Constructors

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is called whenever an object of its class is created. It is called constructor because it construct the values of data member of the class.

A constructor is declared and defines as follow:

```
Class integer
{
        Int m,n;
        Public:
                integer(void);          // constructor declared
                .......
                .......
};

integer :: integer(void)        // constructor defined
{
        m=0;
        n=0;
}
```

The declaration

```
        integer i;      //object created
```

this statement not only create the object I of type integer but also initialize its data member m and n to zero. There is no need to write any statement to call the constructor function.

A constructor that accept no parameter is called the default constructor. The default constructor for class A is A::A(). If no such constructor is defined, then the compiler supplies a default constructor. So the statement

```
        A a;
```

Call the default constructor.

Characteristics of constructor.

- They should be declared in the public section.
- They are called automatically when the objects are created.
- They do not have return type, not even void and therefore they cannot return values.
- They cannot be inherited. A derived class can call the base class constructor.
- They can have default arguments.
- Constructor cannot be virtual.
- We cannot refer to their address.
- They make implicit call to the operators new and delete when memory allocation is required.

Parameterized Constructors

The constructor integer( ), defined above, initializes the data member of all the objects to zero. However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called parameterized constructor.

```
class integer
{
        int m, n;
        public :
                integer (int x, int y);          //      constructor declared
                … …
};

integer :: integer (int x, int y)                //      constructor defined
{
        m = x;
        n = y;
}
```

We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways.

- By calling the constructor explicitly
- By calling a constructor implicitly

```
Example :   integer int1 = integer(0, 100);          //      Explicit call
            Integer int1(0,100);                     //      Implicit call
```

Remember when the constructor is parameterized, we must provide appropriate arguments for the constructor otherwise it may not work if default constructor is not provided. Parameter of constructor can be of any type except of the class itself to which it belongs, but it can accept a reference to its own class s a parameter and such a constructor is called copy constructor.

```
#include<iostream.h>

Class integer
{
        Int m,n;
        Public:
                Integer(int,int);           // constructor declared



                Void display(void)
                {
                        Cout<<"m="<<m<<"\n";
                        Cout<<"n="<<n<<"\n";
                }
};

Integer :: integer(int x, int y)            // constructor defined
{
        m=x;
        n=y;
}

Int main()
{
        Integer imp(0,100);                 // constructor called implicit
        Integer exp = integer(25,75);       // constructor called explicit

        Cout<<"OBJECT 1"<<"\n";
        imp.display();

        Cout<<"OBJECT 2"<<"\n";
        exp.display();

        return 0;
}
```

Multiple Constructors in class

Like normal functions, constructors can also be overloaded.

```
class integer
{
        int m, n;
        public :
                integer ( )             //      default constructor
                {
                        m = n = 0;
                }
                integer (int x, int y)  //      two argument constructor
                {
                        m = x;
                        n = y;
                }
                integer (integer & i )          //      copy constructor
                {
                        m = i.m;
                        n = i.n;
                }
};
```

The declaration of object

```
Integer i1;                     // will call default constructor
Integer i2(20,40);              // will call two argument constructor
Integer i3(i2);                 // will call copy constructor
```

Constructors with Default Arguments

It is possible to define constructors with default arguments. For example, the constructor can be declared as follows:

```
complex (float  real,   float  imag = 0);
```

The default value of the argument imag is zero. Then, the statement

```
complex C(5.0);
```

assigns the value 5.0 to the real variable and 0.0 to imag (by default).

However, the

```
complex  C(2.0,3.0);
```

assigns 2.0 to real and 3.0 to imag.

The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor A::A( ) and the default argument constructor
A::A (int = 0). The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as A a;
The ambiguity is whether to call A::A ( ) or A :: A(int = 0);

Copy Constructor

We briefly mentioned about the copy constructor early. We used the copy constructor

integer(integer &i);

as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare and initialize an object from another object. For example, the statement

integer int12 (int1);

would define the object int2 and at the same time initialize it to the values of int1. Another form of this statement is
integer int2 = int1;

The process of initializing through a copy constructor is known as copy initialization

Remember, the statement int1 = int2; will not invoke the copy constructor and it is the task of assignment operator.

```
class code
{
        Int id;
        Public:
                Code()                  // default constructor
                {
                }
                Code(int a)             // parameterized constructor
                {
                        Id = a;
                }
```

```
        Code(code & x)          // copy constructor
        {
                Id = x.id;
        }

        Void display(void)
        {
                Cout<< id ;
        }
};

Int main()
{
        Code A(100);                    // object A created and initialized
        Code B(A);                      // copy constructor called
        Code C = A;                     // copy constructor called again

        Cout<<"\n id of A = ";
        A.display();

        Cout<<"\n id of B = ";
        B.display();

        Cout<<"\n id of C = ";
        C.display();

        Return 0;
}
```

Dynamic Initialization of Objects

Class objects can be initialized dynamically too. That is to say, the initial value of an object be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment.

Constructing Two-dimensional Arrays

We can construct matrix variables using the class type objects. The example in Program illustrates how to construct a matrix of size m x n.

```
#include <iostream>
class matrix
{
        int **p;                        // pointer to matrix
        int d1,d2;                      //dimensions

        public:
        matrix(int x,  int y);

        void get_element(int i,  int j,  int value)
        {
                p[i][j] = value;
        }

         int & put_e1ement(int  i,  int j)
         {
                return  p[i][j];
         }
};
matrix  :: matrix(int x,  int y)
{
        d1 = x;
        d2 = y;
        p = new int *[d1];
        for(int  i  = 0;  i  < d1;  i++)
                p[i] = new int[d2];   // creates space for each row
}

int main()
{
        int m, n;

        cout « "Enter size of matrix: ";
        cin » m » n;

        matrix A(m,n); // matrix object A constructed

        cout « "Enter matrix elements row by row \n";
```

```
        int i, j, value;
        for(i =0; i < m; i++)
        {
                for(j = 0; j < n; j++)
                {
                        cin » value;
                        A.get_element(i,j,value);
                }

                cout « "\n";
                cout « A.put_element(l,2);
        }
}
```

Dynamic Constructors

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the object is not of the same size, thus resulting in the saving of memory. Allocation of memory to object at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator.

```
#include <iostream.h>
#include <string.h>

class String
{
        char *name;
        int   length;

        public:

                String()        // constructor - 1
                {
                        length = 0;
                        name = new char[length + 1];
                }

                String(char *s)    // constructor - 2
                {
                        length = strlen(s);
                        name = new char[length + 1];
                        strcpy(name s);
                }
```

```
        void display( void)
        {
                cout << name << "\n" ;
        }

        void join (String & a, String & b)
        {
                length = a.length + b.length;
                delete name;
                name = new char[length + 1];
                strcpy(name, a.name);
                strcpy(name, b.name);
        }
};

int main( )
{
        char *first = "kamani";
        String name1(first), name2("science"), name3("college"), s1, s2;

        sl.join (namel, name2);
        s2.join (sl, name3);
        namel.display ( );
        name2.display();
        name3.display();
        sl.display();
        s2.display();

        return 0;
}
```

This Program uses two constructors. The first is an empty constructor that allows us to declare an array of strings. The second constructor initializes the length of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end-of-string character.

The member function join( ) concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string functions strcpy ( ) and strcat ( ). Note that in the function join( ) length and name are members of the object that calls the function, while a.length and a.name are members of the argument object a. The main( ) function program concatenates three strings into one string.

The output is as shown below:

Kamani
Science
College
Kamani science
Kamani science college

Destructors

A destructor as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor for the class integer can be defined as shown below:

```
~integer( )
{  }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case be) to clean up storage that is no longer accessible. It is a good practice to declare destructor in a program since it releases memory space for future use.

Whenever new is used to allocate memory in the constructors, we should use delete to free that memory. For example, the destructor for the matrix class discussed above may be defined as follows:

```
matrix  ::  ~matrix ( )
{
        for (int  i = 0;   i<d1;   i++)
        delete p[i];
        delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

As the objects are created and destroyed they increase and decrease the count. Note that the objects are destroyed in the reverse order of creation.