```
// interface functions set...()
void Person::setname( char const *str )
{
   if( name )
      delete name;
   name = new char[ strlen(str) + 1 ];
   strcpy( name, str );
}
void Person::setaddress( char const *str )
{
   if( address )
      delete address;
   address = new char[ strlen(str) + 1 ];
   strcpy( address, str );
}
void Person::setphone( char const *str )
{
   if( phone )
      delete phone;
   phone = new char[ strlen(str) + 1 ];
   strcpy( phone, str );
}
inline char const *Person::getname() const
{
   return name;
}
inline char const *Person::getaddress() const
{
   return address;
}
inline char const *Person::getphone() const
{
   return phone;
}
void printperson( Person const &p )
{
   if( p.getname() )
      cout << "Name    : " << p.getname() << endl;
   if( p.getaddress() )
      cout << "Address: " << p.getaddress() << endl ;
   if( p.getphone() )
      cout << "Phone   : " << p.getphone() << endl;
}
void main()
{
   Person p1, p2;
   p1.init();
   p2.init();
   p1.setname( "Rajkumar" );
   p1.setaddress( "E-mail: raj@cdacb.ernet.in" );
```

```
    p1.setphone( "90-080-5584271" );
    printperson( p1 );
    p2.setname( "Venugopal K R" );
    p2.setaddress( "Bangalore University" );
    p2.setphone( "-not sure-" );
    printperson( p2 );
    p1.clear();
    p2.clear();
}
```

Run

```
Name    : Rajkumar
Address: E-mail: raj@cdacb.ernet.in
Phone   : 90-080-5584271
Name    : Venugopal K R
Address: Bangalore University
Phone   : -not sure-
```

As illustrated in this program, the keyword `const` occurs following the argument list of functions. Again the following *Const-Rule* applies: *whichever appears before the keyword const must not alter its contents and if any attempt is made to alter data, the compiler issues an error message.* The same specification must be repeated in the definition of member functions:

```
char const *Person::getname() const
{
   return name;
}
```

A member function, which is declared and defined as `const`, should not alter any data fields of its class. In other words, a statement like

```
name = 0;
```

in the above `const` function `getname()` would lead to a compilation error.

The formal parameter to the function

```
void printperson( Person const &p )
```

is declared as a constant object. The private data members, by specification itself cannot be modified. If the object parameter is declared as const, even its public data members cannot be modified. Thus the function `printperson()` can only read public data members, but cannot modify them.

The purpose of `const` functions lies in the fact that C++ allows `const` objects to be created. For such objects only the `const` member, which does not modify them has to be called. The only exception to the rule are the constructors and destructors: these are called *automatically*. This feature is comparable to the definition of a variable `int const max=10;` such a variable may be initialized on its definition. Analogously, the constructor can initialize its object at the definition, but subsequent assignments cannot be performed. Generally, it is good to declare member functions which do not modify their object to be `const`.

10.16 Structures and Classes

Structures and classes in C++ are given the same set of features. For example, structures may also be used to group data as well as functions. In C++, the difference between structures and classes is that by default, structure members have public accessibility, whereas class members have private access control unless otherwise explicitly stated. The declaration for a structure in C++ is similar to a class specification. It is illustrated in the following declaration:

```
class complex

  private:           // private part
     float real;      // real part of complex number
     float imag;      // imaginary part of complex number
  public:                // public part
     void getdata();
     void outdata( char *msg );
     complex AddComplex( complex c2 );
};
```

A similar structure may be created as shown below:

```
struct complex
{
  private:           // private part
     float real;     // real part of complex number
     float imag;     // imaginary part of complex number
  public:            // public part
     void getdata();
     void outdata( char *msg );
     complex AddComplex( complex c2 );
};
```

The above declarations of class and structure can be written without any loss of meaning as follows:

```
class complex
{
  // by default private part, the keyword private is omitted
  float real;     // real part of complex number
  float imag;     // imaginary part of complex number
  public:         // public part
     void getdata();
     void outdata( char *msg );
     complex AddComplex( complex c2 );
};
```

Thus, in the absence of the keyword `private`, the members of a class are treated as private till another access-specifier keyword (private or public) is encountered. However, in a structure, the members are treated as public by default. It is illustrated in the following declaration:

```
struct complex
{
  // by default public, the keyword public is omitted
  void getdata();
  void outdata( char *msg );
  complex AddComplex( complex c2 );
  private:          // private part
     float real;    // real part of complex number
     float imag;    // imaginary part of complex number
};
```

Note: Most programmers prefer to use a class to group data as well as functions, a structure to group only data, following the conventions of C. It is advisable to use the keywords `private` and `public` explicitly in the declaration of classes and structures to improve readability of the program code.

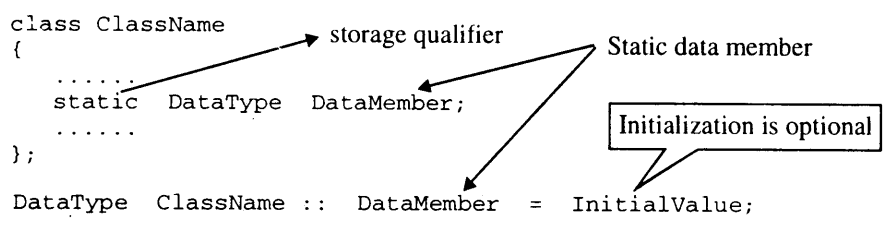
10.17 Static Data and Member Functions

Earlier examples of classes have shown that each object of a class has its own set of public or private data. Each public or private function then accesses the object's own version of the data. In some situations, it is desirable to have one or more common data fields, which are accessible to all objects of the class. An example of such a situation is keeping the status of how many objects of a class are created and how many of them are currently active in the program. Another example is a flag variable, which states whether some specific initialization has occurred; only the first object of the class performs the initialization and then sets the flag to *done*.

Such situations are analogous to C code, where several functions need to access the same variable. A common solution in C is to define all these functions in one source file and to declare the variable as static: the variable name is then not known beyond the scope of the source file. This approach is quite valid, but does not agree with the philosophy of one data or function per program having multiple source files. Another C-like solution is to create the variable in question with unusual names such as `__MYFLAG`, `_6ULDV8`, etc., with the hope that other parts of the program (libraries, link modules, etc.) do not make use by defining these variables by accident. Neither the first, nor the second C-like solution is elegant. C++ therefore allows static data and functions, which are common to all objects of a class.

Static Data Member Definition

In Turbo C++ version 1.0, static data members were not required to be explicitly defined. When the linker finds undefined static data, it would automatically define them and allocate storage for them instead of generating errors, but both new versions of Turbo C++ and Borland C++ insist on the explicit definition; no other way to define a static data exists. The syntax of defining static data member of a class is shown in Figure 10.22.



```
class ClassName
{
   ......
   static   DataType   DataMember;
   ......
};
DataType   ClassName ::   DataMember   =   InitialValue;
```

Figure 10.22: Static data member declaration in a class and its definition outside the class

The static data members can be initialized during their definition outside all the member functions, in the same way as global variables are initialized. The definition and initialization of a static data member usually occur in one of the source files of the class functions. The statement which defines and initializes the variable `MyClass::count` (`count` is a data member of `MyClass`) is always valid whether `count` is declared private, public or protected inside the class `MyClass`. The reason is that static data members accessed in this way are essentially global data.

Private static data members

When a data member is required to be accessible to more than one function, the normal procedure adopted in a function-oriented language is to declare it as an external variable. But this technique may be dangerous as it exposes external data variable to accidental modification, which may have undesirable effects on the efficient and reliable working of the program.

C++ provides an elegant solution to that problem in the form of static data members. The usual technique that is adopted is to declare the static data member in the private section of a class. Thus, effective data hiding is achieved, as the data is only accessible through the member functions, while providing access to all the objects of that class. This is illustrated in the program count.cpp.

```
// count.cpp: counts how many calls are made to a member function set()
#include <iostream.h>
class MyClass
{
   static int count; // static member
   int number;
   public:
      // initializes object's member and increments function call
      void set( int num )
      {
         number = num;
         ++count;
      }
      void show()
      {
          cout << "\nNumber of calls made to 'set()' through any object: "
               << count;
      }
};
// static member count is shared by all the objects of class MyClass
int MyClass::count = 0; // definition and initialization of a data member
void main()
{
   MyClass obj1;
   obj1.show();
   obj1.set( 100 );
   obj1.show();
   MyClass obj2, obj3;
   obj2.set( 200 );
   obj2.show();  //same result even with obj1.show and obj3.show();
   obj2.set( 250 );
   obj3.set( 300 );
   obj1.show();  //same result even with obj2.show and obj3.show();
}
```

Run

```
Number of calls made to 'set()' through any object: 0
Number of calls made to 'set()' through any object: 1
Number of calls made to 'set()' through any object: 2
Number of calls made to 'set()' through any object: 4
```

Omission of the statemcnt

```
int MyClass::count = 0;
```

in the above program would generate linking error although program is compiled successfully. This is because the statement in the class MyClass

```
static int count;
```

would not have been defined anywhere and it is a static variable within a class. Hence, an error would be generated if a value is assigned to count without any memory being allocated to it. It is possible to omit initialization of a static membei variable when it is defined, as shown below:

```
int MyClass::count;
```

Irrespective of whether the data member is private, public or protected, it must always be defined using the scope resolution operator. Static variables act like a bridge between objects of the same class. The linker allocates storage for a static member when the variable is defined even if no objects are actually created from the class.

Access Rules for Static Data Members

The public static data members can be accessed using the scope resolution operator or through objects with member access operator. Using the scope resolution operator is a completely new notation for member access. However, the accessibility of private static data members is same as that of normal private members.

The static data members which are declared public are similar to *normal* global variables. They can be addressed by the program by prefixing class name and scope resolution operator. It is illustrated in the following code fragment:

```
class Test
{
  public:
    static int public_int;
  private:
    static int private_int;
};
void main()
{
  Test::public_int = 145;  // ok
  Test::private_int = 12;  // wrong, do not touch the private data members
  Test myobj;
  myobj.public_int = 145;  // ok
  myobj.private_int = 12;  // wrong, do not access the private data member
}
```

The static data member public_int defined in the class Test can be accessed using the scope resolution operator prefixed by its class name as follows:

```
Test::public_int = 145;          // ok
```

Whereas, the data member private_int cannot be accessed using the scope resolution operator. Therefore, the statement

```
Test::private_int = 12; // wrong, do not touch the private data members
```

leads to a compilation error. Objects accessing the static data member access the same data that is accessed by using the scope resolution operator. The statement

```
myobj.public_int = 145;        // ok
```

refers to the public static data member. However, a private static data member cannot be accessed either by using the scope resolution or the dot operator.

Static Member Functions

Besides static data, C++ allows the definition of static functions. These static functions can access only the static members (data or function) declared in the same class; *non-static* data are unavailable to these functions. Static member functions declared in the public part of a class declaration can be accessed without specifying an object of the class. It is illustrated in the program `dirs.cpp`.

```
// dirs.cpp: static data and member functions of a class
#include <iostream.h>
#include <string.h>
class Directory
{
  public:
     // the static string
     static char path [];   // declaration
     // constructors, destructors etc. not shown here
     // here's the static public function
     static void setpath( char const *newpath );
};
// the static function
void Directory::setpath (char const *newpath)
{
   strcpy( path, newpath );
}
// definition of the static variable
char Directory::path [199] = "/usr/raj";   // definition
void main ()
{
   // static data member access, which is defined as public
   cout << "Path: " << Directory::path << endl;
   // Alternative (1): calling setpath() without
   // an object of the class Directory
   Directory::setpath ("/usr");
   cout << "Path: " << Directory::path << endl;
   // Alternative (2): with an object
   Directory dir;
   dir.setpath ("/etc");
   cout << "Path: " << dir.path;
}
```

Run

```
Path: /usr/raj
Path: /usr
Path: /etc
```

Static member functions can also be defined in the private region of a class. Such private static member functions can access only static data members and can invoke static member functions. The following points should be noted about static members:

- Only one copy of static data member exists for all the instances of a class.
- Static member functions can access only static members of its class.
- Static data members must be defined and initialized like global variables, otherwise the linker generates errors.
- Static members defined as public can either be accessed through the scope resolution operator as

```
ClassName::MemberName
```

  or it can be accessed through the object of a class as

```
ObjectName.MemberName
```

  That is, static members can be accessed using only the class name, without referring to a particular object.

10.18 Class, Objects and Memory Resource

When a class is declared, memory is not allocated to the data members of the class. Thus, there exists a template, but data members cannot be manipulated unless an instance of this class is created by defining an object. It might give an impression that when an object of a particular class is created, memory is allocated to both its data members and member functions. This is partly true. When an object is created, memory is allocated only to its data members and not to member functions.

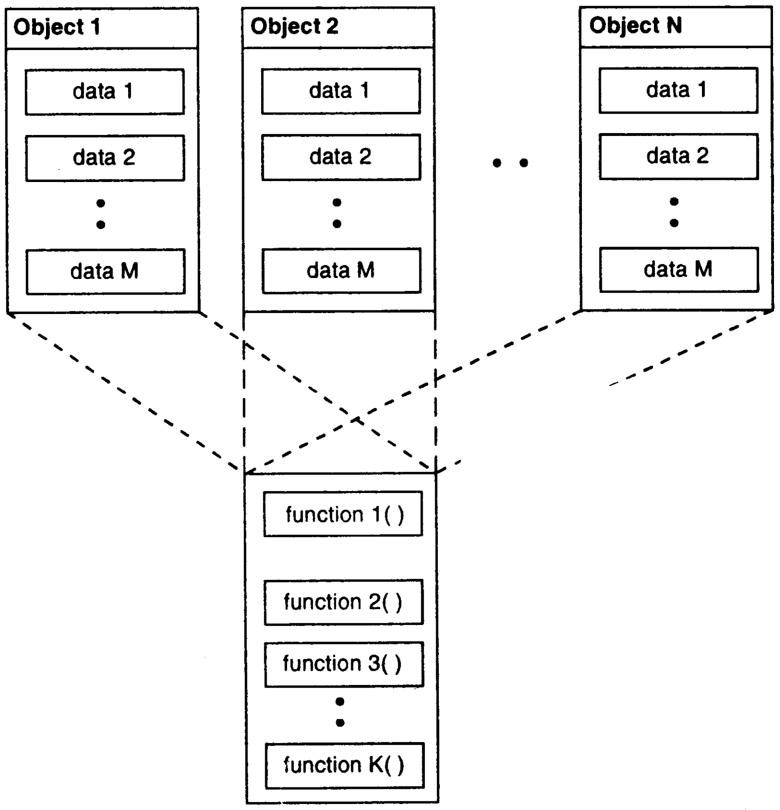
Member functions are created and stored in memory only once when a class specification is declared. All objects of that class have access to the same area in the memory where the member functions are stored. It is also logically true as the member functions are the same for all objects and there is no point in allocating a separate copy for each and every object created using the same class specification. However, separate storage is allocated for every object's data members since they contain different values. It allows different objects to handle their data in a manner that suits them.

The organization of memory resource for the objects is depicted in Figure 10.23. It can be observed that N objects of the same class are created and data members of those objects are stored in distinct memory locations, whereas the member functions of `object1` to `objectN` are stored in the same memory area. Thus, each object has a separate copy of data members and the different objects share the member functions among them. It is simpler to visualize each object as containing both its own data and functions. But the knowledge of what happens behind the scene is useful in estimating the time and space complexity of a program during its execution.

Static Data Members

Whenever a class is instantiated, memory is allocated to the created object. But there exists an exception to this rule. Storage space for data members which are declared as `static` is allocated only once during the class declaration. Subsequently, all objects of this class have access to this data member, i.e., all instances of the class access the same data member. When one of them modifies the static data member, the effect is visible to all the instances of the class.

The organization of memory resource for the object's `static` data members is shown in Figure 10.24. It can be observed that in the `N` objects of the same class, *automatic* data members (of each object) are stored in distinct memory locations, whereas *static* data members (of all objects) are stored in the same memory locations. Thus each object has a separate copy of the automatic data members and they share static data members among them.

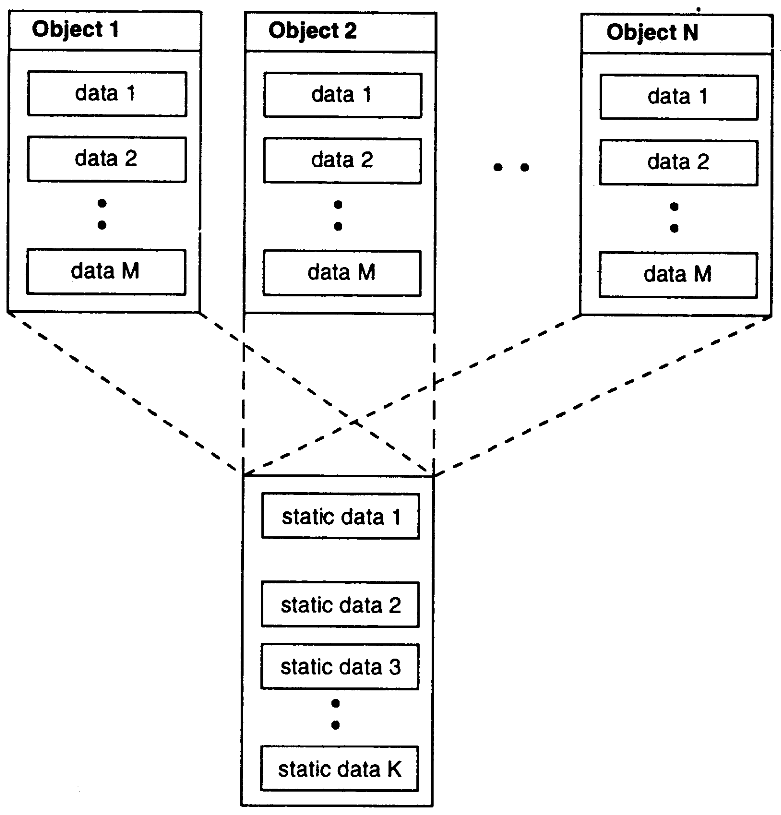


Separate memory for object's data members
Shared memory for class functions

Figure 10.23: Memory for objects' data and function members

A static data member is allocated a fixed area of storage at link time, like a global variable, but the variable's identifier is accessed only using the scope resolution operator with the class name. Thus static data is useful when all the objects of the same class must share a common item of information having same characteristics as non-static members. It is visible only within the class, but its extent (lifespan) is the entire program execution period.

Data members are generally allocated with the same storage class. If an object is declared `auto`, all its data is `auto`; static objects have static data members. Static data members are an *exception* to this rule; when an object is created, memory is not allocated to its static members (if there are any), because this would cause multiple copies of the static data member appear in every object.



Separate memory for class's automatic data members
Shared memory for class's static data members

Figure 10.24: Memory for objects' static and automatic data members

10.19 Class Design Steps

As pointed out by the designer of C++, Dr. Bjarne Stroustrup, "Considering designing a single class is typically not a good idea. Concepts do not exist in isolation; rather, a concept is defined in the context of other concepts. Similarly, a class does not exist in isolation, but is declared together with logically related classes. Such a set is often called a class library or a component. Sometimes all classes in a component constitute a single class hierarchy, sometimes they do not."

The set of classes in a component is united by some logical criteria, often by a component style and by a reliance on common services. A component is thus the unit of design, documentation, ownership, and often reuse. However, to use any part of a component, one needs to understand the logical criteria that define the component, the conventions and style embodied in the design of the components and its documentation, and the common services (if any).

The design of a component is a challenging task. It can be easily handled by breaking it into steps so that focus can be placed on the various sub-tasks in a logical and complete way. (Unlike structured programming, OOPs concentrates on data decomposition instead of algorithm decomposition.) However, there is no one right method for component design, Here is a series of steps that have worked well in the design of components with most designers:

[1]. Find the concepts/classes and their most fundamental relationships.

[2]. Refine the classes by specifying the sets of operations on them.

a. Classify these operations. In particular, consider the needs for construction, copying, and destruction. C++ features for defining such operations are discussed in the chapter on *Object Initialization and Cleanup*.

b. Provide standard interface. It must provide the same look and feel of standard data types to user defined data types. C++ has constructs for defining such standard interfaces and are discussed in the chapter on *Operator Overloading*.

c. Consider minimalism, completeness, and convenience.

[3]. Refine the classes by specifying their dependencies on other classes:

a. Inheritance. (Discussed in the chapter on *Inheritance*.)

b. Use dependencies.

[4]. Specify the interfaces for the classes.

a. Separate functions into private, public, and protected operations.

b. Specify the exact type of the operations on the classes.

Note that these steps are iterative in nature and hence, several sequences over these steps are required to produce a design code. It is advisable to design these classes as template classes as discussed in the chapter *Generic Programming with Templates*. The error handling model adopted in these classes must use exceptions to report runtime errors; discussed in the chapter *Exception Handling*. Once objects are created dynamically, there must be provision to invoke operations on these objects dynamically. These features are discussed in the chapter *Virtual Functions*. Apart from the class design steps, a true object-oriented development passes through object-oriented analysis, design, testing, etc., phases; discussed in the chapter *OO Analysis, Design and Development*.

Review Questions

10.1 What is a class ? Describe the syntax for declaring a class with examples.

10.2 What are the differences between structures and classes in C++ ?

10.3 What are objects ? Describe the syntax for defining objects with examples. Explain how C++ supports encapsulation and data abstraction.

10.4 Write a program illustrating class declaration, definition, and accessing class members.

10.5 Explain the client-server model of object communication.

10.6 The University requires an interactive student database package that permits one to keep track of the dynamic student population in the campus. This database maintains at the minimum, a student's name, roll-no, marks of three *hardcore* subjects and three *softcore* subjects. The information about any student can come at any time.

(a) What kind of data structure is suited for the above implementation and why ?

(b) Give the class specification.

(c) Given a student's roll-no, how do we determine the marks scored by the student ?

10.7 What are the guidelines that need to be followed for deciding whether to make the member functions inline or not ?

10.8 What is the difference between member functions defined inside and outside the body of a class? How are inline member functions defined outside the body of a class ?

10.9 What is data hiding ? What are the different mechanisms for protecting data from the external users of a class's objects ?

10.10 What are empty classes ? Can instances of empty class be created ? Give reasons.

10.11 Write a program for adding two vectors (which are objects of the class `Vector`). Use dynamic data members instead of arrays for storing vector elements.

10.12 Explain the different methods of passing object parameters.

10.13 Write an interactive program for manipulating objects of the `Distance` class. Support member functions for adding and subtracting distance members of two objects.

10.14 What are friend functions and friend classes ? Write a normal function which adds objects of the complex number class. Declare this normal function as friend of the `Complex` class.

10.15 Write a program for processing objects of the `Student` class. Declare member functions such as `show()` as read-only member functions.

10.16 Bring out the differences between `auto` and `static` storage class data members. Can `static` member functions of a class access all types of members of a class. Give reasons. What are the access rules for accessing `static` members ?

10.17 Discuss memory requirements for classes, objects, data members, member functions, static and non-static data members.

10.18 Why object-oriented programming approach is the preferred form of programming over other approaches.

10.19 Write a program for manipulating coordinates in Rectangle coordinate system. Represent points as objects. The class `Point` must include members such as `x` and `y` (as data members), and `add()`, `sub()`, `angle()`, etc. (as member functions).

10.20 Write a program for manipulating coordinates in Polar coordinate system. Represent points as objects. The class `Polar` must include data members such as `radius` and `theta`, and member functions such as `add()`, `sub()`, `angle()`, etc.

10.22 Explain steps involved in designing class components as suggested by the C++ designer.

11

Object Initialization and Cleanup

11.1 Class Revisited

A class encapsulates both data and functions manipulating them into a single unit. It can be further used as an abstract data type for defining a class instance called object. As with standard data types, there must exist a provision to initialize objects of a class during their definition itself. Consider an example of the class `bag` having two data members: `contents` to hold fruits and `ItemCount` to hold the number of items currently stored in the `bag`. It has interface functions such as `SetEmpty()`, `put()`, and `show()` whose usage is illustrated in the program `bag.cpp`.

```
// bag.cpp: Bag into which fruits can be placed
#include <iostream.h>
const int MAX_ITEMS = 25;  // Maximum number of items that a bag can hold
class Bag
{
   private:
      int contents[MAX_ITEMS];    // bag memory area
      int ItemCount;              // Number of items present in a bag
   public:
      // sets ItemCount to empty
      void SetEmpty()
      {
         ItemCount = 0;           //When you purchase a bag, it will be empty
      }
      void put( int item )        // puts item into bag
      {
         contents[ ItemCount++ ] = item; // counter update
      }
      void show();
};
// display contents of a bag
void Bag::show()
{
   for( int i = 0; i < ItemCount; i++ )
      cout << contents[i] << " ";
   cout << endl;
}
void main()
{
   int item;
   Bag bag;
   bag.SetEmpty();   // set bag to empty
   while( 1 )
   {
      cout<<"Enter Item Number to be put into the bag <0-no item>: ";
      cin >> item;
      if( item == 0 )    // items ends, break
         break;
      bag.put( item );
      cout << "Items in Bag: ";
      bag.show();
   }
}
```

Run

```
Enter Item Number to be put into the bag <0-no item>: 1
Items in Bag: 1
Enter Item Number to be put into the bag <0-no item>: 3
Items in Bag: 1 3
Enter Item Number to be put into the bag <0-no item>: 2
Items in Bag: 1 3 2
Enter Item Number to be put into the bag <0-no item>: 4
Items in Bag: 1 3 2 4
Enter Item Number to be put into the bag <0-no item>: 0
```

In `main()`, the statement

```
Bag bag;
```

creates the object `bag` without initializing the `ItemCount` to 0 automatically. However, it is performed by a call to the function `SetEmpty()` as follows:

```
bag.SetEmpty();   // set bag to empty
```

According to the philosophy of OOPs, when a new object such as `bag` is created, it will naturally be empty. To provide such a behavior in the above program, it is necessary to invoke the member function `SetEmpty` explicitly. In reality, when a bag is purchased, it might contain some items placed inside the bag as gift items. Such a situation in C++ can be simulated by

```
Bag bag1 = 2;
```

It creates the object bag and initializes it with 2, indicating that the bag is sold with two gift items. It resembles the procedure of initialization of a built-in data type during creation, i.e., there must be a provision in C++ to initialize objects during creation itself.

It is therefore clear that OOPs must provide a support for initializing objects when they are created, and destroy them when they are no longer needed. Hence, a class in C++ may contain two special member functions dealing with the internal workings of a class. These functions are the *constructors* and the *destructors*. A constructor enables an object to initialize itself during creation and the destructor destroys the object when it is no longer required, by releasing all the resources allocated to it. These operations are called *object initialization* and *cleanup* respectively.

11.2 Constructors

A constructor is a special member function whose main operation is to allocate the required resources such as memory and initialize the objects of its class. A constructor is distinct from other member functions of the class, and it has the same name as its class. It is executed automatically when a class is instantiated (object is created). It is generally used to initialize object member parameters and allocate the necessary resources to the object members. The constructor has no return value specification (not even void). For instance, for the class `Bag`, the constructor is `Bag::Bag()`.

The C++ run-time system makes sure that *the constructor of a class is the first member function to be executed automatically when an object of the class is created.* In other words, the constructor is executed everytime an object of that class is defined. Normally constructors are used for initializing the class data members. It is of course possible to define a class which has no constructor at all; in such a case, the run-time system calls a dummy constructor (i.e., which performs no action) when its object is created. The syntax for defining a constructor with its prototype within the class body and the actual definition outside it, is shown in Figure 11.1. Similar to other members, the constructor can be defined either within, or outside the body of a class. It can access any data member like all other member functions but cannot be invoked explicitly and must have public status to serve its purpose. The constructor which does not take arguments explicitly is called *default constructor.*

```
class ClassName
{
     .....  // private members
     public :  ◄──   must be public
          // public members
         ▲ClassName ( ) ;  ◄──   Constructor prototype
};       │
┌────────┴──── no return type nor void
▼
ClassName :: ClassName( )  ◄──   Constructor definition
{
     // constructor body definition
}
```

Figure 11.1: Syntax of constructor

The initialization may entail calling functions, allocating dynamic storage, setting variables to specific values, and so on. Since the constructor is executed every time an object is created, it can be used to assign initial values to the data members of the object. It will reduce the burden on the programmer to specifically initialize the data within each object that is created and hence, prevent errors. These constructors do not have any return type, since they are invoked during the creation of objects transparently. But they can have as many arguments as necessary.

The program `newbag.cpp` has a counter, which can be used to count events or objects placed in a bag. Since the counter has to start from zero value and count upwards, a mechanism is required by which the counter can be set to zero as soon as it is created. An appropriate solution to this situation, is to use a constructor.

```
// newbag.cpp: Bag into which fruits can be placed with constructor
#include <iostream.h>
const int MAX_ITEMS = 25;  // Maximum number of items that a bag can hold
class Bag
{
   private:
      int contents[MAX_ITEMS];    // bag memory area
      int ItemCount;              // Number of items present in a bag
   public:
      // sets ItemCount to empty
      Bag()                    // constructor
      {
         ItemCount = 0;         // When you purchase a bag, it will be empty
      }
      void put( int item )      // puts item into bag
      {
         contents[ ItemCount++ ] = item; // item into bag, counter update
      }
      void show();
};
// display contents of the bag
void Bag::show()
{
   for( int i = 0; i < ItemCount; i++ )
      cout << contents[i] << " ";
   cout << endl;
}
void main()
{
   int item;
   Bag bag;
   while( 1 )
   {
      cout << "Enter Item Number to be put into the bag <0-no item>: ";
      cin >> item;
      if( item == 0 )   // items ends, break
         break;

      bag.put( item );
      cout << "Items in Bag: ";
      bag.show();
   }
}
```

Run

```
Enter Item Number to be put into the bag <0-no item>: 1
Items in Bag: 1
Enter Item Number to be put into the bag <0-no item>: 3
Items in Bag: 1 3
Enter Item Number to be put into the bag <0-no item>: 2
Items in Bag: 1 3 2
Enter Item Number to be put into the bag <0-no item>: 4
Items in Bag: 1 3 2 4
Enter Item Number to be put into the bag <0-no item>: 0
```

In `main()`, the class instantiation statement

```
Bag bag;
```

creates the object `bag` and initializes `ItemCount` to zero by invoking the no-argument constructor

```
        Bag::Bag()
```

automatically. In the earlier program `bag.cpp`, these actions are performed by the following statements

```
        Bag bag;
        bag.SetEmpty();   // set bag to empty
```

First, the object bag is created and then, `SetEmpty()` is explicitly invoked to initialize the data member `ItemCount` to zero.

When an object is a local non-static variable in a function, the constructor `Bag()` is called when the function is invoked. When an object is a global or a static variable, the constructor `Bag()` is invoked before the execution of `main()` as illustrated in the program `test1.cpp`.

```
// test1.cpp: a class Test with a constructor function
#include <iostream.h>
class Test
{
   public:                  // 'public' function:
      Test();               // the constructor
};
Test::Test()                // here is the definition
{
   cout << "constructor of class Test called" << endl;
}
// and here is the test program:
Test  G;                    // global object
void func()
{
   Test L;                  // local object in function func()
   cout << "here's function func()" << endl;
}
void main()
{
   Test X;                  // local object in function main()
   cout << "main() function" << endl;
   func ();
}
```

Run

```
constructor of class Test called       (global object G)
constructor of class Test called       (object X in main())
main() function
constructor of class Test called       (object L in func())
here's function func()
```

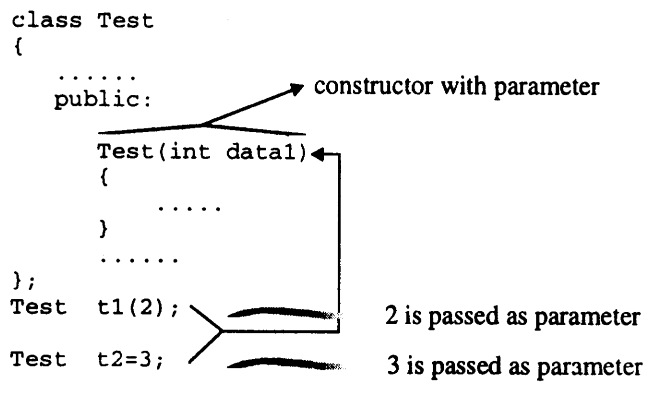
The output produced by the program is as desired (see ***Run*** - the text in parentheses indicates comment). The program shows how a class `Test` is defined, which consists of only one function: the constructor. The constructor performs only one action; a message is printed. The program contains three objects of the class `Test`: first a global object, second a local object in `main()`, and third another local object in `func()`.

A constructor has the following characteristics:

- It has the same name as that of the class to which it belongs.
- It is executed automatically whenever the class is instantiated.
- It does not have any return type.
- It is normally used to initialize the data members of a class.
- It is also used to allocate resources such as memory, to the dynamic data members of a class.

11.3 Parameterized Constructors

Constructors can be invoked with arguments, just as in the case of functions. The argument list can be specified within braces similar to the argument-list in the function. Constructors with arguments are called *parameterized constructors*. The distinguishing characteristic is that the name of the constructor functions have to be the same as that of its class name. In the earlier program `newbag.cpp`, another constructor with arguments could have been provided with one integer value to initialize the data members `ItemCount` and `contents[]` . The syntax of parameterized constructors and their access is shown in Figure 11.2.



```
class Test
{
   ......
   public:
      Test(int data1)
      {
         .....
      }
      ......
};
Test  t1(2);
Test  t2=3;
```

Figure 11.2: Parameterized constructor

Since C++ allows function overloading, a constructor with arguments can co-exist with another constructor without arguments. The class `Bag` would thus have two constructors. The usage of a constructor with arguments is illustrated in the modified program `giftbag.cpp` of `newbag.cpp`. The object is initialized during its creation.

```
// giftbag.cpp: Bag which has some items when gifted
#include <iostream.h>
const int MAX_ITEMS = 25;  // Maximum number of items that a bag can hold
class Bag
{
   private:
      int contents[MAX_ITEMS];    // bag memory area
      int ItemCount;              // Number of items present in a bag
```

```
  public:
     // sets ItemCount to empty, it is gifted as empty bag
     Bag()                // constructor without arguments
     {
        ItemCount = 0;
     }
     Bag( int item )      // constructor with arguments
     {
        contents[ 0 ] = item;  // when bag is gifted, it'll have some items
        ItemCount = 1;
     }
     void put( int item ) // puts item into bag
     {
        contents[ ItemCount++ ] = item; // item into bag, counter update
     }
     void show();
};
// display contents of a bag
void Bag::show()
{
   if( ItemCount )
      for( int i = 0; i < ItemCount; i++ )
         cout << contents[i] << " ";
   else
      cout << "Nil";
   cout << endl;

void main()
{
   int item;
   Bag bag1;      // uses Bag::Bag() constructor
   Bag bag2 = 4;  // uses Bag::Bag( int item ) constructor
   cout << "Gifted bag1 initially has: ";
   bag1.show();
   cout << "Gifted bag2 initially has: ";
   bag2.show();
   while( 1 )
   {
      cout << "Enter Item Number to be put into the bag2 <0-no item>: ";
      cin >> item;
      if( item == 0 )    // items ends, break
         break;
      bag2.put( item );
      cout << "Items in bag2: ";
      bag2.show();
   }
}
```

Run

```
Gifted bag1 initially has: Nil
Gifted bag2 initially has: 4
Enter Item Number to be put into the bag2 <0-no item>: 1
Items in bag2: 4 1
Enter Item Number to be put into the bag2 <0-no item>: 2
Items in bag2: 4 1 2
Enter Item Number to be put into the bag2 <0-no item>: 3
Items in bag2: 4 1 2 3
Enter Item Number to be put into the bag2 <0-no item>: 0
```

The `Bag` class has two constructors. The first constructor does not have any arguments. The next constructor has a single argument. The statement

```
Bag bag1;
```

creates the object `bag1` and initializes its data member `ItemCount` by invoking the no-argument constructor `Bag::Bag()`. The next statement

```
Bag bag2 = 4;
```

creates the object `bag2` and sets its data members `ItemCount` to 1 and `contents` to 4 by invoking the one-argument constructor `Bag::Bag( int item )`. The concept of having multiple constructors and their invocation based on suitable arguments during the creation of objects `bag1` and `bag2` with user interface is shown in Figure 11.3.

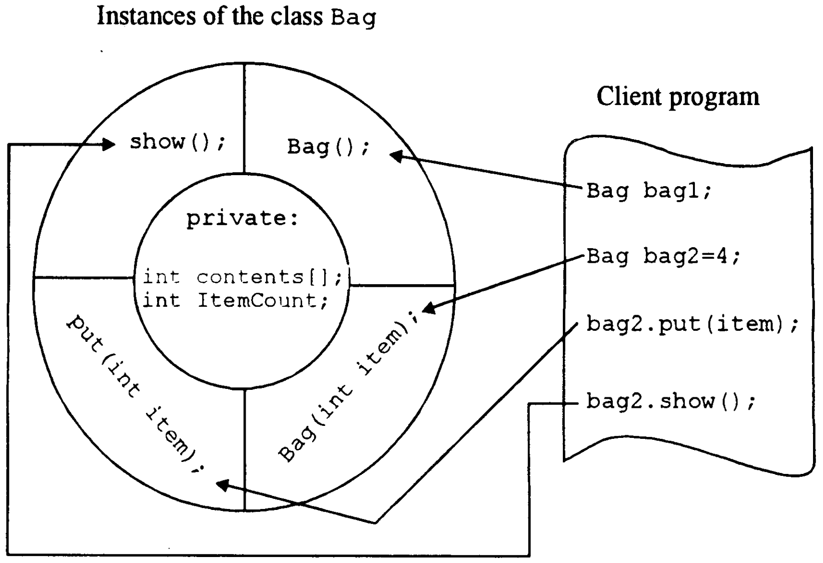


Figure 11.3: Bag class and parameterized constructor

When a constructor is declared not to accept any arguments, it is called a *default constructor*. It is invoked when the object is instantiated with no arguments. The constructor `Bag()` is a default constructor. Since a default constructor takes no arguments, it follows that each class can have only one default constructor. The operation of the default constructor function is usually to initialize data, used subsequently by other member functions. It can also be used to allocate the necessary resources such as memory, dynamically.

11.4 Destructor

When an object is no longer needed it can be destroyed. A class can have another special member function called the *destructor*, which is invoked when an object is destroyed. This function complements the operation performed by any of the constructors, in the sense that, it is invoked when an object ceases to exist. For objects which are local non-static variables, the destructor is called when the function in which the object is defined is about to terminate. For static or global variables, the destructor is called before the program terminates. Even when a program is interrupted using an `exit()` call, the destructors are called for all objects which exist at that time.

The syntax of the destructor is shown in Figure 11.4. Destructor is a member function having the character ~(tilde) followed by the name of its class and brackets (i.e., `~classname()`). It is invoked automatically to reclaim all the resources allocated to the object when the object goes out of scope and is no longer needed.

```
class ClassName
{
      .....  // private members
      public :  ←  must be public
             // public members
             ~ ClassName();  ←  Destructor prototype
};           ↑
             └──→ Tilde character, destructor returns nothing
ClassName :: ~ ClassName()  ←  Destructor definition
{
      // destructor body definition
}
```

Figure 11.4: Syntax of destructor

Similar to constructors, a destructor must be declared in the `public` section of a class so that it is accessible to all its users. Destructors have no return type. It is incorrect to even declare a `void` return type. *A class cannot have more than one destructor.* The program `test.cpp` illustrates the use of destructors.

```
// test.cpp: a class Test with a constructor and destructor
#include <iostream.h>
class Test
{
   public:                  // 'public' function:
      Test();               // the constructor
      ~Test();              // the destructor
};
Test::Test()                  // here is the definition of constructor
{
   cout << "constructor of class Test called" << endl;
}
Test::~Test()                 // here is the definition of destructor
{
   cout << "destructor of class Test called" << endl;
}
```

```
void main()
{
   Test x;     // constructor is called while creating
   cout << "terminating main()" << endl;
}  // object x goes out of scope, destructor is called
```

Run

```
constructor of class Test called
terminating main()
destructor of class Test called
```

An interesting aspect of constructors and destructors is illustrated in the program count.cpp. It keeps track of the number of objects created and how many of them are still alive.

```
// count.cpp: counts how many objects are created and how may are alive
#include <iostream.h>
int nobjects = 0;    // number of objects of the class MyClass
int nobj_alive = 0;  // number of objects present of the class MyClass
class MyClass
{
   public:
      MyClass()       // increments objects count
      {
         ++nobjects;    // add to total
         ++nobj_alive;  // add to the active
      }
      ~MyClass()  // decrements active objects count
      {
         --nobj_alive;  // deduct one from active objects list
      }
      void show()
      {
         cout << "Total number of objects created: " << nobjects << endl;
         cout<<"Number of objects currently alive: "<<nobj_alive << endl;
      }
};
void main()
{
   MyClass obj1;
   obj1.show();
   {  // new block
      MyClass obj1, obj2;
      obj2.show();    // can be obj1.show()
   }  // obj1 and obj2 goes out of scope, hence deleted
   obj1.show();
   MyClass obj2, obj3;
   obj2.show();    // can be obj1.show() or obj3.show()
}
```

Run

```
Total number of objects created: 1
Number of objects currently alive: 1
```

```
Total number of objects created: 3
Number of objects currently alive: 3
Total number of objects created: 3
Number of objects currently alive: 1
Total number of objects created: 5
Number of objects currently alive: 3
```

The constructor in the above program increments the global variables `nobjects` and `nobj_alive`, by one. Whenever an object is created, the constructor is invoked automatically and counters are updated to maintain the object's statistics. The destructor decrements only the count variable `nobj_alive` by one. Whenever objects go out of scope, the destructor is invoked automatically and the counters will get updated (decremented). The status can be retrieved by using the member function `show()` of the class `MyClass`. It prints the same message irrespective of the object invoking it; (it uses global data, which remains the same irrespective of the object's message).

The following rules need to be considered while defining a destructor for a given class:

- The destructor function has the same name as the class but prefixed by a tilde (~). The tilde distinguishes it from a constructor of the same class.
- Unlike the constructor, the destructor does not take any arguments. This is because there is only one way to destroy an object.
- The destructor has neither arguments, nor a return value.
- The destructor has no return type like the constructor, since it is invoked automatically whenever an object goes out of scope.
- There can be only one destructor in each class. This is essentially a violation of the rule that a function can take arguments, thereby making function overloading impossible.

11.5 Constructor Overloading

An interesting feature of the constructors is that a class can have multiple constructors. This is called *constructor overloading*. All the constructors have the same name as the corresponding class, and they differ only in terms of their signature (in terms of the number of arguments, or data types of their arguments, or both) as illustrated in the program `account.cpp`.

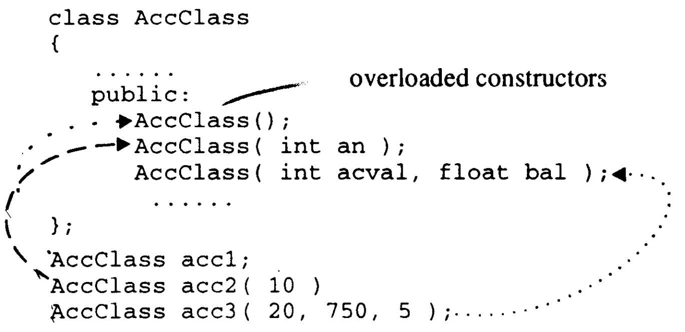
```
// account.cpp: passing objects as parameters to functions
#include<iostream.h>
class AccClass
{
   private:                // class data members
      int accno;
      float balance;
   public:                 // class function members
      AccClass()           // Constructor no.1
      {
         cout << "Enter the account number for acc1 object: ";
         cin >> accno;
         cout << "Enter the balance: ";
         cin >> balance;
      }
      AccClass(int an)      // Constructor no.2
      {
         accno = an;
         balance = 0.0 ;
      }
      AccClass(int acval, float bal)   // Constructor no.3
      {
         accno = acval;
         balance = bal;
      }
      void display()
      {
         cout << "Account number is: " << accno << endl;
         cout << "Balance is: " << balance << endl;
      }
      void MoneyTransfer( AccClass & acc, float amount );
};
// acc1.MoneyTransfer( acc2, 100 ),transfers 100 rupees from acc1 to acc2
void AccClass::MoneyTransfer( AccClass & acc, float amount )
{
   balance = balance - amount;   // deduct money from source
   acc.balance = acc.balance + amount; // add money to destination
}
void main()
{
   int trans_money;
   AccClass acc1;                   // uses constructor 1
   AccClass acc2( 10 );             // uses constructor 2
   AccClass acc3( 20, 750.5 );      // uses constructor 3
   cout << "Account Information..." << endl;
   acc1.display();
   acc2.display();
   acc3.display();
   cout << "How much money is to be transferred from acc3 to acc1: ";
   cin >> trans_money;
   // transfer trans_money from acc3 to acc1
   acc3.MoneyTransfer( acc1, trans_money );
   cout << "Updated Information about accounts..." << endl;
   acc1.display();
   acc2.display();
   acc3.display();
}
```

Run

```
Enter the account number for acc1 object: 1
Enter the balance: 100
Account Information...
Account number is: 1
Balance is: 100
Account number is: 10
```

```
Balance is: 0
Account number is: 20
Balance is: 750.5
How much money is to be transferred from acc3 to acc1: 200
Updated Information about accounts...
Account number is: 1
Balance is: 300
Account number is: 10
Balance is: 0
Account number is: 20
Balance is: 550.5
```

In case of a class having multiple constructors, a constructor is invoked during the creation of an object depending on the number and type of arguments passed. The default constructor can also be defined along with other constructors, if necessary. The invocation of different constructors during the creation of an object of the class `AccClass` is shown in Figure 11.5.



```
class AccClass
{
    ......
    public:
       AccClass();
       AccClass( int an );
       AccClass( int acval, float bal );
       ......
};
AccClass acc1;
AccClass acc2( 10 )
AccClass acc3( 20, 750, 5 );
```

Figure 11.5: Constructor overloading

In this program, whenever a new account is created, one of the three steps is chosen:

- If no arguments are passed, then the program prompts the user for an account number and balance by invoking the no-argument constructor, `AccClass()`.
- If only an `int` argument, is provided, then the account number is initialized with the value passed as an input argument while, the balance is set to 0.0 by invoking the one-argument constructor `AccClass(int)`.
- If both an `int` as well as a `float` argument is provided, then the account number is set to the `int` value while the balance is set to the `float` value by invoking the two-argument constructor, `AccClass(int, float)`.

Differences between Constructors and Destructors

The following are the differences between constructors and destructors:

- Arguments cannot be passed to destructors.
- Only one destructor can be declared for a given class as a consequence of the fact that destructors cannot have arguments and hence, destructors cannot be overloaded.
- Destructors can be virtual, while constructors cannot be virtual. More details can be found in the chapter *Virtual Functions*.

11.6 Order of Construction and Destruction

The possibility of defining constructors with arguments, offers an opportunity to monitor (examine) the exact moment at which an object is created or destroyed during the execution of a program. This has been illustrated in the program, `test2.cpp` using the `Test` class.

```
// test2.cpp: the class Test with a constructor and destructor function
#include <iostream.h>
#include <string.h>
class Test
{
   private:
      char *name;
   public:                  // 'public' function:
      Test();               // the constructor
      Test( char *msg );    // one-argument constructor
      ~Test();
};
Test::Test()                // here is the
{                           // definition
   name = new char[ strlen("unnamed")+1 ];
   strcpy( name, "unnamed" );
   cout << "Test object 'unnamed' created" << endl;
}
Test::Test( char *NameIn )
{
   name = new char[  strlen(NameIn)+1 ];
   strcpy( name, NameIn );
   cout << "Test object " << NameIn << " created" << endl;
}
Test::~Test ()
{
   cout <<"Test object " << name << " destroyed" << endl;
   delete name;            // release memory
}
// and here is the test program:
Test  g("global");                   // global object
void func()
{
   Test l("func");                   // local object in function func()

   cout << "here's function func()" << endl;
}
void main()
{
   Test x("main");                   // local object in function main()
   func ();
   cout << "main() function - termination" << endl;
}
```

Run

```
Test object global created
Test object main created
Test object func created
here's function func()
Test object func destroyed
main() function - termination
Test object main destroyed
Test object global destroyed
```

By defining objects of the class `Test` with specific names, the construction and destruction of these objects can be monitored. In the above program, global objects are created first, hence the statement

```
Test g("global");
```

creates the object `g` and initializes its member `name` to `"global"`. In `func()`, the statement

```
Test l("func");
```

creates the local object `l` and initializes its member `name` to `"func"`. In `main()`, the statement

```
Test x("main"); // local object in function main()
```

creates the local object `x` and initializes its member `name` to `"main"`.

The object which goes out of scope is immediately destroyed. In the above program, the function `func()` terminates first and hence, the local object `l` is destroyed first, which can also be observed from the program output. Secondly, the object `x` is destroyed during the termination of the function `main()`. Finally, the global object `g` is destroyed. When more than one object is created globally, or locally, they are destroyed in the reverse chronological order (*object created most recently is the first one to be destroyed*).

11.7 Constructors with Default Arguments

Like any other function in C++, constructors can also be defined with default arguments. If any arguments are passed during the creation of an object, the compiler selects the suitable constructor with default arguments. The program `complex1.cpp` illustrates the usage of default arguments during the creation of objects of the `complex` type class.

```
// complex1.cpp: default arguments to complex class
#include <iostream.h>
#include <math.h>
class complex
{
  private:
    float real;     // real part of complex number
    float imag;     // imaginary part of complex number
  public:
    complex()       // constructor 0
    {
      real = imag = 0.0;
    }
```

```
      complex( float real_in, float imag_in = 0.0 )   // constructor1
      {
         real = real_in;
         imag = imag_in;
      }
      void show( char *msg )     // display complex number in x+iy form
      {
         cout << msg << real;
         if( imag < 0 )
            cout << "-i";
         else
            cout << "+i";
         cout << fabs(imag) << endl;
      }
      complex add( complex c2 ); // Addition of complex numbers
};
// temp = default object + c2;
complex complex::add( complex c2 )      // add default and c2 complex objects
{
   complex temp;                        // object temp of complex class
   temp.real = real + c2.real;          // add real parts
   temp.imag = imag + c2.imag;          // add imaginary parts
   return( temp );                      // return complex object
}
void main()
{
   complex c1( 1.5, 2.0);  // uses constructor1
   complex c2( 2.2 );      // uses constructor1 with default imag value
   complex c3;             // uses constructor0
   c1.show("c1 = ");
   c2.show("c2 = ");
   c3 = c1.add( c2 );      // add c1 and c2 assign to c3
   c3.show( "c3 = c1.add( c2 ): ");
}
```

Run

```
c1 = 1.5+i2
c2 = 2.2+i0
c3 = c1.add( c2 ): 3.7+i2
```

The constructor `complex()`, in the class complex is declared as

```
complex( float real_in, float imag_in = 0.0 )  // constructor1
```

The default value of the argument `imag_in` is zero. Then, the statement in `main()`,

```
complex c2( 2.2 );
```

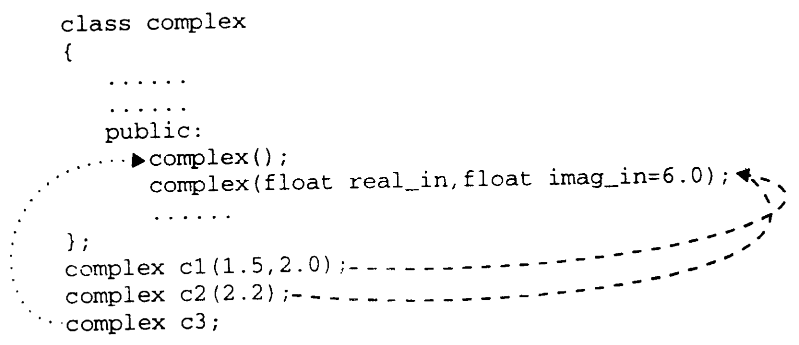
passes only one parameter explicitly to the constructor. The compiler treats this statement as,

```
complex c2( 2.2, 0.0 );
```

by assuming the second argument to have default argument value (`image_in = 0.0`) specified at the declaration of the constructor. However, the statement,

```
complex c1( 1.5, 2.0);
```

assigns 1.5 to `real_in` and 2.0 to `imag_in`. If the actual parameter is explicitly specified, it overrides the default value. As stated earlier, the missing arguments must be the trailing ones. The invocation of a constructor with default arguments while creating objects of the class `complex` is shown in Figure 11.6.



```
class complex
{
   ......
   ......
   public:
      complex();
      complex(float real_in,float imag_in=6.0);
      ......
};
complex c1(1.5,2.0);
complex c2(2.2);
complex c3;
```

Figure 11.6: Default arguments to constructor

Suppose the specification of the constructor `complex(float,float)` is changed to,

```
complex( float real_in = 0.0, float imag_in = 0.0 )
```

in the above program, it causes ambiguity while using a statement such as,

```
complex c1;
```

The confusion is whether to call the no-argument constructor,

```
complex::complex()
```

or the two argument default constructor

```
complex::complex( float = 0.0, float = 0.0)
```

Hence, such a specification should be avoided. If no constructors are defined, the compiler tries to generate a default constructor. This default constructor simply allocates storage to build an object of its class. A constructor that has all default arguments is similar to a default (no-argument) constructor, because it can be called without any explicit arguments. This may also lead to errors as shown in the following program segment:

```
class X
{
   int value;
   public:
      X()
      {
         value=0;
      }
      X(int i=0)
      {
         value=i;
      }
};
```

```
void main()
{
   X c;     // Error: This leads to errors as compiler will not be
            // able to decide which constructor should be called
   X c1(4); // OK
}
```

Trying to create an object of the class X without any arguments, will cause an error as two different constructors satisfy the requirement. Hence, the statement,

```
X c;
```

causes the ambiguity whether to call `X::X()` or `X::X(int i= 0)`. In this, if the default constructor is removed, the program works properly.

11.8 Nameless Objects

C++ not only supports the creation of named objects, but also the creation of unnamed objects. In the object creation statement, the name of an object need not be mentioned. The general format for instantiating nameless objects is shown in Figure 11.7.

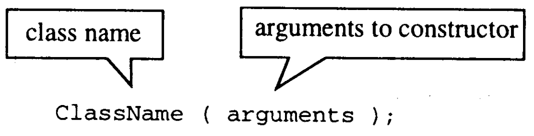


Figure 11.7: Syntax of creating nameless objects

In the above syntax, the name of the object is not mentioned. However, the method of passing arguments to a constructor, and the procedure for creating the nameless object is similar to the procedure for creating named objects. Passing arguments to an object is optional and if no-arguments are mentioned, a default constructor of the class is invoked. If arguments are mentioned in the object creation statement, C++ invokes a constructor of the class that matches with the argument types. After execution of the constructor, nameless objects are immediately destroyed and the destructor of the class is invoked as a part of the object cleanup activity. Hence, the scope of a nameless object is limited only to the statement in which it is created.

The feature of nameless object creation is useful in functions returning an object. The program `noname.cpp` demonstrates the creation of nameless objects.

```
// noname.cpp: Nameless object creation
#include <iostream.h>
class nameless
{
   int a;
   public:
      nameless()
      {
         cout << "Constructor" << endl;
      }
      ~nameless()
      {
         cout << "Destructor" << endl;
      }
};
void main()
{
   nameless(); // nameless object is created as well as destroyed here
   nameless n1;
   nameless n2;
   cout << "Program terminates" << endl;
}
```

Run

```
Constructor    <— nameless()
Destructor     <— nameless()
Constructor    <— nameless n1()
Constructor    <— nameless n2()
Program terminates
Destructor     <— during program termination
Destructor     <— during program termination
```

From the output it is observed that the first two output statements are generated by the statement

```
    nameless();// nameless object is created as well as destroyed here
```

It can be observed that, a nameless object is created and destroyed at the same point. But this is not the case with named objects. The statements,

```
   nameless n1;
   nameless n2;
```

create the named objects n1 and n2 and they are destroyed during the termination of the program.

11.9 Dynamic Initialization through Constructors

Object's data members can be dynamically initialized during runtime, even after their creation. The advantage of this feature is that it supports different initialization formats using overloaded constructors. It provides flexibility of using different forms of data at runtime depending upon the user's need.

Consider an example of naming persons. Some persons have only the first name (person name), some have the first and second name (person name and surname), and others have all the three (person name, surname, and third name). The program name.cpp illustrates the use of objects for holding names and constructing them at runtime using dynamic initialization.

```
// name.cpp: object with different name pattern
#include <iostream.h>
#include <string.h>
class name
{
   private:
      char first[15];   // first name
      char middle[15];  // middle name
      char last[15];    // last name
   public:
      name()   // constructor0
      {
         // initialize all string pointers to NULL
         first[0] = middle[0] = last[0] = '\0';
      }
      name( char *FirstName );   // constructor1
      name( char *FirstName, char *MiddleName );  // constructor2
       //constructor3
      name( char *FirstName, char *MiddleName, char *LastName );
      void show( char *msg );
};
inline name::name( char *FirstName )
{
   strcpy( first, FirstName );
   middle[0] = last[0] = '\0';   // others to NULL
}
inline name::name( char *FirstName, char *MiddleName )
{
   strcpy( first, FirstName );
   strcpy( middle, MiddleName );
   last[0] = '\0';               // others to NULL
}
name::name( char *FirstName, char *MiddleName, char *LastName )
{
   strcpy( first, FirstName );
   strcpy( middle, MiddleName );
   strcpy( last, LastName );
}
void name::show( char *msg )
{
   cout << msg << endl;
   cout << "First Name: " << first << endl;
   if( middle[0] )
      cout << "Middle Name: " << middle << endl;
   if( last[0] )
      cout << "Last Name: " << last << endl;
}
void main()
{
   name n1, n2, n3;  // constructor0
   n1 = name( "Rajkumar" );               // constructor1
   n2 = name( "Savithri", "S" );          // constructor2
   n3 = name( "Venugopal", "K", "R" );    // constructor3
   n1.show( "First person details..." );
   n2.show( "Second person details..." );
   n3.show( "Third person details..." );
};
```

Run

```
First person details...
First Name: Rajkumar
Second person details...
First Name: Savithri
Middle Name: S
Third person details...
First Name: Venugopal
Middle Name: K
Last Name: R
```

The program has four constructors. The arguments to the last three constructors are passed during runtime. The user input is used to initialize the `name` class's objects in one of the following form:

- No name at all: `default` constructor (constructor0) is invoked
- The first name: constructor1 is invoked
- The first and second name: constructor2 is invoked
- The first, second, and third name: constructor3 is invoked

The compiler selects an appropriate constructor while creating objects by choosing one that matches the input values. For instance, in the situation

```
        n2 = name( "Savithri", "S" );          // constructor2
```

the compiler selects the two argument constructor

```
        name( char *FirstName, char *MiddleName ); // constructor2
```

which matches the call for initializing the object `n2`'s data members.

11.10 Constructors with Dynamic Operations

A major application of constructors and destructors is in the management of memory allocation during runtime. It will enable a program to allocate the right amount of memory during execution for each object when the object's data member size is not the same. Allocation of memory to objects at the time of their construction is known as *dynamic construction.* The allocated memory can be released when the object is no longer needed (goes out of scope) at runtime and is known as *dynamic destruction.* The program `vector1.cpp` shows the use of `new` and `delete` operators during object creation and destruction respectively.

```
// vector1.cpp: vector class with array dynamically allocated
#include <iostream.h>
class vector
{
    int *v;   // pointer to a vector
    int sz;   // size of a vector
  public:
    vector( int size )    // constructor
    {
      sz = size;
      v = new int[ size ];    // dynamically allocate vector
    }
```

```
      ~vector()          // destructor
      {
         delete v;       // release vector memory
      }
      void read();
      void show_sum();
};
void vector::read()
{
   for( int i = 0; i < sz; i++ )
   {
      cout << "Enter vector[ " << i << " ] ? ";
      cin >> v[i];
   }
}
void vector::show_sum()
{
   int sum = 0;
   for( int i = 0; i < sz; i++ )
      sum += v[i];
   cout << "Vector Sum = " << sum;
}
void main()
{
   int count;
   cout << "How many elements are in the vector: ";
   cin >> count;
   // create an object of vector class and compute sum of vector elements
   vector v1( count );
   v1.read();
   v1.show_sum();
}
```

Run

```
How many elements are in the vector: 5
Enter vector[ 0 ] ? 1
Enter vector[ 1 ] ? 2
Enter vector[ 2 ] ? 3
Enter vector[ 3 ] ? 4
Enter vector[ 4 ] ? 5
Vector Sum = 15
```

In `main()`, the statement,

```
vector v1( count );
```

creates the object `v1` of the class vector dynamically of the size specified by the variable `count` (it is also read at runtime). The function `read()` accepts elements of the vector from the console and `show_sum()` computes the sum of all the vector elements and prints the same on the console.

The following points can be emphasized on dynamic initialization of objects.

- A constructor of the class makes sure that the data members are initially 0-pointers (NULL).

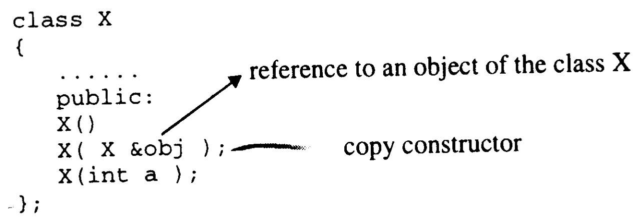
- A constructor with parameters allocates the right amount of memory resources. .
- A destructor releases all the allocated memory.

11.11 Copy Constructor

The parameters of a constructor can be of any of the data types except an object of its own class as a value parameter. Hence declaration of the following class specification leads to an error.

```
class X
{
   private:
    ...
    ...
   public:
      X ( X obj );    // Error: obj is value parameter
       ...
};
```

However, a class's own object can be passed as a reference parameter. Thus the class specification shown in Figure 11.8 is valid.



```
class X
{
   ......
   public:
   X()
   X( X &obj );
   X(int a );
};
```

Figure 11.8: Copy constructor

Such a constructor having a reference to an instance of its own class as an argument is known as *copy constructor*.

The compiler copies all the members of the user-defined source object to the destination object in the assignment statement, when its members are statically allocated. The data members, which are dynamically allocated must be copied to the destination object explicitly. It can be performed by either using the assignment operator, or the copy constructor. Consider the following statements,

```
vector v1( 5 ), v2( 5 );
v1 = v2;          // operator = invoked
vector v3 = v2;   // copy constructor is invoked
```

Assuming that `v1` and `v2` are the predefined objects of the class vector. The statement

```
v1 = v2;
```

will not invoke the copy constructor even though `v1` and `v2` are the objects of class vector. It must cause the compiler to copy the data from `v2`, member-by-member, into `v1`. This is the task of the assignment operator. For more details on assignment operator overloading refer to the chapter on *Operator Overloading*. The next statement,

```
vector v3 = v2;
```

initializes one object with another object during definition. The data members of v2 are copied member-by-member, into v3. It is the default action performed by the copy constructor. The statement,

```
        vector v3( v2 )
```

is treated in the same way as the statement,

```
        vector v3 = v2;
```

by the compiler.

The default actions performed by the compiler are insufficient if data members of an object are dynamically changeable. It can be overcome by overriding these default actions. The program vector2.cpp illustrates the concept of overriding default operations performed by an user-defined copy constructor.

```
// vector2.cpp: copy constructor for vector elements copying
#include <iostream.h>
class vector
{
      int * v;    // pointer to vector
      int size;   // size of vector v
   public:
      vector( int vector_size )
      {
         size = vector_size;
         v = new int[ vector_size ];
      }
      vector( vector &v2 );
      ~vector()
      {
         delete v;
      }
      int & elem( int i )
      {
         if( i >= size )
         {
            cout << endl << "Error: Out of Range";
            return -1;      // illegal access
         }
         return v[i];
      }
      void show();
};
// copy constructor, vector v1 = v2;
vector::vector( vector &v2 )
{
   cout << "\nCopy constructor invoked";
   size = v2.size;            // size of v1 is equal to size of v2
   v = new int[ v2.size ];    // allocate memory of the vector v1
   for( int i = 0; i < v2.size; i++ )
      v[i] = v2.v[i];
}
```

```
void vector::show()
{
   for( int i = 0; i < size; i++ )
      cout << elem( i ) << ", ";
}
void main()
{
   int i;
   vector v1( 5 ), v2( 5 );
   for( i = 0; i < 5; i++ )
      v2.elem( i ) = i + 1;
   v1 = v2;          // copy constructor is not invoked
   vector v3 = v2;   // copy constructor is invoked, vector v3(v2)
   cout << "\nvector v1: ";
   v1.show();
   cout << "\nvector v2: ";
   v2.show();
   cout << "\nvector v2: ";
   v3.show();
}
```

Run

```
Copy constructor invoked
vector v1: 1, 2, 3, 4, 5,
vector v2: 1, 2, 3, 4, 5,
vector v2: 1, 2, 3, 4, 5,
```

A copy constructor copies the data members from one object to another. The function also prints the message (`copy constructor invoked`) to assist the user in keeping track of its execution. The copy constructor takes only one argument, an object of the type vector, passed by reference. The prototype is

```
        vector( vector &v2 );
```

It is essential to use a reference in the argument of a copy constructor. It should not be passed as a value; if an argument is passed by value, its copy constructor would call itself to copy the actual parameter to the formal parameter. This process would go on-and-on until the system runs out of memory. Hence, in a copy constructor, the argument must always be passed by reference, preventing creation of copies. A copy constructor also gets invoked when the arguments are passed by value to functions, and when values are returned from functions. If an object is passed by value, the argument on which the function operates is created using a copy constructor. If an object is passed by address, or reference, the copy constructor would not be invoked, since, in such a case, copies of the objects need not be created. When an object is returned from a function, the copy constructor is invoked to create a copy of the value returned by the function.

11.12 Constructors for Two-dimensional Arrays

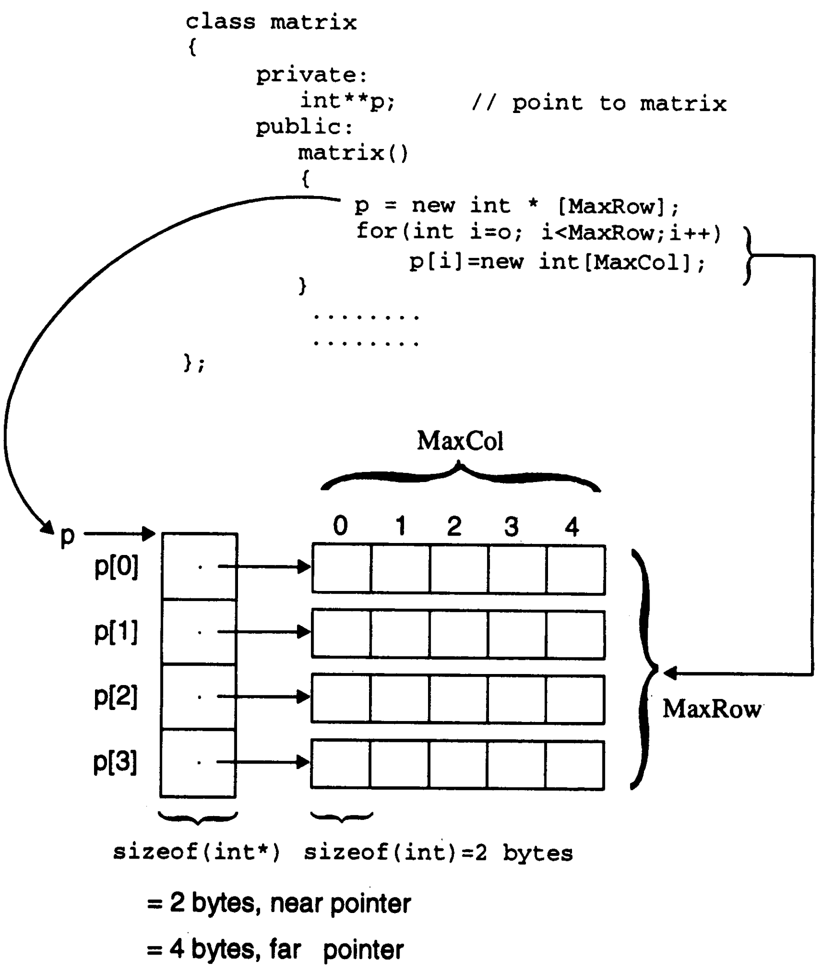
A class can have multidimensional arrays as data members. Their size can be either statically defined or dynamically varied during runtime. A matrix class can be designed to contain data members for storing matrix elements, which are created dynamically. The program `matrix.cpp` illustrates the method of constructing a matrix of size `MaxRow x MaxCol`. It has member functions to perform various matrix operations such as addition, subtraction, etc. The destructor releases memory allocated to the matrix whenever an object of the class `matrix` goes out of scope.

```
// matrix.cpp: Matrix manipulation class with dynamic resource allocation
#include <iostream.h>
#include <process.h>
const int TRUE = 1;
const int FALSE = 0;
class matrix
{
   private:
      int MaxRow;     // number of rows
      int MaxCol;     // number of columns
      int **p; // pointer to 2 dimensional array
   public:
      matrix()
      {
         MaxRow = 0; MaxCol = 0;
         p = NULL;
      }
      matrix( int row, int col );
      ~matrix();
      void read();
      void show();
      void add( matrix &a, matrix &b );
      void sub( matrix &a, matrix &b );
      void mul( matrix &a, matrix &b );
      int eql( matrix &b );
};
matrix::matrix( int row, int col )     // constructor
{
   MaxRow = row;
   MaxCol = col;
   p = new int *[ MaxRow ];   // dynamic allocation
   for( int i = 0; i < MaxRow; i++ )
      p[i] = new int[ MaxCol ];
}
matrix::~matrix()                      // destructor
{
   for( int i = 0; i < MaxRow; i++ )
      delete p[i];
   delete p;
}
// addition of matrices, c3.add(c1, c2): c3 = c1+c2
void matrix::add( matrix &a, matrix &b )
{
   int i, j;
   MaxRow = a.MaxRow;
   MaxCol = a.MaxCol;
   if( a.MaxRow != b.MaxRow || a.MaxCol != b.MaxCol )
   {
      cout << "Error: Invalid matrix order for addition";
      exit( 1 );
   }
   for( i = 0; i < MaxRow; i++ )
      for( j = 0; j < MaxCol; j++ )
         p[i][j] = a.p[i][j] + b.p[i][j];
}
// summation of matrices, c3.sub(c1, c2): c3 = c1-c2
void matrix::sub( matrix &a, matrix &b )
{
   int i, j;
   MaxRow = a.MaxRow;
   MaxCol = a.MaxCol;
   if( MaxRow != b.MaxRow || MaxCol != b.MaxCol )
   {
      cout << "Error: Invalid matrix order for subtraction";
      exit( 1 );
   }
   for( i = 0; i < MaxRow; i++ )
      for( j = 0; j < MaxCol; j++ )
         p[i][j] = a.p[i][j] - b.p[i][j];
}
// multiplication of matrices, c3.mul(c1, c2): c3 = c1*c2
void matrix::mul( matrix &a, matrix &b )
{
   int i, j, k;
   MaxRow = a.MaxRow;
   MaxCol = b.MaxCol;
   if( a.MaxCol != b.MaxRow )
   {
      cout << "Error: Invalid matrix order for multiplication";
      exit( 1 );
   }
   for( i = 0; i < a.MaxRow; i++ )
     for( j = 0; j < b.MaxCol; j++ )
     {
        p[i][j] = 0;
        for( k = 0; k < a.MaxCol; k++ )
           p[i][j] += a.p[i][k] * b.p[k][j];
     }
}
// compare matrices
int matrix::eql( matrix &b )
{
   int i, j;
   for( i = 0; i < MaxRow; i++ )
      for( j = 0; j < MaxCol; j++ )
         if( p[i][j] != b.p[i][j] )
            return 0;
   return 1;
}
void matrix::read()
{
   int i, j;
   for( i = 0; i < MaxRow; i++ )
      for( j = 0; j < MaxCol; j++ )
      {
         cout << "Matrix[" << i << "," << j << "] = ? ";
         cin >> p[i][j];
      }
}
void matrix::show()
{
   int i, j;
   for( i = 0; i < MaxRow; i++ )
   {
      cout << endl;
      for( j = 0; j < MaxCol; j++ )
         cout << p[i][j] << " ";
   }
}
void main()
{
   int m, n, p, q;
   cout << "Enter Matrix A details..." << endl;
   cout << "How many rows ? ";
   cin >> m;
   cout << "How many columns ? ";
   cin >> n;
   matrix a( m, n );
   a.read();
   cout << "Enter Matrix B details..." << endl;
   cout << "How many rows ? ";
   cin >> p;
   cout << "How many columns ? ";
   cin >> q;
   matrix b( p, q );
   b.read();
   cout << "Matrix A is ...";
   a.show();
   cout << endl << "Matrix B is ...";
   b.show();
   matrix c( m, n );
   c.add( a, b );
   cout << endl << "C = A + B...";
   c.show();
   matrix d( m, n );
   d.sub( a, b );
   cout << endl << "D = A - B...";
   d.show();
   matrix e( m, q );
   e.mul( a, b );
   cout << endl << "E = A * B...";
   e.show();
   cout << endl << "(Is matrix A equal to matrix B) ? ";
   if( a.eql( b ) )
      cout << "Yes";
   else
      cout << "No";
}
```

Run

```
Enter Matrix A details...
How many rows ? 3
How many columns ? 3
Matrix[0,0] = ? 2
Matrix[0,1] = ? 2
Matrix[0,2] = ? 2
Matrix[1,0] = ? 2
Matrix[1,1] = ? 2
Matrix[1,2] = ? 2
Matrix[2,0] = ? 2
Matrix[2,1] = ? 2
Matrix[2,2] = ? 2
Enter Matrix B details...
How many rows ? 3
How many columns ? 3
Matrix[0,0] = ? 1
Matrix[0,1] = ? 1
Matrix[0,2] = ? 1
Matrix[1,0] = ? 1
Matrix[1,1] = ? 1
Matrix[1,2] = ? 1
Matrix[2,0] = ? 1
Matrix[2,1] = ? 1
Matrix[2,2] = ? 1
Matrix A is ...
2 2 2
2 2 2
2 2 2
Matrix B is ...
1 1 1
1 1 1
1 1 1
C = A + B...
3 3 3
3 3 3
3 3 3
D = A - B...
```

```
1 1 1
1 1 1
1 1 1
E = A * B...
6 6 6
6 6 6
6 6 6
(Is matrix A equal to matrix B) ? No
```



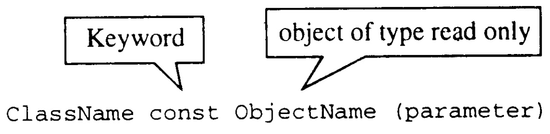
```
class matrix
{
     private:
        int**p;       // point to matrix
     public:
        matrix()
        {
            p = new int * [MaxRow];
            for(int i=o; i<MaxRow;i++)
               p[i]=new int[MaxCol];
        }
        ........
        ........
};
```

Figure 11.9: Constructor creating matrix dynamically

The constructor first creates a *vector pointer* to a list of integers of size `MaxRow`. It then allocates an integer type vector of size `MaxCol` pointed to by each element `p[i]`. Figure 11.9 shows the allocation of memory for the elements of a matrix whose size is `MaxRow x MaxCol` dynamically.

11.13 Constant Objects and Constructor

C++ allows to define constant objects of user-defined classes similar to constants of standard data types. The syntax for defining a constant object is shown in Figure 11.10.



```
ClassName const ObjectName (parameter)
```

Figure 11.10: Constant object creation

The data members of a constant object can be initialized only by a constructor, as a part of object creation procedure. Once a constant object is created, no member functions of its class can modify its data members. They can only read the contents of the data member. Such data members are termed as *read-only data members* and the object is termed as *constant*, or *read-only object*. The `const` objects behave like a ROM (Read Only Memory) of a computer. In such a memory, the data is stored during their fabrication, like constant objects are initialized only by a constructor during its creation. It is illustrated in the program `person.cpp`.

```
// person.cpp: person class with const member functions
#include <iostream.h>
#include <string.h>
class Person
{
   private:
      char *name;             // name of person
      char *address;          // address field
      char *phone;            // telephone number
   public:
     Person( char *NameIn, char *AddressIn, char *PhoneIn );
     ~Person();
     // functions to set fields
     void Person::changename( char const *NameIn );
     // functions to inspect fields
     char const *getname(void) const;
     char const *getaddress(void) const;
     char const *getphone(void) const;
};
// constructor
void Person::Person( char *NameIn, char *AddressIn, char *PhoneIn )
{
   name = new char[ strlen( NameIn )+1];
   strcpy( name, NameIn );
   address = new char[ strlen( AddressIn )+1];
   strcpy( address, AddressIn );
   phone = new char[ strlen( PhoneIn )+1];
   strcpy( phone, PhoneIn );
}
```

```
// destructor, release memory allocated to class data members
inline void Person::~Person()
{
   delete name;
   delete address;
   delete phone;
}
// interface functions get...()
inline char const *Person::getname() const
{
   return name;
}
inline char const *Person::getaddress() const
{
   return address;
}
inline char const *Person::getphone() const
{
   return phone;
}
void Person::changename( char const *NameIn )
{
   if( name )
      delete name;
   name = new char[ strlen( NameIn )+1 ];
   strcpy( name, NameIn );
}
void printperson( Person const &p )
{
   if( p.getname() )
      cout << "Name    : " << p.getname() << endl;
   if( p.getaddress() )
      cout << "Address: " << p.getaddress() << endl ;
   if( p.getphone() )
      cout << "Phone   : " << p.getphone() << endl;
}
void main()
{
  Person const me("Rajkumar","E-mail: raj@cdacb.ernet.in",
        "91-080-5584271");
   printperson( me );
   Person you( "XYZ", "-not sure-", "-not sure-" );
   cout << "You XYZ by default..." << endl;
   printperson( you );
   you.changename( "ABC" );
   cout << "You XYZ changed to ABC ..." << endl;
   printperson( you );
}
```

Run

```
Name    : Rajkumar
Address: E-mail: raj@cdacb.ernet.in
```

```
Phone  : 91-080-5584271
You XYZ by default...
Name   : XYZ
Address: -not sure-
Phone  : -not sure-
You XYZ changed to ABC ...
Name   : ABC
Address: -not sure-
Phone  : -not sure-
```

The above program shows how a constant object of the class `Person` can be defined. At the point of the definition of an object, the data fields are initialized (this is the action of the constructor). Following the definition,

```
        Person const me("Rajkumar", "raj@cdacb.ernet.in", "91-080-5584271");
```

it would be illegal to try to redefine the name, address, or phone number for the object `me`; hence, the statement

```
        me.setname("Bill Gates");
```

would not be accepted by the compiler. Generally, it is a good habit to define objects and member functions, which do not modify their data as constant type.

11.14 Static Data Members with Constructors and Destructors

Each object of a class has its own public or private data members, which are accessible only to its member functions. In certain situations, it is desirable to have one or more common data fields, which are accessible to all the objects of the class. An example of such a situation is to keep track of the status of *how many objects of a class* are created and *how many of them* are currently active in the program. Based on the number of objects present, some specific initialization has to be performed; only the first object of the class would then perform the initialization and set the flag to *done*.

The use of static data members with constructors and destructors is illustrated by the program `graph.cpp`. It has a class called `Graphics`, which defines the communication of a program with a graphics device (such as EGA or VGA screen). The initial preparation of the device, i.e., switching from text mode to graphics mode, is an action of the constructor and depends on a static flag variable `nobjects`. The variable `nobjects` simply counts the number of objects of the class `Graphics` present at that time. Similarly, the destructor of a class may switch back from graphics mode to text mode when the last graphical object ceases to exist.

```
// graph.cpp: keeps count of how many objects are created
#include <iostream.h>
class Graphics
{
  private:
    // counter of number of objects
    static int nobjects;
    // hypothetical functions to switch to graphics
    // mode or back to text mode
    void setgraphicsmode ()
    {}
```

```
      void settextmode ()
       {}
   public:
      // constructor, destructor
      Graphics ();
      ~Graphics ();
      //other interface is not shown here,to draw lines, or circles etc.
      int get_count() const
       {
          return nobjects;
       }
};
// the constructor
Graphics::Graphics ()
 {
   if (! nobjects)
      setgraphicsmode ();
   nobjects++;
}
// the destructor
Graphics::~Graphics ()
{
   nobjects—;
   if (! nobjects)
      settextmode ();
}
void my_func()
{
   Graphics obj; // nobject is incremented by its constructor
   cout<<"\nNo. of Graphics Object's while in my_func = "<<obj.get_count();
} // obj goes out of scope, destructor is called
// the static data member
int Graphics::nobjects = 0;//global:if not defined generates linker error
void main()
{
   Graphics obj1;
   cout<<"No. of Graphics Object's before my_func = "<<obj1.get_count();
   my_func();
   cout<<"\nNo. of Graphics Object's after my_func = "<<obj1.get_count();
   Graphics obj2, obj3, obj4;
   cout<<"\nValue of static member nobjects after all 3 more objects...";
   cout << "\nIn obj1 = " << obj1.get_count();
   cout << "\nIn obj2 = " << obj2.get_count();
   cout << "\nIn obj3 = " << obj3.get_count();
   cout << "\nIn obj4 = " << obj4.get_count();
}
```

Run

```
No. of Graphics Object's before my_func = 1
No. of Graphics Object's while in my_func = 2
No. of Graphics Object's after my_func = 1
Value of static member nobjects after all 3 more objects...
In obj1 = 4
In obj2 = 4
In obj3 = 4
In obj4 = 4
```

The purpose of the variable `nobjects` is to count the number of objects of the class `Graphics`, which exist at a given time. When the first object is created, the graphics device is initialized. When the last object is destroyed, the switch from graphics mode to text mode is made. The statement

```
int Graphics::nobjects = 0;
```

defines and initializes the static data member. If this statement is missing, the linker will generate the error: `undefined Graphics::nobjects symbol.`

It is obvious that when the class `Graphics` defines more than one constructor, each constructor would need to increment the variable `nobjects` and possibly would have to initialize the graphics mode. The constructor

```
Graphics::Graphics ()
```

increments the variable `nobjects` by one and the destructor

```
Graphics::~Graphics ()
```

decrements the variable `nobjects` by one. Therefore, for every object created, the variable `nobjects` is incremented by one and whenever an object of the class `Graphics` goes out of scope, the variable `nobjects` is decremented by one.

11.15 Nested Classes

The power of abstraction of a class can be increased by including other class declarations inside a class. A class declared inside the declaration of another class is called *nested class*. Nested classes provide classes with non-global status. Host and nested classes follow the same access rules for members that exist between non-nested classes. Nested classes could be used to hide specialized classes and their instances within a host class.

A member of a class may itself be a class. Such nesting enables building of very powerful data structures. The `Student` class can be enhanced to accommodate the date of birth of a student. The new member data type `date` is a class by itself as shown below:

```
class Student
{
   private:
      int roll_no;
      char name[25];
      char branch[15];
      int marks;
   public:
      class date
      {
         int day;
         int month;
         int year;
```

```
        public:
           date()
           {
             ... // initializing members of date class
           }
           read();
           ..... // other member functions of date class
        } birthday;   // instance of the nested date class
        Student()
        {
           ....// initialize members of Student class
        }
        ~Student()
        {
           ....
        }
        read()
        {
          //read members of Student class's object including 'birthday'
          cin >> roll_no;
           ....
          birthday.read();  // accessing member of a nested class date
        }
        .... // other member functions of Student class
};
```

The embedded class `date` is declared within the enclosing class declaration. An object of type `Student` can be defined as follows:

```
Student s1;
```

The year in which the student `s1` was born can be accessed as follows:

```
s1.birthday.year
```

A statement such as,

```
s1.date.day = 2;      // error
```

is invalid, because members of the nested class must be accessed using its object name.

The feature of nesting of classes is useful while implementing powerful data structures such as linked lists and trees. For instance, the stack data structure can be implemented having a node data member which is an instance of another class (node class).

Review Questions

11.1 What are constructors and destructors ? Explain how they differ from normal functions.

11.2 What are the differences between default and parameterized constructors ?

11.3 What are copy constructors and explain their need ?

11.4 What is the order of construction and destruction of objects ?

11.5 What are read-only objects ? What is the role of constructor in creating such objects ?

11.6 State which of the following statements are TRUE or FALSE. Give reasons.

(a) Constructors must be explicitly invoked.
(b) Constructors defined in private section are useful.
(c) Constructors can return value.
(d) Destructors are invoked automatically.
(e) Destructors take input parameters.
(f) Destructors can be overloaded.
(g) Constructors cannot be overloaded.
(h) Constructors can take default arguments.
(i) Data members of nameless objects can be initialized using constructors only.
(j) Constructors can allocate memory during runtime.
(k) A class member function can take its class's objects as value arguments.
(l) Constant objects can be initialized by using constructors only.
(m) Data members of a class can be initialized at the point of their definition.

11.7 Consider a class called MyArray having pointer to integers as its data member. Its objects must appear like arrays, but they must be dynamically re-sizable. Write a program to illustrate the use of constructors in MyArray class.

11.8 Write a program to model Time class using constructors.

11.9 Distinguish between the following two statements:

```
String name( "Smrithi" );
String name = "Smrithi";
```

11.10 Declare a class called `String`. It must have constructors which allow definition of objects in the following form: (The class `String` has data member `str` of type `char *`)

```
String name1;       // str points to NULL
String name2 = "Minu"; // one-argument constructor is invoked
String name3 = name2; //one-argument constructor taking String object
```

Write a program to model String class and to manipulate its objects. The destructor must release memory allocated to the `str` data member by its counterpart.

11.11 Create a class, which keeps track of the number of its instances. Use static data member, constructors, and destructors to maintain updated information about active objects.

12

Dynamic Objects

12.1 Introduction

C++ takes the middle ground between languages (such as C and Pascal) which support dynamic memory allocation (discussed in the chapter *Pointers and Runtime Binding*) and languages (like Java), in which all variables are dynamically allocated. C++ supports creation of objects with scoped lifetimes (stack-based objects) and with arbitrary lifetimes (heap-based objects). Stack-based objects are managed by the compiler implicitly, whereas heap-based objects are managed by the programmers explicitly.

C++ is different from C because it not only allocates memory for an object, but also initializes them. Thus when a dynamic object is created, it creates a *live object*, and not just a chunk of memory big enough to hold the object. It is initialized with necessary data at runtime. Unlike dynamic memory allocation which just allocates memory, dynamic object creation supported by C++ allocates and initializes objects at runtime.

A class can be instantiated at runtime and objects created by such instantiation are called *dynamic objects*. The lifetime of dynamic objects in C++ (which is allocated from heap memory—the free store) is managed explicitly by the program. The program must guarantee that each dynamic object is deleted when it is no longer needed, and certainly before it becomes garbage. (There is no garbage collection in standard C++, and few programs can afford to produce garbage.) For each dynamic allocation, a policy that determines the objects's lifetime must be found by the programmer and implemented. These policies used in managing dynamic objects will be discussed at the end of this chapter. The lif-time of an object in C++ is the interval of time it exists by occupying memory. Creation and deletion of objects as and when required, offers a great degree of flexibility in programming.

Objects with scoped lifetimes are created in the stack memory. Stack memory is a store house which holds local variables or objects, and whenever they go out of scope, the memory allocated for them in the stack is released automatically. *Objects with arbitrary lifetimes* are created in the heap memory. These dynamic objects can be created or destroyed as and when required, explicitly by the programmer. The operators `new` and `delete` used with standard data type variable's management can also be used for creating or destroying objects at runtime respectively.

12.2 Pointers to Objects

The C++ language defines two operators which are specific for the allocation and deallocation of memory. These operators are `new` and `delete`. The `new` operator is used to create dynamic objects and `delete` operator is used to release the memory allocated to the dynamic object by the `new` operator. A pointer to a variable can be defined to hold the address of an object, which is created statically or dynamically. Such pointer variables can be used to access data or function members of a class using the `*` or `->` operators.