Listing 2.24 exemplifies this.

```
/*Beginning of friendClass.cpp*/
class B;  //forward declaration... necessary because
          //definition of class B is after the statement
          //that declares class B a friend of class A.
class A
{
    int x;
  public:
    void setx(const int=0);
    int getx()const;
    friend class B; //declaring B as a friend of A
};

class B
{
    A * APtr;
  public:
    void Map(const A * const);
    void test_friend(const int);
};

void B::Map(const A * const p)
{
  APtr = p;
}

void B::test_friend(int i)
{
  APtr->x=i;   //accessing the private data member
}
/*End of friendClass.cpp*/
```

Listing 2.24 Effect of declaring a friend class

As we can see, member functions of class B are able to access private data member of objects of the class A although they are not member functions of class A. This is because they are member functions of class B that is a friend of class A.

Friendship is not transitive. For example, consider Listing 2.25.

```
class B;
class C;
/*Beginning of friendTran.cpp*/
class A
{
    friend class B;
    int a;
};
class B
{
    friend class C;
};

class C
{
    void f(A * p)
    {
      p->a++;  //error: C is not a friend of A
               //despite being a friend of a friend
    }
};
/*End of friendTran.cpp*/
```

Listing 2.25 Friendship is not transitive

Friend Member Functions

How can we make some specific member functions of one class friendly to another class? For making only 'B::test_friend()' function a friend of class A, replace the line

```
friend class B;
```

in the declaration of the class A with the line

```
friend void B::test_friend();
```

The modified definition of the class A is:

```
class A
{
  /*
    rest of the class A
  */
    friend void B::test_friend();
};
```

However, in order to compile this code successfully, the compiler should first see the definition of the class B. Otherwise, it does not know that 'test_friend()' is a member function of the class B. This means that we should put the definition of class B before the definition of class A.

However, a pointer of type A * is a private data member of class B. So, the compiler should also know that there is a class A before it compiles the definition of class B. This problem of *circular dependence* is solved by forward declaration. This is done by inserting the line

```
class A;        //Declaration only! Not definition!!
```

before the definition of class B. Now, the declarations and definitions of the two classes appear as follows.

```
/*Beginning of friendMemFunc.h*/
class A;

class B
{
    A * APtr;
  public:
    void Map(const A * const);
    void test_friend(const int=0);
};

class A
{
    int x;
  public:
    friend void B::test_friend(const int=0);
};
/*End of friendMemFunc.h*/
```

Listing 2.26 Forward declaring a class that requires a friend

Another problem arises if we try to define the 'B::test_friend()' function as an inline function by defining it within class B itself.

```
class B
{
  /*
```

```
    rest of the class B
  */
  public:
    void test_friend(const int p)
    {
      APtr->x=p;     //will not compile
    }
};
```

Listing 2.27 Problem in declaring a friend member function inline

But how will the code inside 'B::test_friend()' function compile? The compiler will not know that there is a data member 'x' inside the definition of class A. For overcoming this problem, merely prototype 'B::test_friend()' function within class B; *define it as inline* after the definition of class A in the header file itself. The revised definitions appear as follows.

```
/*Beginning of friendMemFuncInline.h*/
class A;

class B
{
    A * APtr;
  public:
    void Map(const A * const);
    void test_friend(const int=0);
};

class A
{
    int x;
  public:
    friend void B::test_friend(const int=0);
};

inline void B::test_friend(const int p)
{
  APtr->x=p;
}
/*End of friendMemFuncInline.h*/
```

Listing 2.28 Declaring a friend member function inline

Friends as Bridges

Friend functions can be used as bridges between two classes.

Suppose there are two unrelated classes whose private data members need a simultaneous update through a common function. This function should be declared as a friend to both the classes.

```
class B;  //forward declaration

class A
{
  /*
    rest of the class A
  */
  friend void ab(const A&, const B&);
};

class B
{
  /*
    rest of the class B
  */
  friend void ab(const A&, const B&);
};
```

Listing 2.29 Friends as bridges

Static Members

Static Member Data

Static data members hold global data that is common to all objects of the class. Examples of such global data are

- count of objects currently present,
- common data accessed by all objects, etc.

Let us consider class 'Account'. We want all objects of this class to calculate interest at the rate of say 4.5%. Therefore, this data should be globally available to all objects of this class (Listing 2.30).

This data cannot and should not be a member of the objects themselves. Otherwise, multiple copies of this data will be embedded within the objects taking up unnecessary space. Same value would have to be maintained for this data in all objects. This is very difficult. Thus, this data cannot be stored in a member variable of class 'Account'.

At the same time, this data should not be stored in a global variable. Then the data is liable to be changed by even non-member functions. It will also potentially lead to name conflicts. However, this means that it should be stored in a member variable of class 'Account'!

How can this conflict be resolved? Storing the data in a *static variable* of the class resolves this conflict. Static data members are members of the *class* and not of any *object* of the class, that is, they are not contained inside any object.

We prefix the declaration of a variable within the class definition with the keyword `static` to make it a static data member of the class.

```
/*Beginning of Account.h*/
class Account
{
    static float interest_rate;     //a static data member
    /*
      rest of the class Account
    */
};
/*End of Account.h*/
```

Listing 2.30 Declaring a static data member

A statement declaring a static data member inside a class will obviously not cause any memory to get allocated for it. Moreover, memory for a static data member will not get allocated when objects of the class are declared. This is because a static data member is not a member of any object. Therefore, we must not forget to write the statement to define (allocate memory for) a static member variable. Explicitly defining a static data member outside the class is necessary. Otherwise, the linker produces an error. The following statement allocates memory for 'interest_rate' member of class 'Account'.

```
float Account::interest_rate;
```

The above statement initializes 'interest_rate' to zero. If some other initial value (say 4.5) is desired instead, the statement should be rewritten as follows.

```
float Account::interest_rate=4.5;
```

Static data members should be defined in the implementation files only. The header file is included in both the implementation file and the driver program. If a static data member is defined in the header file, the static data member's definition would be in two files—the library file created from the implementation file and the object file created from the driver program. But in order to get the executable, the linker will have to link these files. Upon finding two definitions of the static data member, the linker would throw an error.

Making static data members private prevents any change from non-member functions as only member functions can change the values of static data members.

Introducing static data members does not increase the size of objects of the class. Static data members are not contained within objects. There is only one copy of the static data member in the memory. Let us try the following program to find out.

```
/*Beginning of staticSize.cpp*/

class A
{
    int x;
    char y;
    float z;
    static float s;
};

float A::s=1.1;

void main()
{
  cout<<sizeof(A)<<endl;
}

/*End of staticSize.cpp*/
```

Output

9

Listing 2.31 Static data members are not a part of objects

Static data members can be of any type. For example, name of the bank that has the accounts can be stored as a character array in a static data member of the class as follows.

```
/*Beginning of Account.h*/

class Account
{
    static float interest_rate;
    static char name[30];
```

```
    /*
      rest of the class Account
    */
};

/*End of Account.h*/

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

float A::interest_rate=4.5;
char A::name[30]="The Rich and Poor Bank";
/*
  definitions of the rest of the functions of class Account
*/

/*End of Account.cpp*/
```

Listing 2.32 Static data member can be of any type

Static data members of integral type can be initialized within the class itself if the need arises. For example,

```
/*Beginning of Account.h*/

class Account
{
    static int nameLength=30;
    static char name[nameLength];
    /*
      rest of the class Account
    */
};

/*End of Account.h*/

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

int A::nameLength;
char A::name[nameLength]="The Rich and Poor Bank";
/*
  definitions of the rest of the functions of class Account
*/

/*End of Account.cpp*/
```

Listing 2.33 Initializing integral static data members within the class itself

We must notice that the static data member that has been initialized inside the class must be still defined outside the class to allocate memory for it. Once the initial value has been supplied within the class, the static data member must not be re-initialized when it is defined.

Non-integral static data members cannot be initialized like this. For example,

```
/*Beginning of Account.h*/

class Account
{
    static char name[30]="The Rich and Poor Bank";//error!!
    /*
      rest of the class Account
    */
};

/*End of Account.h*/
```

Listing 2.34 Non-integral static data members cannot be initialized within the class

In Listing 2.33, the variable 'nameLength' is referred to directly without the class name and the scope resolution operator while defining the variable 'name'. One static data member can directly refer to another without using the scope resolution operator.

Member functions can refer to static data members directly. An example follows.

```
/*Beginning of Account.h*/

class Account
{
    static float interest_rate;
  public:
    void updateBalance();
    /*
      rest of the class Account
    */
};

/*End of Account.h*/

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

float Account::interest_rate=4.5;
```

```
void Account::updateBalance()
{
  if(end_of_year)
    balance+=balance*interest_rate/100;
}

/*
  definitions of the rest of the functions of class Account
*/

/*End of Account.cpp*/
```

Listing 2.35 Accessing static data members from non-static member functions

The 'object-to-member access operator' can be used to refer to the static data member of a class with respect to an object. The class name with the scope resolution operator can do this directly.

```
f=a1.interest_rate; //a1 is an object of the class Account
f=Account::interest_rate;
```

There are some things static data members can do but non-static data members cannot.

- A static data member can be of the *same type* as the class of which it is a member.

```
class A
{
    static A A1;        //OK : static
    A * APtr;           //OK : pointer
    A A2;               //ERROR!! : non - static
};
```

Listing 2.36 Static data members can be of the same type as their class

- A static data member can appear as the *default value* for the formal arguments of member functions of its class.

```
class A
{
    static int x;
    int y;
```

```
  public:
    void abc(int=x);   //OK
    void def(int=y);   //ERROR!! : object required
};
```

Listing 2.37 A static data member can appear as the default argument in the member functions

A static data member can be declared to be a constant. In that case, the member functions will be able to only read it but not modify its value.

Static Member Functions

How do we create a member function that need not be called with respect to an existing object? This function's sole purpose is to access and/or modify static data members of the class. Static member functions fulfill the above criteria. Prefixing the function prototype with the keyword `static` specifies it as a static member function. However, the keyword static should not reappear in the definition of the function.

Suppose there is a function 'set_interest_rate()' that sets the value of the 'interest_rate' static data member of class 'Account'. The application programmer should be able to call this function even if no objects have been declared. As discussed previously, this function should be static. Its definition can be as follows.

```
/*Beginning of Account.h*/
class Account
{
    static float interest_rate;
  public:
    static void set_interest_rate(float);
    /*
      rest of the class Account
    */
};
/*End of Account.h*/

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

float Account::interest_rate = 4.5;

void Account::set_interest_rate(float p)
{
  interest_rate=p;
}
```

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

Listing 2.38 Static member function

Now, the 'Account::set_interest_rate()' function can be called directly without an object.

```
Account::set_interest_rate(5);
```

Static member functions do not take the 'this' pointer as a formal argument. Therefore, accessing non-static data members through a static member function results in compile-time errors. *Static member functions can access only static data members of the class.*

Static member functions can still be called with respect to objects.

```
a1.set_interest_rate(5); //a1 is an object of the class
                         //Account
```

2.3 Objects and Functions

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. The following examples illustrate all this.

```
/*Beginning of Distance.h*/
class Distance
{
  public:
    /*function to add the invoking object with another
    object passed as a parameter and return the resultant
    object*/
    Distance add(Distance);
    /*
      rest of the class Distance
    */
};
/*End of Distance.h*/

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

```
Distance Distance::add(Distance dd)
{
  Distance temp;
  temp.iFeet=iFeet+dd.iFeet;
  temp.setInches(fInches+dd.fInches);
  return temp;
}
/*
  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,d3;
  d1.setFeet(5);
  d1.setInches(7.5);
  d2.setFeet(3);
  d2.setInches(6.25);
  d3=d1.add(d2);
  cout<<d3.getFeet()<<" "<<d3.getInches()<<endl;
}

/*End of Distmain.cpp*/
```

Output

9 1.75

Listing 2.39 Returning class objects

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

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

```
Distance& larger(const Distance& dd1,const 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

Listing 2.40 Returning class objects by reference

2.4 Objects and Arrays

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

Arrays of Objects

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

```
/*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

Listing 2.41 Array of objects

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 following class definition illustrates this.

```
#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];
}
```

Listing 2.42 Arrays inside objects

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.

```
/*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*/
```

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

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.

```
/*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*/
```

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

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. The following listing shows how this is done.

```
/*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*/
```

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

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.

```
/*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*/
```

Listing 2.46 Cumbersome long names for namespace

Assigning a suitably short alias to such a long namespace name solves the problem.

```
/*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*/
```

Listing 2.47 Providing an alias for a namespace

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 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 the following example, class B is defined in the private section of class A.

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

Listing 2.48 Nested classes

In the following example, class B is defined in the public section of class A.

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

Listing 2.49 A public nested class

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 the above two listings (Listing 2.48 and Listing 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.

```
/*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
```

Listing 2.50 Size of objects of the enclosing class

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 the following example.

```
/*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*/
```

Listing 2.51 Defining member functions of nested classes

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.

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

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

Listing 2.52 Defining a nested class outside the enclosing class

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.

```
/*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*/
```

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

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.

```
/*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*/
```

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

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.

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.
    (i) Structures in C++ can have member functions also.
    (ii) Structure members are private by default.
    (iii) The 'this' pointer is always a constant pointer.
    (iv) Member functions cannot be overloaded.
    (v) Default values can be given to the formal arguments of member functions.
    (vi) Only constant member function can be called for constant objects.
    (vii) The keyword `friend` should appear in the prototype as well as the definition of the function that is being declared as a friend.
    (viii) A friend function can be prototyped in only the public section of the class.
    (ix) Friendship is not transitive.
    (x) A static data member can be of the same type as the class of which it is a member.
    (xi) The size of objects of an enclosing class is affected by the presence of nested classes.
    (xii) An object of the nested class can be used in any of the member functions of the enclosing class without the scope resolution operator.
    (xiii) An object of the nested class cannot be a member of the enclosing class.
    (xiv) 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 associated 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.

Chapter 3

Dynamic Memory Management

OVERVIEW

This chapter explains the use of tools that are available in C++ for dynamic memory management. It begins with a brief explanation of static memory management and its limitation. This is followed by an elucidation of the mechanism of dynamic memory management.

The middle portion of the chapter deals with the use and usage of the new operator and the delete operator. Methods for allocating and deallocating memory for single objects and array of objects are explained.

The chapter also explains how the size of a dynamically allocated memory block is stored.

The last portion of the chapter explains the use of the 'set_new_handler()' function for specifying our own new handler.

3.1 Introduction

Let us have an overview of static memory management. Memory for program variables gets allocated and deallocated during run time only. For example,

we write

```
int x;
```

in some function in the source code. When the source code containing this statement (apart from the other statements) is compiled and linked, an executable file is generated. Besides containing equivalent instructions for the other statements, the executable file also contains the equivalent instructions for this statement. When the executable file is executed, all the instructions contained inside it, including the ones to allocate memory for 'x', are executed. *Thus, memory gets allocated for 'x' during run time.* This is known as static memory allocation (although memory gets allocated during run time only).

The compiler writes instructions in the executable to deallocate the memory previously allocated for 'x' when it encounters the end of the function, in which 'x' was declared, in the source code. When the executable file is executed, all instructions contained inside it including the ones to deallocate memory for 'x' are executed. *Thus, memory for 'x' gets deallocated during run time.* This is known as static memory deallocation (although memory gets deallocated during run time only).

Static allocation and deallocation of memory has a limitation. It is rigid. The programmers are forced to predict the total amount of data the program will utilize. They write statements to declare pre-calculated amounts of memory. During run time, if more memory is required, static memory allocation cannot fulfill the need. Once a certain memory block is no longer of any use to the program, memory allocated to it cannot be released immediately. The memory will continue to be held up until the end of the block in which the variable was created.

Dynamic memory management is a feature provided and supported in C and C++. It overcomes the drawbacks of static memory allocation. Just like in static memory allocation and deallocation, in dynamic memory allocation and deallocation also, memory gets allocated and deallocated during *run time* only. However, the decisions to do so can be taken *dynamically in response to the requirements arising during run time itself.*

If the program is running and the user indicates the need to feed in more data, a memory block sufficient to hold the additional amount of data is immediately allocated. For this, code utilizing the relevant functions and operators provided by C and C++ has to be explicitly written in the source code. Again, once a certain block of memory is no longer required, it can immediately be returned to the OS. For this again, code utilizing the relevant functions and operators provided by C and C++ has to be explicitly written in the source code. The OS can then allocate the deallocated memory block if the need arises.

3.2 Dynamic Memory Allocation

Dynamic memory allocation is achieved in C through the 'malloc()', 'calloc()', and 'realloc()' functions. In C++, it is achieved through the new operator. An illustrative example and its explanation follow.

```
/*Beginning of dynamic.cpp*/
#include<iostream.h>
void main()
{
  int * iPtr;
  iPtr=new int;
  *iPtr=10;
  cout<<*iPtr<<endl;
}
/*End of dynamic.cpp*/
```

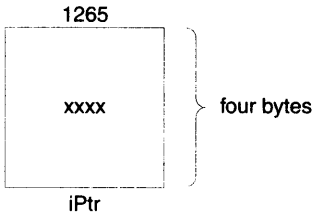
Output

10

Listing 3.1 Using the new operator for dynamic memory allocation

The word new is a keyword in C++. It is an operator. It takes a predefined data type as an operand (int in Listing 3.1). It then allocates memory to hold one value of the data type that is passed as a parameter to it in the heap (four bytes in Listing 3.1). Finally, it returns the address of the allocated block. This address need not be explicitly typecast since the new operator returns the address with the correct cast (int * in this case). This address can then be stored in a pointer of an appropriate type ('iPtr' in this call). The allocated block of memory can then be accessed through the pointer.

Statement: `int * iptr;`



Four bytes get allocated for **iPtr** containing junk value at the bytes with addresses from **1265** to **1268** (say).

Diagram 3.1(i) Dynamic memory allocation

Statement: `iPtr = new int;`

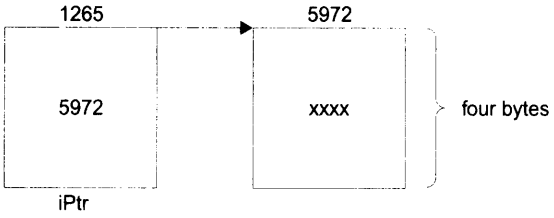


Diagram 3.1(ii) Dynamic memory allocation

The `new` operator allocates memory in the heap to hold one integer type value. Suppose, the block from the byte with address 5972 to the byte with address 5975 gets allocated. The `new` operator returns the base address of the block (5972). This value gets stored in 'iPtr'.

Statement: `*iPtr = 10;`

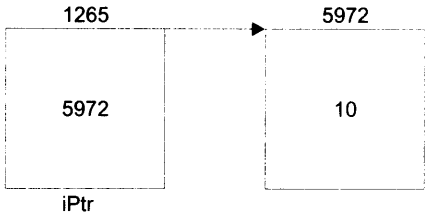
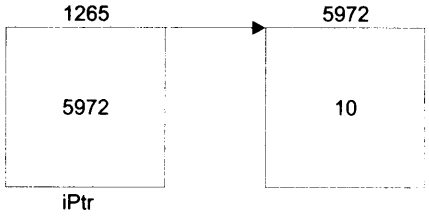


Diagram 3.1(iii) Dynamic memory allocation

'iPtr' is dereferenced and the value 10 gets written into the memory block of four bytes at which 'iPtr' points (5972 to 5975).

Statement: `cout<<*iPtr<<endl;`



'iPtr' is again dereferenced and the value (10) stored in the memory block to which 'iPtr' points (5972 to 5975) is read.

Diagram 3.1(iv) Dynamic memory allocation using the new operator

The general syntax of the new operator is:

```
<pointer> = new <data_type>;
```

The `new` operator can be used to create multiple blocks of memory also. This is shown in the following program (Listing 3.2).

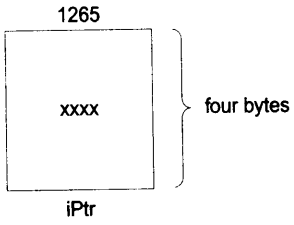
```
/*Beginning of DynArray1.cpp*/
#include<iostream.h>
#define SIZE 10
void main()

{
  int * iPtr;
  iPtr = new int[SIZE];
  for(int i=0;i<SIZE;i++)
    iPtr[i]=i;              //can write cin>>iPtr[i]; also
  for(int j=0;j<SIZE;j++)
    cout<<iPtr[j]<<endl;
}
/*End of DynArray1.cpp*/
```

Output

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

Listing 3.2 Creating an array dynamically using the new operator

Statement: `int * iPtr;`



Four bytes get allocated for 'iPtr' containing junk value at the bytes with addresses from, say, 1265 to 1268.

Diagram 3.2(i) Memory allocation for an array using 'iPtr'

Statement: `iPtr = new int[SIZE]; //SIZE=10`

The new operator allocates memory in the heap to hold ten integer type values [see Diagram 3.2(ii)]. If the block from the byte with address 5972 to the byte with address 6012 gets allocated, the new operator returns the base address of the block 5972. This value gets stored in 'iPtr'. After this, 'iPtr' is simply dereferenced within the for loop by using the subscript operator. All the elements of the array at whose first element the

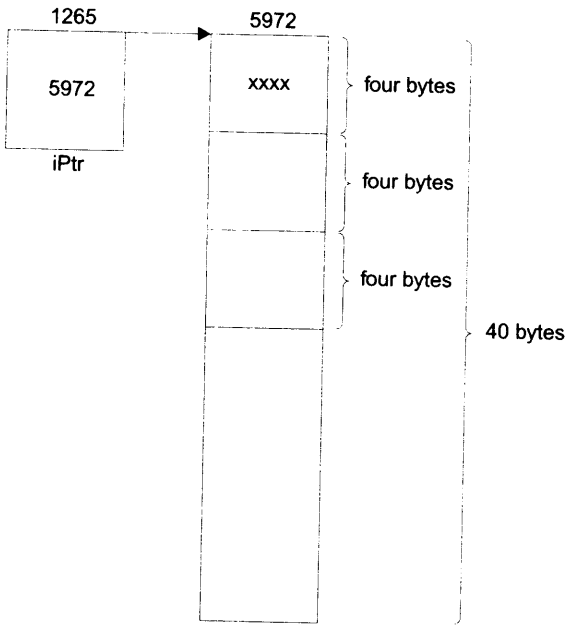


Diagram 3.2(ii) Dynamically allocating memory for an array using the new operator

pointer is pointing are accessed. The syntax for using the new operator to create an array is as follows:

```
<pointer> = new <data_type>[<number_of_elements>];
```

Now, let us make the program interactive to exploit the power of the new operator. The value that we passed inside the subscript while allocating the memory using the new operator can be that of a variable. In Listing 3.3, we will first ask the user to enter the size of the array and store it in a variable. Next, we will pass the variable into the subscript while using the new operator to allocate memory. The address returned by the new operator will then be stored in a pointer. Finally, we will access the array thus created through the pointer. The program is shown Listing 3.3.

```
/*Beginning of DynArray2.cpp*/
#include<iostream.h>
void main()
```

```
{
  int * iPtr;
  unsigned int iSize;
  cout<<"Enter the size of the array : ";
  cin>>iSize;
  iPtr = new int[iSize];
  for(int i=0;i<iSize;i++)
  {
    cout<<"Enter the value for element "<<i+1<<" : ";
    cin>>iPtr[i];
  }
  for(int j=0;j<iSize;j++)
    cout<<iPtr[j]<<endl;
}
/*End of DynArray2.cpp*/
```

Output

Enter the size of the array : **3***<enter>*
Enter the value for element 1 : **12***<enter>*
Enter the value for element 2 : **7***<enter>*
Enter the value for element 3 : **19***<enter>*
12
7
19

Listing 3.3 Creating an array dynamically when its size is specified during run time

We must note that the `new` operator has enabled us to allocate memory *dynamically*. In Listing 3.3, memory is getting allocated during run time (just like in static memory allocation). However, the amount of memory to be allocated is being decided during run time itself.

Same methodology can be applied for dynamically creating arrays of the other predefined fundamental data types. Arrays of class objects can also be created dynamically in the same way. The following program (Listing 3.4) is a case in point.

```
/*Beginning of DynDist.cpp*/
#include<iostream.h>
#include"Distance.h"
void main()
{
  Distance * dPtr;
  unsigned int iSize;
  cout<<"Enter the number of elements : ";
  cin>>iSize;
```

```
    dPtr = new Distance[iSize];
    for(int i=0;i<iSize;i++)
    {
      cout<<"Enter the feet : ";
      cin>>a;
      cout<<"Enter the inches : ";
      cin>>b;
      dPtr[i].setFeet(a);
      dPtr[i].setInches(b);
    }
    for(int j=0;j<iSize;j++)
    {
      cout<<dPtr[j].getFeet()<<" "
          <<dPtr[j].getInches()<<endl;
    }
}
/*End of DynDist.cpp*/
```

Output

Enter the number of elements : **3**<*enter*>
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

Listing 3.4 Creating an array of objects dynamically during run time

In Listings 3.3 and 3.4, the user is explicitly asked to enter the size of the array he/she wants to create. This is a little abrupt. Requirements for more memory may arise during run time in a more subtle fashion (say, while creating data structures such as linked lists, trees, etc.). Nevertheless, the basic technique of using the `new` operator remains the same.

3.3 Dynamic Memory Deallocation

We already know that a block of memory allocated dynamically can be deallocated dynamically. Once it is not in use any more, a dynamically allocated block of memory should definitely be returned to the OS.

In C, dynamic memory deallocation is achieved through the 'free()' function. Dynamically allocated blocks of memory can be returned to the OS in C++ through the `delete` operator.

What is the need to deallocate a dynamically allocated block of memory? What will happen if a dynamically allocated block of memory is not returned to the OS? These questions are answered by the following program (Listing 3.5) and the explanatory diagram (Diagram 3.3) that follows.

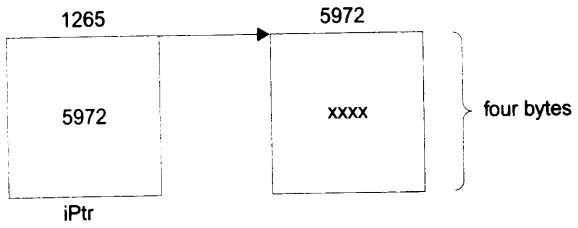
```
/*Beginning of memleak.cpp*/
#include<iostream.h>
void abc();
void main()
{
  abc();  //call to the abc() function
  /*
    rest of the main() function
  */
}
void abc()
{
  int * iPtr;
  iPtr = new int;
  /*
    rest of the abc() function
  */
}
/*End of memleak.cpp*/
```

Listing 3.5 Memory leak

The following statement executes from within the 'abc()' function which is called from the 'main()' function.

Statement: `iPtr = new int;`

As a result, the following scenario emerges.



The `new` operator allocates memory in the heap to hold one integer type value. Suppose the block from the byte with address 5972 to the byte with address 5975 gets allocated. The `new` operator returns the base address of the block 5972. This value gets stored in 'iPtr'.

After 'abc()' finishes execution, memory for 'iPtr' itself is deallocated. But, the memory in the heap area remains locked up as an orphan (unreferenced) locked up block of memory.

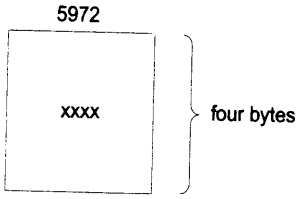


Diagram 3.3 Memory leak

As it can be seen from Diagram 3.3, after the 'abc()' function terminates, four bytes of memory are lost. Since they have not been returned to the OS, they remain locked up. This is known as a *memory leak*. If more memory is required, the OS will *not* allocate this block of memory. Moreover, this block of memory cannot be accessed since the only pointer ('iPtr') that was pointing at it has itself been removed from the stack.

This block of memory that is no longer of any use can and should be returned to the OS. A dynamically allocated block of memory can be deallocated by passing the pointer pointing to it as an operand to the `delete` operator. For example, the following statement should be inserted before the end of the 'abc()' function in Listing 3.5.

```
delete iPtr;
```

The foregoing statement is executed just before the 'abc()' function terminates. The memory block at which 'iPtr' points gets deallocated (it becomes available for the OS). Next, the memory allocated for 'iPtr' itself is deallocated. Finally, the function terminates. Thus, memory leak is prevented.

When the `new` operator is used, the OS blocks a block of memory of the requested size. The OS never allocates this particular block of memory in response to subsequent requests for memory blocks as long as this block of memory is not deallocated. When the `delete` operator is used on the pointer that points at this block of memory, the memory block gets deallocated, that is, freed and made available for the OS. In other words, the OS, in response to subsequent requests for memory blocks, may allocate this freed block of memory.

A dynamically allocated block of memory remaining locked up is frequently a blessing. The fact that the block of memory locked up by the code in a certain function persists even after the function terminates is frequently desirable. A called function may allocate a memory block and a pointer local to the calling function can be made to point at it. Even after the called function terminates, the dynamically allocated block of memory will remain persistent, but not unreferenced. The following code illustrates this.

```
void abc(int  ** p)
{
  /*
    some complex algorithm
  */

  *p = new int;

  /*
    rest of the abc() function
  */
}
void main()
{
  int * iPtr;
  abc(&iPtr);
  /*
    rest of the main() function
  */
}
```

Listing 3.6 Making a dynamically allocated block of memory available to the calling function

In Listing 3.6, the address of 'iPtr' that is local to the calling function ('main()' function) is passed as a parameter to the called function ('abc()' function). Its value needs to be changed by the 'abc()' function. Its address is stored in a double pointer (a pointer to a pointer has to be a double pointer). A block of memory is allocated and its address is stored in 'iPtr' by dereferencing the pointer that points at it. It is our obvious desire that the dynamically allocated block of memory persists even after the 'abc()' function terminates. After the 'abc()' function terminates, 'iPtr' that is a local variable in the calling function will point at the dynamically allocated block of memory.

The general syntax of the `delete` operator to deallocate a single block of memory is:

```
delete <pointer>;
```

In the foregoing listings, the memory block was deallocated only at the end of the functions that allocated it. However, dynamic memory deallocation is usually conditional.

```
void abc(int ** p)
{
  if(memory_not_required)
  {
    delete *p;
```

```
    *p = NULL;
  }
  /*
    rest of the abc() function
  */
}
```

Listing 3.7 C++ allows deallocation of memory as and when required

A misconception about the `delete` operator is due to the commonly used phrase 'deleting the pointer'. An uninitiated reader may think that the memory being occupied by the pointer itself gets removed if the `delete` operator is used on the pointer. In reality, nothing of this sort happens.

When the `delete` operator is used on a pointer, the pointer continues to occupy its own block of memory and continues to have the same value that is the address of the first byte of the block of memory that has just got deallocated. Thus, the pointer continues to point at the same block of memory. This will lead to run-time errors if the pointer is dereferenced.

We can see in Listing 3.7 that the pointer being pointed at by 'p' was deliberately nullified after the memory that the pointer was pointing at had been deallocated. This is a very common practice to indicate that the pointer (the pointer whose address is passed from the calling function in this case) no longer points at a valid dynamically allocated block of memory. In other words, it is highly desirable that *either the pointer points at a valid block of memory or be NULL*. It is not possible to ensure this due to the low level of representation of pointers. A pointer is unlikely to be NULL at the time of its creation. But that does *not* mean that the value it contains is the address of some valid allocated block of memory. *There is no guaranteed initialization of data*. This problem is solved by the use of constructors, which have been discussed in Chapter 4.

A multiple block of memory is deallocated by suffixing the `delete` operator with an empty pair of square brackets followed by the pointer that points at the multiple block of memory.

```
int * iPtr;
....
iPtr = new int[10];
....
delete[] iPtr;
```

Listing 3.8 Deallocating memory that was allocated for an array

If we write `delete` 'iPtr' instead of `delete[]` 'iPtr', only the first four bytes of the block of 40 bytes at which 'iPtr' is pointing, at will be deallocated. Using `delete[]` deallocates the entire block of 40 bytes. The syntax for using the 'delete' operator to deallocate an array is as follows:

```
delete [] <pointer>;
```

The size of the array to be created is passed as a parameter to the `new` operator. But while deallocating the memory allocated for the array, the size is not passed (the square brackets are empty). Then how does the compiler know how much of memory is to be deallocated? The answer is that when the `new` operator executes to allocate a block of array, the OS stores the size passed. The size of the memory block, which is captured during run-time, is prefixed to the memory block itself. When the `delete` operator is used followed by the empty pair of square brackets, the compiler uses the size stored and deallocates the entire block correctly.

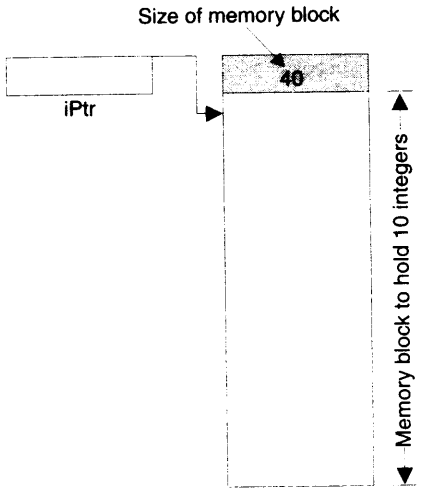


Diagram 3.4 Size of the allocated memory is prefixed to the memory block

Blocks of memory containing arrays of other types can also be deallocated similarly. For example,

```
Distance * dPtr;
dPtr = new Distance[5];  //creates an array of 5 objects of
                         //the class Distance
....
....
delete[] dPtr;           //de-allocates the memory
                         //allocated for the entire array
```

Listing 3.9 Deallocating memory that was allocated for an array of objects

3.4 The set_new_handler() function

We already know that the new operator attempts to capture more chunks of memory from the heap during run time. But, what happens if no more memory is available to satisfy this attempt? We get an out-of-memory condition.

The new operator, when faced with an out-of-memory condition, calls a global function and then throws an exception of type 'bad_alloc' (the chapter on exception handling deals with the mechanism of throwing and catching exceptions). This global function is known as the *new handler function*.

We can specify our own new handler function also. We can specify that the new operator, upon encountering an out-of-memory condition, calls a function of our choice. We can do this by calling the 'set_new_handler()' function and passing the name of the desired function as a parameter to it. The prototype of the 'set_new_handler()' function clarifies this. This prototype is in the 'new.h' header file.

```
new_handler set_new_handler(new_handler);
```

Obviously, 'new_handler' is a data type. It is a function pointer type. The formal argument of the 'set_new_handler()' function is a function pointer. If we pass the name of our desired function as a parameter to the 'set_new_handler()' function, all subsequent out-of-memory conditions cause the new operator to call it. Our desired function becomes the new handler. Moreover, when the 'set_new_handler()' function is called, it returns a pointer to the *previous new handler function*.

An illustrative example follows.

```
/*Beginning of newHandler.cpp*/
#define BIG_NUMBER 9999999
#include<new.h>          //for set_new_handler() function
void myNewHandler()
{
  /*
    code to handle out-of-memory condition
  */
}
void main()
{
   new_handler oldHandler;
   //set the function myNewHandler as the new handler

   oldHandler  = set_new_handler(myNewHandler);
   int * p = new int[BIG_NUMBER];  //probably cause out-of-
                                   //memory condition
}
/*End of newHandler.cpp*/
```

Listing 3.10 Specifying a new handler function

If the OS is unable to allocate the requested amount of memory, which is quite likely in Listing 3.10, the new operator fails. The new handler function gets called. The call to the 'set_new_handler()' function, just prior to the call to the new operator, has already set the function 'myNewHandler' as the new handler. Therefore, the function 'myNewHandler' gets called.

An important characteristic of the new operator is that when its request for memory fails, it calls the new handler function *repeatedly* until its request is satisfied. This fact helps in meaningfully defining the new handler function.

We can make the new handler function log an error message and then call the 'abort()' function.

```
void myNewHandler()
{
  //statement to log a suitable error message
  abort();
}
```

Listing 3.11 Defining the new handler function

The 'abort()' function simply terminates the program. We can also throw an exception from within the new handler function. The chapter on exception handling explains the syntax for throwing exceptions and its superiority over calling the 'abort()' function.

Another course of action is to replace the existing new handler function by another one. For this, we can call the 'set_new_handler()' function from within the existing new handler function and pass the name of the new handler as a parameter to it. Of course, such a call should be preceded by the code that attempts to resolve the out-of-memory condition first. The new handler should be replaced only if this attempt fails.

```
#include<new.h>
void myNewHandler()
{
  //make an attempt to resolve the out-of-memory
  //condition
  if(above_attempt_fails)
    set_new_handler(myAnotherNewHandler);
}
```

Listing 3.12 Replacing the existing new handler function

An interesting way of defining the new handler is to allocate some buffer memory in advance and free it part by part as the need arises.

Summary

Memory is allocated for program variables during *run time* only. In static memory allocation, the amount of memory to be allocated is decided during *compile time* itself. The instance at which each statically allocated variable would get created during the program's execution is also decided during the program's compilation.

On the other hand, the amount of memory to be allocated is decided during *run time* in case of dynamic memory allocation. Moreover, memory can be allocated in response to conditions that arise during *run time*.

C++ provides the 'new' operator for allocating memory dynamically. The syntax of the 'new' operator for allocating memory for a single block is:

```
<pointer> = new <data_type>;
```

The `new` operator allocates enough memory in the heap area to accommodate one variable of the data type that is passed as its right-hand-side operand. Further, it returns the address of the first byte of this allocated block of memory that can be stored in the pointer on the left-hand-side of the assignment operator as shown in the above statement.

Memory for an array can be allocated by using the `new` operator. The syntax is as follows:

```
<pointer> = new <data_type>[<number_of_elements>];
```

Again, dynamically allocated memory can be dynamically deallocated in response to conditions that arise during *run time*. Dynamically allocated memory must be deallocated, that is, returned to the Operating System. Otherwise, memory leak would occur.

C++ provides the `delete` operator for deallocating dynamically allocated memory. The syntax of the `delete` operator for deallocating memory earlier allocated for a single block is:

```
delete <pointer>;
```

The `delete` operator deallocates the memory in the heap area that the pointer that is passed as its right-hand-side operand points at.

Memory allocated dynamically for an array can also be deallocated by using the `delete` operator. The syntax is as follows:

```
delete [] <pointer>;
```

This version is similar to the previous one with the difference that an empty pair of square brackets appears between the `delete` keyword and the name of the pointer. C++ knows the exact number of bytes to be returned. It stores the *size* of the dynamically allocated block in a block of memory that it prefixes to the allocated block of memory itself. The 'set_new_handler()' function enables us to set a function of our choice as the new handler function.

Key Terms

static memory allocation

static memory deallocation

dynamic memory allocation

dynamic memory deallocation

new operator

delete operator

set_new_handler() function

new handler function

Exercises

1. What is static memory allocation?
2. When is memory allocated and deallocated in static memory allocation—during compile time, link time, or run time?
3. Under what conditions does static memory allocation become unsuitable?
4. What is dynamic memory allocation? How is it different from static memory allocation?
5. When is memory allocated and deallocated in dynamic memory allocation— during compile time, link time, or run time?
6. Under what conditions does the use of dynamic memory allocation become mandatory?
7. What is the syntax of the 'new' operator for
   (i) allocating memory for a single variable?
   (ii) allocating memory for an array?
8. Describe how additional blocks of memory can be captured in C++ during run time based upon existing run time conditions?

9. What is the syntax of the 'delete' operator for
   (i) deallocating memory that has been allocated for a single variable?
   (ii) deallocating memory that has been allocated for an array?
10. The size of the array, whose memory is to be deallocated, is not passed to the 'delete' operator. How does the compiler determine this size?
11. What is memory leak?
12. How can the `delete` operator be used to prevent a memory leak?
13. What is an out-of-memory condition?
14. What is the 'new handler'? How is the 'set_new_handler()' function used to set our own new handler?

Chapter 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 the following listing (Listing 4.1).

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

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

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

```
A A1;      //memory allocated for the object (four 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!
```

The 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.

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

Output

Constructor of class A called
End of program

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

Let us now define our own constructor for the class 'Distance'. What should the constructor do to the invoking the 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.

```
/*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

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

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 as follows. (See Diagram 4.1).

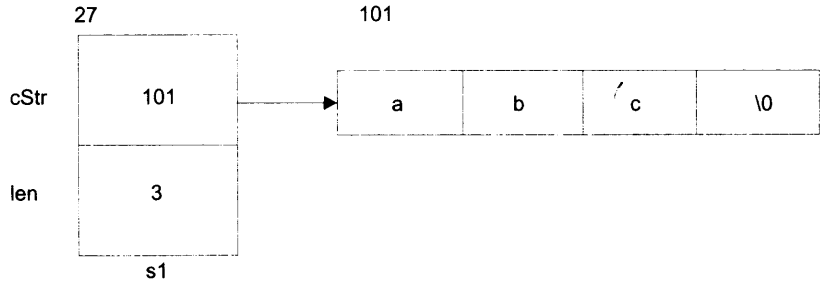


Diagram 4.1 Memory layout of an object of the class 'String'

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'.

1. '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.
2. There should be no memory leaks.

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.

```
/*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*/
```

Listing 4.4 A user-defined constructor

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 follows.

```
/*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 A user-defined parameterized constructor called by creating an object in the stack

---

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.

```
/*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

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

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.

```
/*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*/
```

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

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 as well as 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 following scenario should emerge.

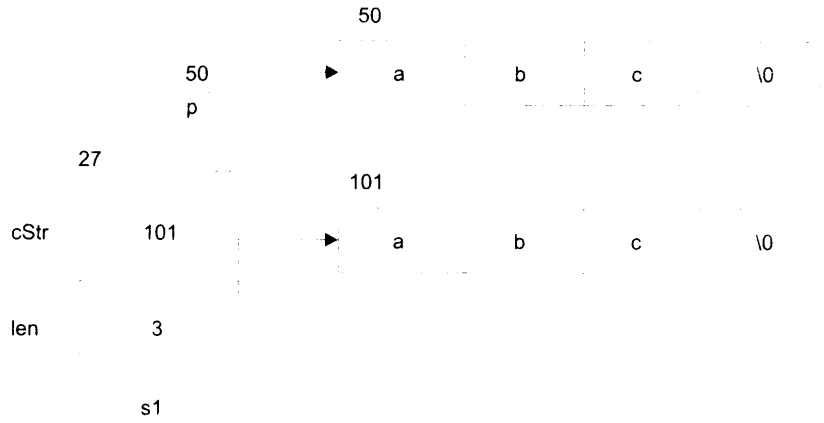


Diagram 4.2 Assigning a string to an object of the class 'String'

In Diagram 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 Diagram 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, 'p' and 'cPtr' both 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 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 Diagram 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

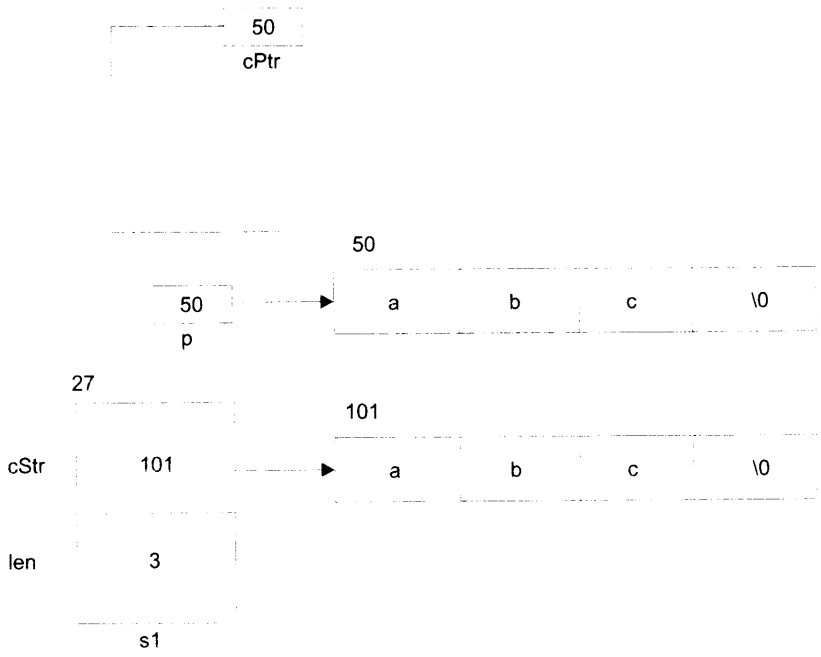


Diagram 4.3 Assigning a string to an object of the class 'String'

contains the base address of the memory block containing the actual contents of the array. This can be seen in Diagram 4.4.

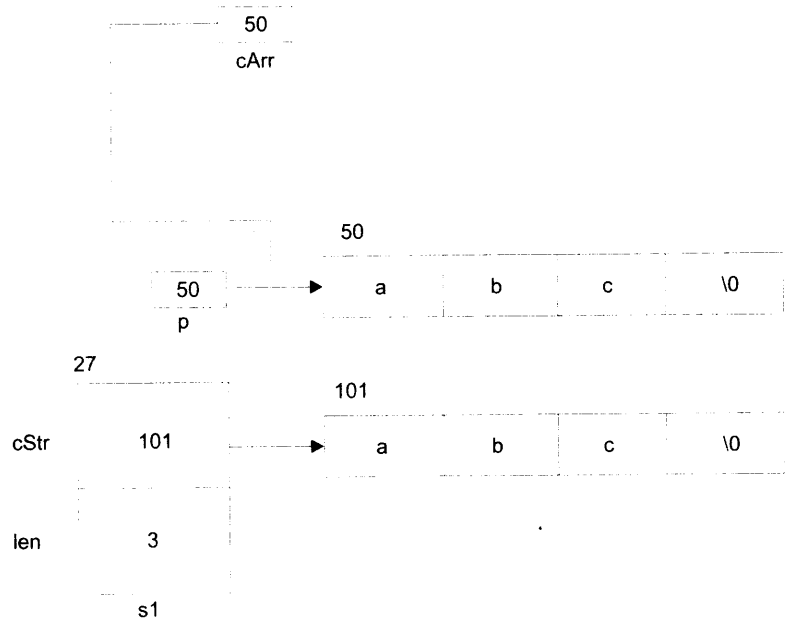


Diagram 4.4 Assigning an array to an object of the class 'String'

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 as follows.

```
/*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

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

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 Diagrams 4.2, 4.3, and 4.4.

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 behavior 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 follows.

```
/*Beginning of String.h*/
class String
{
    char * cStr;
    long unsigned int len;
  public:
    String(const String&);     //our own copy constructor
    /*
      rest of the class String
    */
};
/*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=strlen(ss.len);
    cStr = new char[len+1];    //...dynamically allocate a
                               //separate memory block
    strcpy(cStr,ss.cStr);      //...and copy into it
  }
}
/*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

Listing 4.9 A user-defined copy constructor

In the copy constructor (Listing 4.9), 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 the '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.

```
/*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

Listing 4.10 Destructor gets called for each object when the object is destroyed

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.

```
/*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*/
```

Listing 4.11 A user-defined destructor

4.3 The 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) 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
```

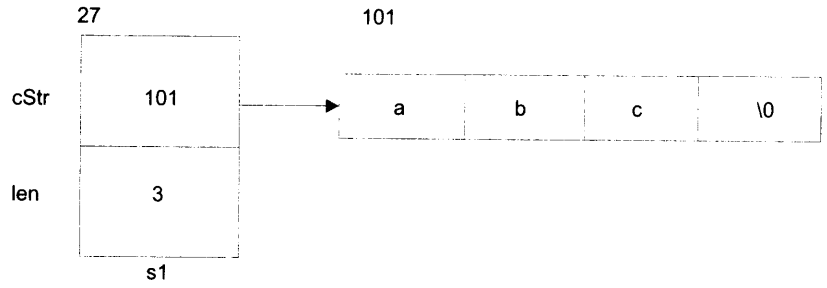
the following things should happen.

- 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).

The following shows adding a character to a stretchable string in the object-oriented way.

Before

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



After

```
s1.addChar('d');
```

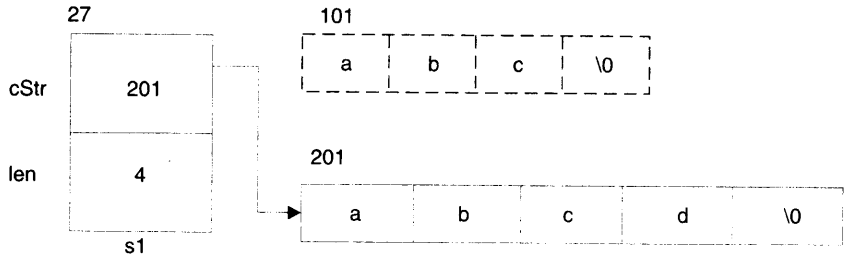


Diagram 4.5 Adding a character to a stretchable string—the object oriented way

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.