CHAPTER-4

Constructor and Destructors:

Definition of constructors & its uses

Types of constructors: default constructor, parameterized constructor, copy constructor, constructor with dynamic allocation, Dynamic Constructors

Constructor Overloading

Destructors

Constructors

What is constructor?

A constructor is a member function of a class which initializes objects of a class. In C++,Constructor is automatically called when object(instance of class) create.It is special member function of the class.

How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Types of Constructors

Default Constructors: Default constructor is the constructor which doesn't take any argument. It has no parameters.

```
// Cpp program to illustrate the concept of Constructors
#include <iostream>
using namespace std;

class construct
{
public:
    int a, b;

        // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};
```

```
int main()
{
 // Default constructor called automatically when the object is created
 construct c;
    cout << "a: "<< c.a << endl << "b: "<< c.b;
    return 1; }
```

Output:

a: 10

b: 20

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly. The default value of variables is 0 in case of automatic initialization.

Parameterized Constructors: It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
// CPP program to illustrate  parameterized constructors
#include<iostream>
using namespace std;
class Point
{
    private:
        int x, y;
    public:
        // Parameterized Constructor
        Point(int x1, int y1)
        {
            x = x1;
            y = y1;
        }
        int getX()
        {
            return x;
        }
        int getY()
        {
            return y;
        }   };
 int main()
{
    // Constructor called
    Point p1(10, 15);
     // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
     return 0;
}
```

Output:

p1.x = 10, p1.y = 15

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
Example e = Example(0, 50); // Explicit call

Example e(0, 50);           // Implicit call
```

Uses of Parameterized constructor:

It is used to initialize the various data elements of different objects with different values when they are created.

It is used to overload constructors.

Can we have more than one constructors in a class?

Yes, It is called Constructor Overloading.

Copy Constructor:

What is a copy constructor?

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

Syntax:

```
ClassName (const ClassName &old_obj);
```

Following is a simple example of copy constructor.

```
#include<iostream>
using namespace std;
 class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()            {  return x; }
    int getY()            {  return y; }
};
```

```
int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```

Output:

p1.x = 10, p1.y = 15

p2.x = 10, p2.y = 15

When is copy constructor called?

1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When compiler generates a temporary object.

When is user defined copy constructor needed?

If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member wise copy between objects. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any run time allocation of resource like file handle, a network connection..etc.

Copy constructor vs Assignment Operator

Which of the following two statements call copy constructor and which one calls assignment operator?

```
MyClass t1, t2;
MyClass t3 = t1;  // calls copy Constructor
t2 = t1;          // calls (operator =) function   t2.operator=(t1)
```

Shallow vs Deep Copies

A shallow copy of an object copies all of the member field values. This works well if the fields are values, but may not be what you want for fields that point to dynamically allocated memory. The pointer will be copied. but the memory it points to will not be copied -- the field in both the original object and the copy will then point to the same dynamically allocated memory, which is not usually what you want. The default copy constructor and assignment operator make shallow copies.

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. To make a deep copy, you must write a copy constructor and overload the assignment operator, otherwise the copy will point to the original, with disasterous consequences.

Deep copies need ...

If an object has pointers to dynamically allocated memory, and the dynamically allocated memory needs to be copied when the original object is copied, then a deep copy is required.

A class that requires deep copies generally needs:

A constructor to either make an initial allocation or set the pointer to NULL.

A destructor to delete the dynamically allocated memory.

A copy constructor to make a copy of the dynamically allocated memory.

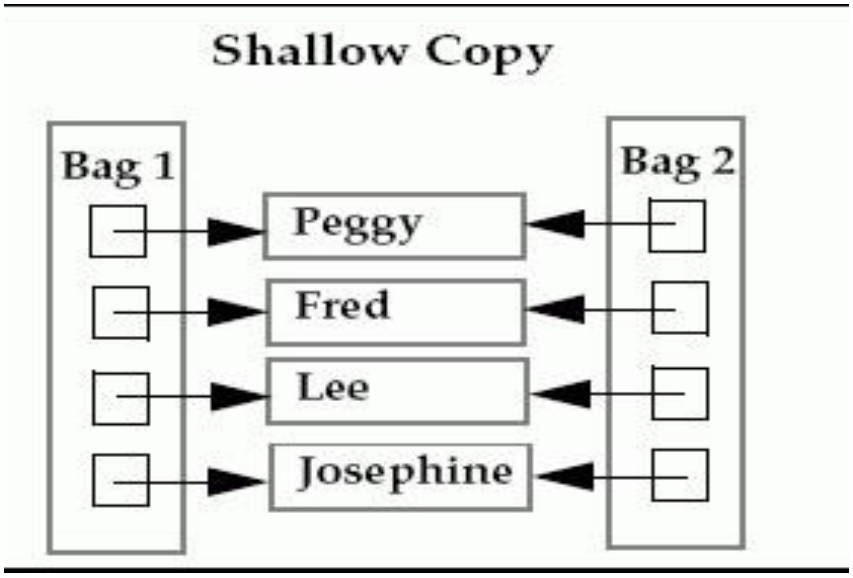
An overloaded assignment operator to make a copy of the dynamically allocated memory.

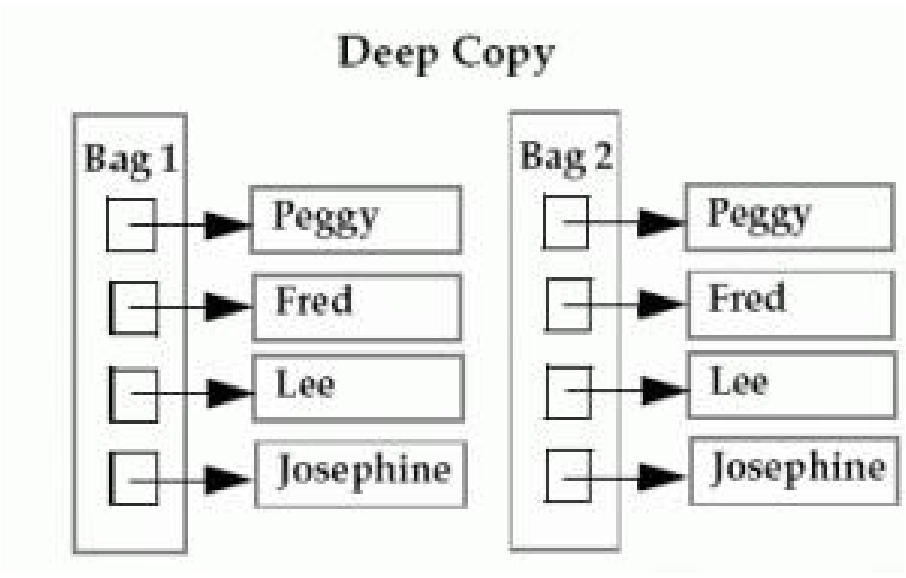
Lets take an example

Shallow Copy: It makes a copy of the reference to X into Y. Think about it as a copy of X's Address. So, the addresses of X and Y will be the same i.e. they will be pointing to the same memory location.

Deep copy: It makes a copy of all the members of X, allocates different memory location for Y and then assigns the copied members to Y to achieve deep copy. In this way, if X vanishes Y is still valid in the memory.

The correct term to use would be cloning, where you know that they both are totally the same, but yet different (i.e. stored as two different locations in the memory space).





Constructor Overloading

- In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments.This concept is known as Constructor Overloading and is quite similar to function overloading.
- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.

While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

```
// Example of overloaded constructors
#include <iostream>
using namespace std;
class Area
{
   private:
      int length;
      int breadth;
   public:
      // Constructor with no arguments
      Area(): length(5), breadth(2) { }

      // Constructor with two arguments
      Area(int l, int b): length(l), breadth(b){ }
      void GetLength()
      {
          cout << "Enter length and breadth
respectively: ";
          cin >> length >> breadth;
      }
      int AreaCalculation() {  return length *
breadth;  }
};
```

```
int main()
{
    Area A1, A2(2, 1);
    int temp;

    cout << "Default Area when no argument is
passed." << endl;
    temp = A1.AreaCalculation();
   cout << "Area: " << temp << endl;

    cout << "Area when (2,1) is passed as
argument." << endl;
    temp = A2.AreaCalculation();
    cout << "Area: " << temp << endl;

    return 0;
}
```

Output

```
Default Area when no argument is passed.
Area: 10
Area when (2,1) is passed as argument.
Area: 2
```

C++ Dynamic Constructor

Allocation of memory during the creation of objects can be done by the constructors too. The memory is saved as it allocates the right amount of memory for each object.

Allocation of memory to object at the time of their construction is known as Dynamic Constructor of objects. In the dynamic constructor, new operator is used for allocation of memory.

```
#include<iostream>
#include<string.h>
using namespace std;
class dynamic
{
  char *name;
  int length;
  public :
    dynamic()
    // First Constructor
    {
      length = 0;
      name = new char[length + 1];
    }
dynamic(char *s)
    // Second Constructor
    {
      length = strlen(s);
      name = new char[length + 1];
      strcpy(name,s);
    }
void show()
    // Method to display name using dynamic allocation in join method
    {
      cout << name << "\n";
      cout << "Number of characters in the string is " << strlen(name) << "\n\n";
    }

void join(dynamic &a, dynamic &b)
    {
      length = a.length + b.length;
      delete name;
      name = new char[length + 1];
      // dynamic allocation of name using new
      strcpy(name, a.name);
      strcat(name, b.name);
 }
  };
```

```
int main ()
{
  char *first = "Hello!";
  dynamic name1(first);
  dynamic name2("Technology");
  dynamic name3("Lovers");

  dynamic s1, s2;
  s1.join(name1, name2);
  s2.join(s1, name3);

  name1.show();
  name2.show();
  name3.show();

  return 0;
}
```

output:

```
Hello!
Number of characters in the string is 6
Technology
Number of characters in the string is 10
Lovers
Number of characters in the string is 6
```

Destructors

What is destructor?

Destructor is a member function which destructs or deletes an object.

When is destructor called?

A destructor function is called automatically when the object goes out of scope:

(1) the function ends

(2) the program ends

(3) a block containing local variables ends

(4) a delete operator is called

How destructors are different from a normal member function?

Destructors have same name as the class preceded by a tilde (~)

Destructors don’t take any argument and don’t return anything

```
//program to demonstrate destructors
#include<iostream>
using namespace std;
class String
{
private:
    char *s;
    int size;
public:
    String(char *); // constructor
    ~String();      // destructor
};

String::String(char *c)
{
    size = strlen(c);
    s = new char[size+1];
    strcpy(s,c);
}
```

```
String::~String()
{
    delete s;
}
int main ()
{
  String *s1=new String("Hello World");
  s1->display();
  delete s1;

  return 0;
}
```

output:

```
Hello World
```

Can there be more than one destructor in a class?

No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

When do we need to write a user-defined destructor?

If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

Can a destructor be virtual?

Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function.