Unit 5

Constructors and Destructor

1. Memory Allocation and Static Data Members

When we create an object in a program, memory is allocated for each of its data members. Consider the student class as shown below:

```
class student
{
  int rollno;
  char name[80];
    ...
};
```

When an object is created of the above class, separate memory is allocated for roll number and name of the object.

Example:

```
student s1, s2;
```

Here, memory is allocated for 2 data members of s1 and two data members of s2. However, if the class has static data members (class member), memory is not allocated separately for the static member for each object.

```
class student
{
   int rollno;
  char name[80];
  static char teachername[80];
     ...
}s1,s2;
```

Here, memory is allocated for two data members of s1 and two data members of s2. Sin teachername is a static field, memory is allocated for only one string of 80 characters irrespective how many objects are created.

The sizeof operator gives the number of bytes allocated for the object.

1.1 Object Initialization

When an object is created, it is important that the object is properly initialized so that its initial sta is well defined. Hence, there should be a mechanism for initializing the object as soon as it created. Once the object is no longer needed, it should be destroyed in a proper manner so that all th resources it uses are properly released. This chapter discusses constructors and destructor which ar used for initializing and destroying objects respectively.

The following program demonstrates an uninitialized object.

Program : Un-initialized Objects

```
#include<iostream.h>
class CRectangle
{
   int x, y;
  public:
     void setdata(int, int);
     int area(void)
     {
        return(x*y);
     }
};
void CRectangle::setdata(int a, int b)
{
   x = a;
   y = b;
}
```

```
int main()
{
  CRectangle recta, rectb;
  recta.setdata(3,4);
  cout << "recta area : " << recta.area() << endl;
  cout << "rectb area : " << rectb.area() << endl;
}
```

Output

recta area: 12
rectb area: -26344

In this *example*, the data members of object recta are assigned values by the setdata function. Hence, the area will be properly calculated. But what about object rectb? Its data members have not been initialized and hence a call to the area() function will result in a garbage value.

In order to avoid this, a class can include a special function called a *constructor*, whose purpose is to properly initialize the object as soon as it is created. Similarly, when your object goes out of scope, the memory used by the object must be reclaimed. C++ provides a special member function, called the *destructor*, which is called whenever your object is destroyed. Before we study how to define constructors and use them, we will study two important memory management operators - new and delete.

1.2 Memory Management Operators - new and delete

In C, dynamic memory allocation and deallocation is done using functions (malloc, calloc, realloc and free). C++ provides two unary operators – *new* and *delete* to allocate and free memory dynamically. They are also called *free store operators* because they operate on the dynamic memory called the free store. *Free store* is a pool of memory available to allocate (and deallocate) storage for objects and data elements during program execution.

Memory is allocated using new and released using the delete operator. Memory allocated explicitly using new will remain allocated until it is explicitly released using delete or until the program ends.

▶ "new" operator

The new operator is used to allocate memory for a data element of any type (built-in or user defined). Its forms are:

```
pointer = new type;
```

or

```
pointer = new type[elements];
```

or

```
pointer = new type(value);
```

i. The first expression is used to allocate memory for a single element of the specified t

Example:

```
int *p ;            //declare pointer
p = new int;        //allocate memory
*p = 20;            //assign value
```

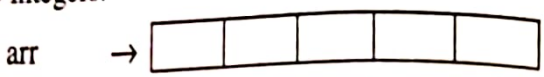
Memory is allocated for one variable of the type int.

ii. The second method is used to allocate memory for a block (an array) of elemer specified *type*.

Example:

```
int *arr;
arr = new int[5];
```

Memory is allocated for 5 integers.



iii. The third form allocates memory for one element and also initializes the memory given value.

Example:

```
int *p;
p = new int(20);
cout << *p;      //displays 20
```

Memory is allocated for one integer and 20 is stored at that position.

> **Note** If the operating system cannot allocate the required number of bytes, an exception will be thrown.

▶ "delete" Operator

When the memory allocated dynamically is no longer needed, it should be freed so that it available for future requests. This can be done by using the delete operator.

It can be used in two ways:

```
delete pointer;
```

or

```
delete[] pointer;
```

The first expression is used to delete memory allocated for a single element, and the secon when memory is allocated for multiple elements (arrays).

Examples:

```
1.  int *p;                 //initialize pointer
    p = new int;            //allocate memory
    ...
    delete p;               //free memory

2.  int *arr;
    arr = new int[5];       //allocate memory for 5 integers
    ...
    delete[] arr;           //free memory
```

The following program shows how we can create an array of n integers dynamically and free the memory after use.

Program : new and delete

```
#include<iostream.h>
int main()
{
  int *p, n;
  cout<<"How many elements:";
  cin>>n;
  p = new int[n];          //allocate memory
  cout<<"Enter "<<n<<" values ";
  for(int i=0; i<n ; i++)
  cin>> p[i];
  cout<<"You have entered :";
  for(i=0; i<n ; i++)
  cout<<" "<<p[i];
  delete[] p;    //release memory
  return 0;
}
```

Output

```
How many elements: 5
Enter 5 values
10  20  30  40  50
You have entered:
10  20  30  40  50
```

▶ Comparison of dynamic memory management in C and C++

The following table compares dynamic memory allocation and de-allocation mechanisms in C and C++.

	Memory Management In C	Memory Management In C++
1.	Dynamic memory management is done using functions like malloc, calloc, realloc and free.	Dynamic memory management is done using operators new and delete.
2.	Functions require additional time and memory to execute.	new and delete are operators; hence they execute faster.
3.	We have to include a header file to use these functions.	Operators are built-in. Hence, there is no need to include any header file.
4.	The sizeof operator is used to calculate the exact number of bytes for allocation.	The compiler automatically calculates the memory size required.
5.	The returned address has to be typecast into the correct pointer type.	There is no need of typecasting. The returned address is of the correct type.
6.	The allocated memory can only be initialized to 0 (using calloc).	We can initialize the memory to any value.
7.	It is not possible to overload functions in C.	new and delete can be overloaded in C++.
8.	malloc and free simply do memory allocation and de-allocation. They do not call the class constructor and destructor.	new and delete automatically invoke the class constructor and destructor respectively.
9.	malloc returns NULL on failure.	'new' throws an exception object of type 'std::bad_alloc'.
10.	malloc and free do not have special forms for arrays; the request is for a specific size of raw memory in bytes.	new and delete have array forms new[] and delete[].

2. Constructors

A constructor is a special member function of a class whose purpose is to initialize an object as as it is created.

It is defined as a non-static member function of a class, having the same name as that of the which automatically gets executed whenever an object of that class comes into existence.

2.1 Rules for Defining Constructors

1. Constructors have the same name as that of the class to which they belong.
2. Constructors do not return any values (not even void).
3. Constructors are invoked automatically when the objects are created.

4. They cannot be declared static, const or volatile.
5. Constructors should have public or protected access and in special circumstances, they can be private.
6. Constructors cannot be virtual or static.
7. Constructors can have default arguments.
8. Constructors can be overloaded.

We shall now see how to declare and define a constructor of a class.

▶ Constructor Declaration

```
class-name();
        OR
class-name(arguments);
```

▶ Constructor Definition

```
class-name::class-name()
{
   // code
}
                OR
class-name::class-name(arguments)
{
   // code
}
```

Like any other member function, there are two ways of defining constructors:

1. Declare and define within the class.
2. Declare within class, define outside the class.

For example

```
class fraction
{
  int numerator, denominator;
  public:
     fraction();    //declaration
     fraction(int n, int d); //declaration
     ...
};
```

```
fraction::fraction()     //constructor defined
{
   numerator = 0;
   denominator = 1;
}
fraction::fraction(int n, int d)     //constructor defined
{
   numerator = n;
   denominator = d;
}
```

2.2 Invoking a Constructor

There are two ways in which the constructor of a class is invoked when an object is created:

1. **Implicit invocation**: A constructor will be automatically called by the compiler as soo object is created. There is no need to give a call to the constructor like we call an member function.

   Syntax:

   ```
   class-name object-name;
                or
   class-name object-name(values);
   ```

   Examples:

   i. `fraction f1; //calls first constructor`

   ii. `fraction f2(5,6); //calls second constructor`

2. **Explicit invocation:** The constructor can also be explicitly called like a regular functi only during object creation and not later. Since the constructor implicitly returns the that is being created, we can assign this object to an object on the left hand side of assig operator. However, explicitly invoking the constructor is not a common practice.

   Syntax:

   ```
   class-name object-name = class-name();
                or
   class-name object-name = class-name(values);
   ```

   Examples:

   i. `fraction f1 = fraction(); //calls first constructor`

   ii. `fraction f2 = fraction(5,6); //calls second constructor`

2.3 Types of Constructors

Constructors come in many forms. These types are:

1. Default constructor
2. Parameterized constructor
3. Constructor with default values
4. Copy constructor
5. Dynamic constructor

1. **Default Constructor:** A default constructor is a constructor that does not take any arguments, whether it is supplied automatically by the compiler or defined by the user.

It is not mandatory that the user must define a constructor for a class. If the class does not have any constructor defined, then the compiler automatically supplies a constructor. A class may also have a constructor without any arguments. Such a constructor is called a default constructor.

Declaration Syntax

```
class-name();
```

Definition Syntax

```
class-name::class-name()
{
  //initialize data members
}
```

Example:

The following program creates a class fraction and defines two constructors for the class.

Program : Default Constructor

```
#include<iostream.h>
class fraction
{
    int numerator, denominator;
    public:
       fraction();          //constructor
       void display();

};
fraction::fraction()     //constructor defined
 {
      numerator = 0;
      denominator = 1;

 }
```

```
void fraction::display()
{
   cout << numerator <<"/" << denominator << endl;
  int main()
  {
    fraction f1;
    f1.display();
    return 0;
  }
```

Output
0/1

In the above program, the fraction class has a default constructor which initializes numerator and denominator to 0 and 1 respectively.

2. **Parameterized Constructor:** We may need to initialize the data members of an object with some values when the object is created. These values can be passed to the constructor.

A constructor which takes arguments is called a parameterized constructor. The arguments values are used to initialize the data members of the object.

Declaration Syntax

```
class-name(argument-list);
```

Definition Syntax

```
class-name::class-name(argument-list)
{
 //initialization using argument values
}
```

Example

Program : Parameterized Constructor

```
#include<iostream.h>
class fraction
{
  int numerator, denominator;
  public:
     fraction(int n, int d); //constructor
     void display();
};
fraction::fraction(int n, int d)    //constructor defined
{
     numerator = n;
     denominator = d;
}
```

```
void fraction::display()
{
     cout << numerator <<"/" << denominator << endl;
}
int main()
{
    fraction f1(5,4);
    f1.display();
    return 0;
}
```

Output
5/4

Constructor with Default values: In the previous chapter, we have seen that function arguments can be assigned default values. If the user does not provide values in the function call, the default values will be used.

Default values can be assigned to arguments in a parameterized constructor. If the user supplies values during object creation, those values will be used, otherwise the default values will be assigned to the object.

Declaration Syntax

```
class-name(argument-list with default values);
```

Definition Syntax

```
class-name::class-name(argument-list)
{
  //initialization using argument values
}
```

Example:

Consider the fraction class described above. We have defined two constructors: one default and one parameterized. The default constructor initializes numerator and denominator to 0 and 1 respectively. The parameterized constructor assigns user given values to numerator and denominator. We could combine these two constructors into a single parameterized constructor which has default values of 0 and 1.

Program : Constructor with Default Values

```
#include<iostream.h>
class fraction
{
   int numerator, denominator;
 public:
  fraction(int n =0, int d =1);     //default values
```

```
void display();
};
fraction::fraction(int n, int d)    //constructor defined
{
  numerator = n;   denominator = d;
}
void fraction::display()
{
  cout << numerator <<"/" << denominator << endl;
}
int main()
{
  fraction f1, f2(5), f3(3,7);
  f1.display();
  f2.display();
  f3.display();
 return 0;
}
```

Output

```
0/1
5/1
3/7
```

4. **Copy Constructor**: Sometimes, we may want to create an object of a class which is to another object of the same class. A copy constructor is a special type of construct creates a new object using another object of the same class i.e. it creates a copy of the object. The object is passed as a parameter to the constructor. The constructor uses th of the object to initialize the object being created. The syntax is given below.

Declaration Syntax

```
class-name(const class-name& object);
```

Definition Syntax

```
class-name::class-name(const class-name& object)
{
   // assign values of object
   // to the data members
}
```

Note The object is passed to the constructor by reference and declared const so that it cannot be modified.

For example, to create object f2 using an existing fraction object f1:

```
fraction f1(5,6);  //creates object f1
fraction f2(f1);   //creates f2 using f1
```

The following program illustrates a copy constructor.

Program : Copy Constructor

```
#include<iostream.h>
class fraction
{
  int numerator, denominator;
  public:
     fraction(int n = 0, int d = 1);  //default values
     fraction(const fraction& f);  //copy constructor
  void display();
};
fraction::fraction(int n, int d) //constructor defined
{
     numerator = n;   denominator = d;
}
fraction::fraction(const fraction& f)   //copy constructor
{
     numerator = f.numerator;
     denominator = f.denominator;
}
void fraction::display()
{
     cout << numerator <<"/" << denominator << endl;
}
int main()
{
     fraction f1(5,4);
     fraction f2(f1);
     f1.display();
     f2.display();
     return 0;
}
```

Output

```
5/4
5/4
```

5. **Dynamic Constructor:** An important task that a constructor performs is memory allocation for the object. We can use a constructor to dynamically allocate memory for an object when it is created. A constructor which allocates memory for the object dynamically is called a *dynamic constructor*.

Dynamic constructor uses the "new" operator to allocate memory dynamically.

Let us consider the following class outline:

```
class intArray
{
  int *ptr;
  int size;
     ...
};
```

The class intArray has two data members: an integer pointer and an integer called size. This class represents an integer array of a user-defined size. The number of elements are passed to the constructor. We have to dynamically allocate memory for the pointer. Hence, we need a dynamic constructor.

The dynamic constructor will be declared as follows:

```
intArray(int s);
```

The dynamic constructor will be defined as follows:

```
intArray::intArray(int s)
{
     size = s;
     ptr = new int[size];  //allocate memory using new
}
```

The class outline will look like this:

```
class intArray
{
  int *ptr;
  int size;
public:
     intArray(int s=5); // dynamic constructor-default size 5
     void accept();
     void display();
};
```

We can create objects of this class in two ways:

```
intArray a1;        //creates an array of size 5
intArray a2(20);    //creates an array of size 20
```

The program is given below:

Program : Dynamic Constructor

```
#include<iostream.h>
class intArray
{
  int *ptr;
  int size;
  public:
    intArray(int s=5); // dynamic constructor
    void accept();
    void display();
 };
intArray::intArray(int s)
{
  size = s;
  ptr = new int[size];  //allocate memory
}
void intArray::accept()
{
  cout<<"Enter "<< size <<" values :";
  for(int i=0; i<size;i++)
  cin >> ptr[i];
}
void intArray::display()
{
  cout<<"The values are: ";
  for(int i=0; i<size; i++)
  cout << "  "<< ptr[i];
}
int main()
{
  intArray a1 ;
  a1.accept() ;
  a1.display() ;
  intArray a2(10) ;
  a2.accept() ;
  a2.display() ;
  return 0;
}
```

Output

Enter 5 values: 1 2 3 4 5
The values are: 1 2 3 4 5
Enter 10 values: 7 2 3 1 6 9 3 4 -3 5
The values are: 7 2 3 1 6 9 3 4 -3 5

2.4 Overloaded Constructors in a Class

So far, we have seen different types of constructors. Since a constructor is a member funct
class, it can be overloaded just like any other function. Hence, a class may implement som
these constructors depending on the requirement. Each constructor will differ in the numb
of arguments.

Consider the following class declaration:

```
class intArray
{
  int *ptr;
  int size;
  public:
     intArray(int s=5);          //dynamic constructor with default
     intArray(int arr[], int n); //parameterized constructor
     intArray(const intArray& obj); //copy constructor
   void accept();
   void display();
   int max();  //returns the maximum element
   int min();  //returns the minimum element
};
```

The class has three constructors. The first one is a dynamic constructor with default valu
second takes two arguments – an array of integers and the number of integers, the third i
constructor. The following program creates three objects and performs operations on the obj

Program : Multiple Constructors

```
#include<iostream.h>
class intArray
{
   int *ptr;
   int size;
   public:
        intArray(int s=5); // dynamic constructor
        intArray(int arr[], int n); //parameterized constructor
        intArray(const intArray& obj); //copy constructor
        void accept();
        void display();
        int max(); //returns maximum
        int min(); //returns minimum
```

```
intArray::intArray(int s)
{
   size = s;
   ptr = new int[size];
}
intArray::intArray(int arr[], int n)
{
   size = n;
   ptr = new int[size];
   for(int i=0; i<size; i++)
   ptr[i] = arr[i];
}
intArray::intArray(const intArray& obj)
{
   size = obj.size;
   ptr = new int[size];
   for(int i=0; i<size; i++)
   ptr[i] = obj.ptr[i];
}
void intArray::accept()
{
   cout<<"Enter "<< size <<" values :";
   for(int i=0; i<size;i++)
   cin >> ptr[i];
}
void intArray::display()
{
   cout<<"The values are :";
   for(int i=0; i<size; i++)
   cout << "  "<< ptr[i];
   cout<<endl;
}
int intArray::max()
{
   int m = ptr[0];
   for(int i=1;i<size;i++)
   if(ptr[i] > m)
        m = ptr[i];
    return m;
}
int intArray::min()
{
   int m = ptr[0];
```

```
  for(int i=1;i<size;i++)
    if(ptr[i] < m)
      m = ptr[i];
    return m;
}
int main()
{
  intArray a1(6);
  a1.accept() ;
  a1.display() ;
  cout << "Maximum is "<<a1.max()<<endl;
  cout << "Minimum is "<<a1.min()<<endl;
  int a[4] = {10,20,30,40};
  intArray a2(a,4);
  a2.display() ;
  intArray a3(a2);
  a3.display();
  return 0;
}
```

Output

```
Enter 6 values:
14 22 3 49 95 18
The values are:
14 22 3 49 95 18
Maximum is 95
Minimum is 3
The values are:
10 20 30 40
The values are:
10 20 30 40
```

3. Destructor

A destructor is a special member function which is automatically invoked when an object is destroyed. It performs clean-up operations like releasing memory, closing a connection, closing files etc. The destructor is the opposite of a constructor.

Like a constructor, the destructor is also a member function whose name is the same as the class name but it is preceded by tilde (~). Even if the class does not contain an explicit destructor, the compiler automatically supplies a destructor for the class.

Declaration Syntax:

```
~class-name();
```

Definition Syntax:

```
class-name::~class-name()
{
   //code
}
```

Rules for Writing A Destructor Function

1. The destructor must have the same name as the class with a tilde (~) as prefix.
2. It does not take any arguments nor returns any value.
3. It cannot be declared as static, volatile or const.
4. It takes no arguments and therefore cannot be overloaded i.e. a class can only have one destructor.
5. It should be declared public.

Note A class can only have one destructor i.e. a destructor cannot be overloaded.

The following program shows object creation and destruction using constructor and destructor respectively.

Program : Constructor and Destructor

```
#include<iostream.h>
class intArray
{
    int *ptr;
    int size;
  public:
     intArray(int s=5); // dynamic constructor
    ~intArray();  //destructor
    void accept();
    void display();

};
intArray::intArray(int s)
{ cout<<"Object constructed"<<endl;
  size = s;
  ptr = new int[size];

}
intArray::~intArray()
{ cout<<"Object destroyed";
  delete[] ptr;

}
void intArray::accept()
{ cout<<"Enter "<< size <<" values :";
  for(int i=0; i<size;i++)
  cin >> ptr[i];

}
```

```
void intArray::display()
{
   cout<<"The values are :";
   for(int i=0; i<size; i++)
   cout << "  "<< ptr[i];
   cout<<endl;
}
int main()
{
   intArray a1(6);
   a1.accept();
   a1.display();
   return 0;
}
```

Output

```
Object constructed
Enter 6 values:
1  2  3  4  5  6
The values are:
1  2  3  4  5  6
Object destroyed
```

We can effectively use constructors and the destructor to keep a track of how many class objects were created and how many have been destroyed. This will tell us how many objects exist in memory at any given point of time.

Program : Program to Count Objects

```
#include<iostream.h>
class A
{
    int num;
    static int count;          //static data member
  public:
    A(int n = 0);              //constructor
    ~A();                      //destructor
    static int getcount(); //static member function
};
int A::count = 0; //initialize static member
A::A(int n)
{
    num = n;
    count++;
    cout<<"Object created with value "<< num <<endl;
}
A::~A()
{
   --count;
    cout<<"Object destroyed"<<endl;
}
int A::getcount()
{
   return count;
}
```

Output

```
Object created with value 0
Object created with value 60
Total objects = 2
Object created with value 100
Total objects = 3
Object destroyed
Total objects = 2
Object destroyed
Object destroyed
```

```
int main()
{
  A a1, a2(60);
  cout<<"Total objects = "<<A::getcount()<<endl;
   {
     A a3(100);
    cout<<"Total objects = "<<A::getcount()<<endl;
  }
  cout<<"Total objects = "<<A::getcount()<<endl;
  return 0;
}
```

Solved Examples

1. Class 'Fraction' having integer data members numerator and denominator with appropriate constructors and four member functions for addition, subtraction, multiplication and division of fraction objects. The fraction should be stored in its reduced form.

```
#include<iostream.h>
class fraction
{
  int numerator, denominator;
  public:
        fraction(int n = 0, int d = 1);     //default values
        void display();
        fraction add(fraction);
        fraction subtract(fraction);
        fraction multiply(fraction);
        fraction divide(fraction);

  private:
        int gcd();

  };
  fraction::fraction(int n, int d)    //constructor defined
  {
    numerator = n;   denominator = d;
    if(d!=1)
    {
      int g = gcd();
      numerator = numerator/g;
      denominator = denominator/g;

    }
```

```
}
int fraction::gcd()
{
     int a=numerator, b=denominator;
     while(a!=b)
     {
       if(a>b)   a=a-b;
       else      b=b-a;
     }
     return a;
}
void fraction::display()
{
     cout << numerator <<"/"<< denominator << endl;
}
fraction fraction::add(fraction obj)
{
     fraction temp(numerator*obj.denominator +
     denominator*obj.numerator , denominator*obj.denominator);
     return temp;
}
fraction fraction::subtract(fraction obj)
{
     fraction temp(numerator*obj.denominator -
     denominator*obj.numerator , denominator*obj.denominator);
     return temp;
}
fraction fraction::multiply(fraction obj)
{
     fraction temp(numerator*obj.numerator ,
     denominator*obj.denominator);
     return temp;
}
fraction fraction::divide(fraction obj)
{
     fraction temp(numerator*obj.denominator,
     denominator*obj.numerator);
     return temp;
}
int main()
{
  fraction f1(5,2), f2(3,8), f3, f4, f5, f6;
  f3 = f1.add(f2);
  f4 = f1.subtract(f2);
```

```
    f5 = f1.multiply(f2);
    f6 = f1.divide(f2);
    cout<<"The addition is: ";
      f3.display();
    cout<<"The subtraction is: ";
      f4.display();
    cout<<"The multiplication is: ";
      f5.display();
    cout<<"The division is: ";
      f6.display();
    return 0;
}
```

Output

```
The addition is: 23/8
The subtraction is: 17/8
The multiplication is: 15/16
The division is: 20/3
```

2. Write a program to implement the class outline given below. Create and display the object of the String class in different ways.

```
class String
{
  char *str;
  public:
  String(); default constructor
  String(char *s);              //parameterized constructor
  String(char ch, int n);       ////parameterized constructor
  String(const String&);        //copy constructor
  void display();
  ~String();                    //destructor
};
```

```
#include<iostream.h>
#include<string.h>
class String
{
  char *str;
public:
  String();                     //default constructor
  String(char *s);              //parameterized constructor
  String(char ch, int n);       ////parameterized constructor
  String(const String&);        //copy constructor
  void display();
  ~String();
};
```

```
String::String()
{
   str = NULL;
}
String::String(char *s)
{
   str = new char[strlen(s)+1];
   strcpy(str,s);
}
String::String(char ch, int n)
{
  str = new char[n+1];
  for(int i = 0; i<n ; i++)
    str[i]=ch+i;
  str[i] = '\0';
}
String::String(const String& sobj)
{ str = new char[strlen(sobj.str)+1];
  strcpy(str,sobj.str);
}
void String::display()
{ cout<<str<<endl;}
String::~String()
{ delete[] str;
  cout<<"Object destroyed"<<endl;
}
int main()
{
   String s1;
   s1.display();
   String s2("Hello");
   s2.display();
   String s3('a',10);
   s3.display();
   String s4(s3);
   s4.display();
   return 0;
 }
```

Output

```
Hello
abcdefghij
abcdefghij
Object destroyed
Object destroyed
Object destroyed
Object destroyed
```

3. **Class 'sequence' which contains a sequence of strings. The size of each string varies and also the number of strings in a sequence.**

Member functions: display() - display the sequence, and search() - search a string in the sequence, reverse() - reverses every string in the sequence.

Friend functions: union() - gives the union of two sequences, intersection() - sequence containing common strings.

```
#include<iostream.h>
#include<string.h>
class Sequence
{
   char **str;
   int no;
public:
   Sequence(); //default constructor
   Sequence(int n); //parameterized constructor
   Sequence(const Sequence& s);
   void display();
   void append(char *str);
   void search(char *str);
   void reverse();
   friend Sequence union1(Sequence&, Sequence&);
   friend Sequence intersection(Sequence&, Sequence&);
};
Sequence::Sequence()
{ }
Sequence::Sequence(int n)
{
  no=n;
  char temp[20];
       str = new char* [no];
  for(int i=0; i<no; i++)
  {
     cout<<"Enter the string "<<i+1;
     cin>>temp;
     str[i] = new char[strlen(temp)+1];
     strcpy(str[i],temp);
  }
}
Sequence::Sequence(const Sequence& s)
{
  no = s.no;
  str=new char* [no];
  for(int i=0; i<no; i++)
```

```
    {
       str[i]=new char[strlen(s.str[i])+1];
       strcpy(str[i], s.str[i]);
     }
}
void Sequence::display()
{
  cout<<"The sequence is :";
  for (int i=0; i<no; i++)
    cout<<" "<<str[i];
}
void Sequence::search(char *key)
{
  for (int i=0; i<no; i++)
    if(strcmp(str[i],key)==0)
    {
       cout<<" String found at position "<<i;
       return;
     }
  cout<<"String not found";
}
void Sequence::reverse()
{
  char temp[20];
  for(int i=0, j=no-1; i<no/2; i++, j--)
  {
    strcpy(temp, str[i]);
    strcpy(str[i], str[j]);
    strcpy(str[j], temp);
  }
}
Sequence intersection (Sequence& s1, Sequence& s2)
{
  Sequence temp;
  int k=0;
  temp.no=s1.no+s2.no; //max no of strings
  temp.str = new char* [temp.no];
  for(int i=0; i<s1.no; i++)
  for(int j=0; j<s2.no; j++)
  if(strcmp(s1.str[i], s2.str[j])==0)
  {
    temp.str[k]=new char[strlen(s1.str[i])+1];
    strcpy(temp.str[k++],s1.str[i]);
   }
    return temp;
}
```

```
Sequence union1(Sequence& s1, Sequence& s2)
{
  Sequence temp;
  int k=0, flag;
  temp.no=s1.no+s2.no; //max no of strings
  temp.str = new char* [temp.no];
  for(int i=0; i<s1.no; i++)
  {
   temp.str[k]=new char[strlen(s1.str[i])+1];
   strcpy(temp.str[k++],s1.str[i]);
  }
  for(i=0; i<s2.no; i++)
  {
   flag=0;
    for(int j=0; j<s1.no; j++)
    {
       if(strcmp(s1.str[j], s2.str[i])==0)
       {
       flag=1;  break;
       }
     }
     if(flag==0)
     {
       temp.str[k]=new char[strlen(s2.str[i])+1];
       strcpy(temp.str[k++],s2.str[i]);
     }
     }
    temp.no=k;
    return temp;

}
int main()
{
  char key[20];
  Sequence s1(4);
  s1.display();
  cout<<"Enter the string to search :";
  cin>>key;
  s1.search(key);
  s1.reverse();
  Sequence s2(3);
  s2.display();
  Sequence s3=union1(s1,s2);
  s3.display();
  Sequence s4=intersection(s1,s2);
  s4.display();
   return 0;
}
```

EXERCISES

A. Multiple Choice Questions

1. By default, all C++ compilers provide a copy constructor. This constructor is invoked when
   a. An argument is passed by reference to a function
   b. An argument is passed by value.
   c. A function returns a value to an object.
   d. None of the above.

2. Whenever an object is destroyed which function is called?
   a. Dynamic Constructor
   b. Destructor
   c. Copy constructor
   d. Default constructor

3. To request dynamic memory which operator is used?
   a. new
   b. Destructor
   c. delete
   d. malloc

B. Fill in the Blanks

1. A constructor has the same name as that of Class.
2. The constructors that can take arguments are called as parametrized constructor.
3. Destructor is prefix with a tilde (~) character.
4. New operator is used to create a memory space dynamically.
5. delete operator is used to destroy a memory space.

C. State True or False

1. Constructors do not return any values. T
2. Constructors can be virtual or static. F
3. A default constructor is a constructor that may be called without any parameters. T
4. A class can have only one constructor. F
5. Destructor function cannot be overloaded. T
6. The arguments of a constructor cannot be assigned default values. F
7. The purpose of a copy constructor is to copy the contents of one existing object to another existing object. T

D. Review Questions

1. What is constructor? What are the uses of declaring a constructor in a program?
2. Explain the different types of constructors.
3. What is parameterized constructor?
4. What is copy constructor? What is its purpose?
5. What is a default constructor?
6. Explain the new and delete operators.
7. What is destructor?
8. What is dynamic constructor?
9. What are the rules for defining a constructor?

E. Programming Exercises

1. Write a class called Account which contains two private data elements, an integer accountNumber and a floating point accountBalance, and three member functions:
   a. A constructor that allows the user to set initial values for accountNumber and accountBalance and a default constructor that prompts for the input of the values for the above data members.
   b. A function called inputTransaction, which reads a character value for transactionType ('D' for deposit and 'W' for withdrawal), and a floating point value for transactionAmount, which updates accountBalance.
   c. A function called printBalance, which prints on the screen the accountNumber and accountBalance.

2. A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise the message "Required copies not in stock" is displayed.

   Design a system using a class called books with suitable member functions and constructors. Use new operator in constructors to allocate memory space required.

3. Define a class 'Fraction' having integer data members numerator and denominator. Define parameterized and default constructors (default values 0 and 1). Parameterized constructor should store the fraction in reduced form after dividing both numerator and denominator by gcd(greatest common divisor). Write a private function member to compute gcd of two integers.

Write four member functions for addition, subtraction. multiplication and division of fraction objects. Each function will have two fraction objects as arguments. Write the main function to illustrate the use of the class.

4. Write a program to create a list of people(using array) and search for a specific person in the list. Use the class outlines given below.

```
class person
{ char *name;
  public:
  person(char * str);          //constructor
  ~person();                  //destructor
  char * getName();            //accessor
  void display();
};
class people
{ person **array;
int length;
 public:
people(int n=5);            //creates a list of n persons
~people();                  //destructor
int search(char *str);
void display();
};
```

Answers

A.

1. a and c
2. b
3. a

B.

1. Class
2. Parameterized constructors
3. Destructor
4. New operator
5. Delete operator

C.

1. True
2. False
3. True
4. False
5. True
6. False
7. False