CHAPTER 6
CONSTRUCTORS AND DESTRUCTORS

Introduction:

- Up to now we have seen examples that uses member functions such as getdata() and setvalues() to provide initial values to the private data member in class.
- **For Ex:** x.getdata(5, 10); or x.setvalues(5, “name”);
- These functions called explicitly using object of same class to initialize values to variables.
- These functions cannot be used to initialize the member variables at the time of creation of their objects.
- Class is user-defined data types similar to built-in types. This means that we should be able to initialize a class type variable (object) when it is declared, much same way as initialization of an ordinary variable.
- Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variables.
- C++ provides a special member function called the **constructor** which enables an object to initialize itself when it is created. This is known as automatic initialization of objects.
- It also provides another member function called the **destructor** which destructs the objects when they are no longer required.

Constructor:

- 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 invoked wherever an object of its associated class is created.
- It is called Constructor because it constructor the values of data member of the class.

A constructor is declared and defined as follows:

```
// class with a constructor
class integer
{
int m, n;
public:
integer(void); // constructor declared
. . . . .
. . . . .
};
integer :: integer(void) // constructor defined
{
m = 0; n = 0;
}
```

- When a class contains a constructor like the one of defined above, it is guaranteed that an object created by the class will be initialized automatically.
- **For example**, the declaration

integer intl; //object intl created

- not only creates the object int1 of type integer but initializes its data members m and n to zero. There is no need to write any statement to invoke the constructor function (as we do with the normal member functions).
- A <u>constructor that accepts no parameters is called the default constructor</u>. The default **constructor for class A is A::A()**. If no such constructor is defined, then the compiler supplies a default constructor. Therefore a statement such as

A a; invokes the default constructor of the compiler to create the object a.

<u>The constructor functions have some special characteristics. These are:</u>

- They should be <u>declared in the public section</u>.
- They are <u>invoked automatically</u> when the objects are created.
- They <u>do not have return types</u>, <u>not even void</u> and therefore, and they cannot return values.
- They <u>cannot be inherited</u>, though a derived class can call the base constructor.
- Like other C++ functions, they <u>can have default arguments</u>.
- Constructors <u>cannot be virtual</u>. (Meaning of virtual will be discussed later).
- We <u>cannot refer to their addresses</u>.
- An object with a constructor (or destructor) <u>cannot be used as a member of a union</u>.
- They <u>make 'implicit calls' to the operators new and delete</u> when memory allocation is required.

<u>Parameterized Constructors:</u>

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

The constructor integer() may be modified to take arguments as shown below:

```
class integer
{
int m, n;
public :
integer(int x, int y); // parameterized constructor
. . . . .
};
integer:: integer(int x, inty)
{
m = x; n = y;
}
```

- When a constructor has been parameterized, the object declaration statement such as

integer int1;

- It may not work. We must pass the initial values as arguments to the constructor function when an object is declared.
- <u>This can be done in two ways:</u>

By calling the constructor explicitly.
By calling the constructor implicitly.

<u>The following declaration illustrates the first method:</u>
integer int1 = integer(0, 100); // explicit call
This statement creates an integer object int1 and passes the values 0 and 100 to it.

<u>The second is implemented as follows:</u>
integer int1(0, 100); // implicit call
This method, sometimes called the <u>shorthand method</u>, is used very often as it is shorter looks better and is easy to implement.

Remember, when the <u>constructor is parameterized</u>, we <u>must provide appropriate arguments</u> for the constructor.
Below Program demonstrates the passing of arguments to the constructor functions.

```
//Class with Parameterized constructors
# 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 int1(0,100); // constructor called implicitly
integer int2 = integer(25,75); // constructor called explicitly
cout << "\nOBJECT1"<<"\n";
int1.display();
cout <<"\n OBJECT2"<<"\n";
int2.display();
return 0;
}
```

<u>OUTPUT:</u>

```
OBJECT 1
m = 0
n = 100
OBJECT 2
m = 25
n = 75
```

The <u>constructor functions</u> can also be <u>defined as</u> <u>inline functions</u>.
For Example,

```
class integer
{
int m, n;
public:
integer (int x, int y) // Inline constructor (Function inside class)
{
m = x; y = n;
}
. . . . .
. . . . .
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs.

```
class A
{
. . . . .
. . . . .
public :
A(A); //illegal
};
```

is illegal.
However, a constructor can accept a reference to its own class as a parameter.

```
Class A
{
. . . . .
. . . . .
public :
A(A&); //legal
};
```

is valid. In such cases, the constructor is called as copy constructor.

Copy Constructor:

A copy constructor is used to declare and initialize an object from another object.
Integer I2 = I1;
The process of initializing through a copy constructor is known as copy initialization.
Remember, the statement
I2 = I1;
will not invoke the copy constructor.
However, if I1 and I2 are objects, this statement is legal and simply assigns the value of I1 to I2 member by member. This is the task of the overloaded assignment operator (=).
A copy constructor takes a reference to an object of the same class as itself as an argument.

//Program that shows copy constructor

```
#include <iostream.h>

class code
{
int id;
public:
code() {} // constructor
code (int a) { id = a; } // constructor again
code(code &x) // copy constructor
{
id = x.id; // copy in the value
}
void display (void)
{
cout << id;
}
};
int main ()
{
code A(100); //object A is created add initialized
code B(A); // copy constructor called
code C = A; // copy constructor called again
code D; // D is created, not initialized
D = A; // copy constructor not called, assignment operator is used
cout << “\n id of A: “;
        A.display ();
cout << “\n id of B: “;
        B.display ();
cout << “\n id of C: “;
        C.display ();
cout << “\n id of D: “;
        D.display ();
return 0;
}
```

OUTPUT:

```
id of A : 100
id of B : 100
id of C : 100
id of D : 100
```

Note: A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor. When no copy constructor is defined the compiler supplier its own copy constructor.

Multiple Constructors in a class:

- We have used two kinds of constructors. They are:

```
integer(); // No arguments
integer(int, int); // Two arguments
```

- In the first case, the constructor itself supplies the data values and no values are passed by the calling program.
- In the second case, the function call passes the appropriate values from main() C++ permits us to use both these constructors in the same class.

For example,

```
class integer
{
int m, n;
        public :
integer() // constructor 1
                {m=0; n=0;}
                integer (int a, int b) // constructor 2
{m = a; n = b;}
                integer (integer & i) // constructor 3
{m = i. m; n = i.n;}
};
```

- This declares three constructors for an integer object. The first constructor receives no arguments, the second receives two integer arguments and the third receives one integer object as an argument.
- **For example,** the declaration
  intger Int1;
- would automatically invoke the first constructor and set both m and n of Int1 to zero. The statement

```
integer Int2(20, 40);
```

- would call the second constructor which will initialize the data members m and n of Int2 to 20 and 40 respectively. Finally, the statement

```
integer Int3(Int2); //copy constructor call
```

- would invoke the third constructor which copies the value of I2 into I3. In other words, it sets the value of every data element of I3 to the value of the corresponding data element of I2. As mentioned earlier, such a constructor is called **the copy constructor**.

- We have learnt that the process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class, we say that the constructor is overloaded.

//Program shows the use of overloaded constructors.

```
# include <iostream.h>

class complex
{
float x, y;
public:
complex () { } // constructor with no org
complex (float a) //constructor with one org
{x = y = a;}
complex (float real, float image) // constructor with two org
{x = real; y = image;}
friend complex sum(complex, complex);
friend void show (complex);
};
complex sum(complex c1, complex c2) // friend
{
complex c3;
c3.x = c1.x + c2.x;
c3.y = c1.y + c2.y;
return(c3);
}

void show (complex c) //friend
{
cout << c.x << " + j" << c.y << "\n";
}
int main()
{
complex A(2.7, 3.5); //define & initialize
complex B(1.6); //define & initialize
complex C; // define
C = sum (A, B); //sum() is a friend
cout << "A = "; show(A); // show() is also friend
cout << "B = "; show(B);
cout << "C = "; show(C);
        // Another way to give initial values (second method)
complex P,Q,R; //define P,Q and R
P = complex (2.5,3.9); // initialize P
        Q = complex (1.6, 2.5); // initialize Q
        R = sum (P,Q);
        cout << "\n";
        cout << "P = "; show(P);
        cout << "Q = "; show(Q);
        cout << "R = "; show(R);
return 0;
}
```

OUTPUT:

```
A = 2.7 + j3.5
B = 1.6 + j1.6
C = 4.3 + j5.1

P = 2.5 + j3.9
Q = 1.6 + j2.5
R = 4.1 + j6.4
```

Note: There are three constructors in the class complex. The first constructor, which takes no argument, is used to create objects which are not initialized; the second which takes one argument, is used to create objects and initialize them; and the third, which takes two arguments, is also used to create objects and initialize them to specific values. Note that the second method of initializing values looks better.

- Let us look at the first constructor again.
  complex () { }
- It contains the empty body and does not do anything. We just stated that this is used to create objects without any initial values called as default constructor.
- Remember, we have defined object(C, c3) in the earlier examples without using such a constructor.
- Why do we need this constructor now? As pointed out earlier, C++ compiler has an implicit constructor which creates objects, even though it was not defined in the class. This works fine as long as we do not use any other constructors in the class.
- However, once we define a constructor, we must also define the "do-nothing" implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.

Constructors with Default Argument:

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

complex (float real, **float image = 0**);

- The default value of the argument image 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 statement

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 (at last position) 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)**

Dynamic Initialization of Object:

- Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time.
- One advantage to 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. It provides different interest rates for different schemes as well as for different periods of investment.
- Program illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

```
// long-term fixed deposit system
#include <iostream.h>
#include<conio.h>

class Fixed_deposit
{
long int P_amount; // principal amount
int Years; // period of investment
float Rate; // Interest rate
float R_value; // Return value of amount
public:
Fixed_deposit() {}
Fixed_deposit (long int p, int y, float r=0.12);
Fixed_deposit (long int p, int y, int r);
Void display (void)
};
Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
P_amount = p;
Year = y;
Rate = r;
R_value = P_amount;
for(int i = l; i<= y; i++)
```

```
R_value = R_value * (1.0 + r);
}
Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
P_amount = p;
Year = y;
Rate = r;
R_value = P_amount;
for(int i=1; i<=y; i++)
R_value = R_value * (1.0 + float(r) / 100);
}
void fixed_deposit :: display (void)
{
cout << “\n”;
cout<< “Principal amount = “<< P_amount<< “\n”;
cout<< “Return Value = “<< R_value<< “\n”;
}
int main()
{
Fixed_deposit FD1, FD2, FD3; // deposits created
long int p; // principal amount
int y; // investment period, years
float r; // interest rate, decimal form
int R; // interest rate, percent form
cout << “Enter amount, period, interest rate(in percent) “<< “\n”;
cin >> p >> y >> R;
FD1 = Fixed_deposit(p,y,R);
Cout << “Enter amount, period, interest rate (in decimal form) “<<” \n”;
cin >> p >> y >> r;
FD2 = Fixed_deposit(p,y,r);
        Cout << “Enter amount and period“<<” \n”;
cin >> p >> y;
FD3 = Fixed_deposit(p,y);
        Cout << “/nDepsoit 1”;
FD1.display(();
cout << “\nDeposit 2”;
FD2.display();
cout << “\nDeposit 3”;
FD3.display();
return 0;
}
```

OUTPUT:

```
Enter amount, period, interest rate(in percent)
10000 3 18
Enter amount, period, interest rate (in decimal form)
```

```
10000 3 0.18
Enter amount and period
10000 3
Deposit 1
Principal Amount = 10000
Return Value = 16430.3
Deposit 2
Principal Amount = 10000
Return Value = 16430.3
Deposit 3
Principal Amount = 10000
Return Value = 14049.3
```

The program uses three overloaded constructors. The parameter values to these constructors are provided at run time. The user can provide input in one of the following forms;

1. Amount, period and interest in decimal form.
2. Amount, period and interest in period form.
3. Amount and period

Note : Since the constructors are overloaded with the appropriate parameters, the one that matches the input value is invoked. For example, the second constructor is invoked for the forms (1) and (3) and the third is invoked for the form (2). Note that for form (3), the constructor with default argument is used. Since input to the third parameter is missing it uses the default value for r.

Dynamic Constructors:

- The constructor 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 objects are not of the same size, thus resulting in the saving of memory.
- Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.
- The memory is allocated with the help of the **new** operator. And memory will be released with the help of **delete** operator.

//Program shows the use of new, in constructors that are used to construct strings in objects.

```
#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[lengh + 1]; // one additional
// character for \0
Strcpy (name, s);
}
void display (void)
{cout << “\nName : ” <<name;}
void join(string &a, String &b)
};
void String : : join (string &a, String &b)
{
length = a.length + b. length;
delete name; //remove allocated memory
name = new char [length +1] // dynamic allocation of memory
strcpy(name, a.name);
strcat(name, b.name);
};
int main()
{
char *first = “Joseph”;
String name (first), name2 (“Louis ”) ,s1,s2;
s1.join (name, naem2);
s2.join (s1, “Language”);
name1.display();
name2.display();
s1.display();
s2.dispaly();
return 0;
}
```

<u>OUTPUT:</u>

```
Joseph
Louis
Joseph Louis
Joseph Louis Lagrange
```

Note: 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 '\0'.

- 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 function 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 one by one.

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() {} //Constructor
~ integer () {} //Destructor

- A destructor never takes any argument nor does it return any value.
- It will be invoked implicitly by compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible.
- It is a good practice to declare destructors in a program since it releases memory space for future use.
- Where **new** is used to allocate memory in the constructors, we should use **delete** to free that memory.

- For example, the destructor for the String class discussed above may be defined as follows:

```
String :: ~String()
{
delete name;
}
```

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

// Program to illustrate that the destructor has been invoked implicitly by the complier.

```
# include <iostream.h>
int count = 0;
class alpha
{
public:
alpha()
{
count++;
count << "\nNo. Of object created "<< count;
}
~alpha()
{
count << “\nNo. Of object destroyed ”<< count;
count - -;
}
};
int main()
{
cout << “\n\nENTER MAIN\n”;
alpha A1, A2, A3, A4;
{
cout << “\n\nENTER BLOCK 1\n”;
alpha A5;
}
{
cout << “\n\nENTER BLOCK 2\n”;
alpha A6;
}
cout << “\n\nRE-ENTER MAIN\n”;
return 0;
}
```

OUTPUT:

```
Enter MAIN

No. of object created 1
No. of object created 2
No. of object created 3
No. of object created 4

ENTER BLOCK 1
No. of object created 5
No. of object destroyed 5

ENTER BLOCK 2
No. of object created 5
No. of object destroyed 5

RE-ENTER MAIN
No. of object destroyed 4
No. of object destroyed 3
No. of object destroyed 2
No. of object destroyed 1
```

Note: As the object are created and destroyed, they increase and decrease the count. Notice that after the first group of object created A5 is created and then destroyed; A6 is created and then destroyed. Finally, the rest of the objects are also destroyed. When the closing brace of a scope is encountered the destructors for each object in the scope are called.
Note that the objects are destroyed in the reverse order of creation.

Constant Member Function & Object:

- We can also make member function as a constant member function and object as constant object same as constant variables using const keyword.
- Declaring a member function with the "const" keyword specifies that the function is a "read-only" function that does not modify the object for which it is called.
- To declare a constant member function, place the "const" keyword after the closing parenthesis of the argument list.
- The "const" keyword is required in both the declaration and the definition. A constant member function cannot modify any data members or call any member functions that aren't constant.
- Whenever user tries to modify data members of class in constant member function compiler generates error.

- We may create and use **constant objects** using "const" keyword before object declaration.
- For example, we may create S as a constant object of the class String as follows:

comst String s("E-Balaguru"); //object s is constant

- A constant object can call only const member functions otherwise compiler will generate warning.

//Program to illustrate constant member function and constant object

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

class test
{
int a,b;
public:
test() { }

test(int x,int y)
{a=x; b=y;}

void add() const;
};
void test::add() const
{
a=9; b=5; //It gives error because it will not allow to change value of a & b
        //Put above statement in comment and then execute program.
```

```
	cout<<”\nAddition of“<<a <<” & ”<<b<<” is : “;
cout<<a+b;
}
void main()
{
clrscr();
	const test t( 7 , 4 ); //Constant Object (can call constant function)
t.add(); //Call to constant function
getch();
}
```

OUTPUT:

```
Addition of 7 & 4 is : 11
```

NOTE: Use const whenever possible. Declare member functions to be const whenever they should not change the object. This lets the compiler help you find errors; it's faster and less expensive than doing it yours