Dynamic Memory Allocation

What is memory allocation?

The memory in C++ can be allocated in two ways:

- Static Memory Allocation:

  In static memory allocation the memory is allocated for the named variables by the compiler. The size and data type of the storage must be known at the compile time.

- Dynamic Memory Allocation:

  In the dynamic memory allocation the memory is allocated during run time. The space which is allocated dynamically usually placed in a program segment which is known as heap. In this, the compiler does not need to know the size in advance.

<u>Memory in your C++ program is divided into two parts:</u>

- stack: All variables declared inside any function takes up memory from the stack.
- heap: It is the unused memory of the program and can be used to dynamically allocate the memory at runtime.

What is Dynamic Memory Allocation?

- In C++, dynamic memory allocation means performing memory allocation manually by programmer. It is allocated on the heap and the heap is the region of a computer memory which is managed by the programmer using pointers to access the memory.
- The programmers can dynamically allocate storage space while the program is running but they cannot create a new variable name.

Dynamic memory allocation requires two criteria:

- Creating the dynamic space in memory.
- Storing its address in a pointer

Use of Dynamic Memory Allocation :

- Dynamic Memory Allocation is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use of dynamic memory allocation is the flexibility as the programmers are free to allocate and deallocate memory whenever we need and when we don’t.

How memory is allocated and deallocated in C++?

In C++, we can allocate and deallocate memory by using two operators new and delete operator respectively which perform the task of allocating and deallocating the memory.

New Operator

The new operator signifies a request for memory allocation on the heap and if the sufficient memory is available then the new operator initializes the memory and returns the address of the newly allocated variable and initialized memory to the pointer variable.

Syntax:

```
datatype pointername = new datatype
```

Here, the pointer-variable is the pointer of type data type. Data type could be any built in data type including array or any user defined data type including class and structure.

For example:

```
int *new_op = new int;
// allocating block of memory
int *new_op = new int[10];
```

Delete Operator

The delete operator is used to deallocate the memory created by new operator at runtime. Once the memory is no longer needed it should be free so that the memory becomes available again for other request of dynamic memory.

Syntax:

```
delete pointer-variable;
```

In the delete operator the pointer-variable is the pointer that points to the data object created by the new.

For example:

```
delete new_op;
```

Example

```
#include <iostream>
using namespace std;

int main()
{
   double* val = NULL;
   val = new double;
   *val = 38184.26;
   cout << "Value is : " << *val << endl;
   delete val;
}
```

C++ Constructors

Constructor is a special member function of the class which is invoked automatically when new object is created. The purpose of constructor is to initialize private data items of the related class. Following are the main features of constructors.

- Constructor has the same name as is the name of class.
- Constructors do not have any return type.
- Constructors are usually declared in public section of the class so that they can be invoked outside the class, when an object is created.
- Constructors can be private also but private access modifier will hide that constructor from outside the class. Private constructors can be called in class itself and in friend functions only.
- There can be more than one constructors in a class known as constructor overloading.
- In case no constructor is defined a default constructor is executed by the compiler.
- Default customer do not take any parameter.
- Constructor defined by the programmer may or may not take the parameters.
- Constructors can also be defined as inline.
- Constructors can have any type of arguments except its own class type. But the constructor can accept a reference to its own class type object as a parameter.
- We cannot refer to their address.

In C++, a constructor has the same name as that of the class and it does not have a return type. For example,

```
class  Wall {
  public:

   // create a constructor
   Wall() {
      // code
   }
};
```

Here, the function Wall() is a constructor of the class Wall. Notice that the constructor

- has the same name as the class,
- does not have a return type, and
- is public

C++ Default Constructor

- A constructor with no parameters is known as a **default constructor**.
- Any constructor without parameters or when all the parameters have default values is known as a default constructor. It can be auto generated or user defined, but in both the cases it must be called without parameters.
- If no constructor is defined by the programmer then the compiler implicitly creates a default constructor without body i.e. without any corresponding statement.
- Any class can have at the most one default constructor.
- Default constructors are also known as implicit constructor.

In the example above, Wall() is a default constructor.

// C++ program to demonstrate the use of default constructor

```
#include <iostream>
using namespace std;

// declare a class
class  Wall {

  private:
      double length;

   public:
    // create a constructor
    Wall() {

        // initialize private variables
        length = 5.5;

        cout << "Creating a wall." << endl;
        cout << "Length = " << length << endl;
    }
};

int main() {

    // create an object
    Wall wall1;

    return 0;
}
```

Output

```
Creating a Wall
Length = 5.5
```

Here, when the *wall1* object is created, the Wall() constructor is called. This sets the *length* variable of the object to 5.5.

C++ Parameterized Constructor

- In C++, a constructor with parameters is known as a parameterized constructor. This is the preferred method to initialize member data.
- This type of constructors are defined with list of arguments. Values for arguments are assigned at the time of creating the objects. Program 1 demonstrates the parameterized constructors.
- The parameterized constructor can be called implicitly or explicitly, for example

  v customer_id c1(555, 250, 20);

  //constructor called implicitly

  v customer_id c2 = customer_id(111, 300, 21);

  // constructor called explicitly

// C++ program to calculate the area of a wall

```
#include <iostream>
using namespace std;

// declare a class
class Wall {
  private:
   double length;
   double height;

  public:
   // create parameterized constructor
   Wall(double len, double hgt) {
      // initialize private variables
      length = len;
      height = hgt;
   }

   double calculateArea() {
      return length * height;
   }
};

int main() {
   // create object and initialize data members
   Wall wall1(10.5, 8.6);
   Wall wall2(8.5, 6.3);
```

```
    cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
    cout << "Area of Wall 2: " << wall2.calculateArea() << endl;

    return 0;
}
```

Output

```
Area of Wall 1: 90.3
Area of Wall 2: 53.55
```

Here, we have created a parameterized constructor Wall() that has 2 parameters: double len and double hgt. The values contained in these parameters are used to initialize the member variables *length* and *height*.

When we create an object of the Wall class, we pass the values for the member variables as arguments. The code for this is:

```
Wall wall1(10.5, 8.6);
Wall wall2(8.5, 6.3);
```

With the member variables thus initialized, we can now calculate the area of the wall with the calculateArea() function.

C++ Copy Constructor

- The copy constructor in C++ is used to copy data of one object to another.
- Copy constructor is used to declare and initialize an object from some already existing object of the class. As discussed above constructors can have any type of arguments except its own class type. But the constructor can accept a reference to its own class type object as a parameter.
- Copy constructor takes a reference to the object of same class as an argument. Copy constructor cannot accept argument by value.

C++ Copy Constructor

```
#include <iostream>
using namespace std;

// declare a class
class Wall {
  private:
   double length;
   double height;

  public:

   // parameterized constructor
   Wall(double len, double hgt) {
      // initialize private variables
      length = len;
```

```
        height = hgt;
    }

    // copy constructor with a Wall object as parameter
    Wall(Wall &obj) {
        // initialize private variables
        length = obj.length;
        height = obj.height;
    }
    double calculateArea() {
        return length * height;
    }
};

int main() {

    // create an object of Wall class
    Wall wall1(10.5, 8.6);

    // print area of wall1
    cout << "Area of Wall 1: " << wall1.calculateArea() << endl;

    // copy contents of wall1 to another object wall2
    Wall wall2 = wall1;

    // print area of wall2
    cout << "Area of Wall 2: " << wall2.calculateArea() << endl;

    return 0;
}
```

Copy constructor is used to

- To initialize an object using another object
- To copy an object to pass it as an argument by value to a function
- Also to copy the object to return it from a function by value.

Output

```
Area of Wall 1: 90.3
Area of Wall 2: 90.3
```

In this program, we have used a copy constructor to copy the contents of one object of the Wall class to another. The code of the copy constructor is:

```
Wall(Wall &obj) {
    length = obj.length;
    height = obj.height;
}
```

Notice that the parameter of this constructor has the address of an object of the Wall class.

We then assign the values of the variables of the first object to the corresponding variables of the second object. This is how the contents of the object are copied.

In main(), we then create two objects *wall1* and *wall2* and then copy the contents of the first object to the second with the code

Wall wall2 = wall1;

C++ Dynamic Constructor

When allocation of memory is done dynamically using dynamic memory allocator new in a constructor, it is known as **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.

Example 1:

```
#include <iostream>
using namespace std;

class geeks {
        int a=10;

public:
        // default constructor
        geeks()
        {
                // allocating memory at run time
                cout<<"Hello";
        }
        void display()
        {
                cout << a << endl;
        }
};

int main()
{
        geeks *obj = new geeks();
        obj->display();
}
```

Example 2:

```
#include <iostream>
```

```
using namespace std;
class geeks
{
        int* p;
        public:
            geeks()
                 {
                 p = new int[3]{ 1, 2, 3 };

                 for (int i = 0; i < 3; i++) {
                 cout << p[i] << " ";
                 }
       cout << endl;
               }
};

int main()
{
     geeks* ptr = new geeks[5];
}
```

C++ Constructor Overloading

Constructors can be overloaded in a similar way as function overloading.
Overloaded constructors have the same name (name of the class) but the different number of arguments. Depending upon the number and type of arguments passed, the corresponding constructor is called. So , The constructor overloading has few important concepts.

- Overloaded constructors must have the same name and different number of arguments
- The constructor is called based on the number and types of the arguments are passed.
- We have to pass the argument while creating objects, otherwise the constructor cannot understand which constructor will be called.

example:
```
number();  //no arguments
number(int);  //one arguments
number(int, int); //two arguments
 number(int, float); //differed by the datatype of arguments
```

```
#include <iostream>
using namespace std;

class construct
{

public:
    float area;

    // Constructor with no parameters
    construct()
    {
```

```
                area = 0;
            }

            // Constructor with two parameters
            construct(int a, int b)
            {
                area = a * b;
            }

            void disp()
            {
                cout<< area<< endl;
            }
        };

        int main()
        {
            construct o;
            construct o2( 10, 20);
            o.disp();
            o2.disp();
            return 1;
        }
```

C++ Constructor with Default Argument

Like other functions in C++, constructors can also be defined with default arguments. If an arguments are initialized at the time of constructor declaration/definition, then that type of constructors are called default argument constructor.
The initialization of arguments at the time of constructor declaration must be in trailing order. And the missing arguments must be the trailing ones.

Program: Illustrate the default argument constructor

```
#include<iostream>
usage namespace std;

class  power
   {
                int  num;
                int  pwr;
                int  ans;
        public:
                power(int   n = 3 , int  p = 2 );
                void  show( )
                 {
                        cout<< num <<"power"<< pwr <<"is:"<< ans ;
                 }
    };
  power :: power( int  n,int p)
    {
            num = n;
            pwr = p;
            ans = pow( n , p )   // math pow function
        }
    int  main( )
     {
            clrscr( );
            power  p1, p2(5);
            p1.show( );
            p2.show( );
            return 0;
        }
```

Destructor

Destructor functions are inverse of constructor functions. They are invoked automatically when objects are destroyed. An objects gets destroyed when its scope of existence finishes or when the delete operator is used.
Like constructor, destructor also follows some rules:

- Destructors are special member functions of the class and they are used to free the memory allocated to object.
- Destructors are functions with same name as class and they do not accept any parameters.
- Even constructors have same name as class, but name of destructor is preceded by '~' (tilde).
- Like constructor, destructors do not have any return type. Not even void
- The Destructor of class is automatically called when the object goes out of scope.
- If there exists no user defined constructor in a class, but is required then the compiler itself declares a destructor implicitly.
- Destructors cannot be inherited.
- The primary usage of a destructor is to clean up and release memory space for future use
- Whenever 'new' operator is used in the constructors to allocate memory, we should use 'delete' to free that memory space.

For Example :

```
* program to illustrate, automatic execution of constructor and destructor */
                #include<iostream>
                using namespace std;
                class test
                {
                        char code;
                        public:
                        test( char c)
                        // constructor defined
                {
                        code = c;
                        cout<<"\n object with code = " << code <<" is
                        created";
                }
                        ~ test ( )
                        // destructor defined
```

```
        {
        cout<<”\n object with code = “<<code<<”is
        destroyed”;
}
};
        // end of class
int main( )
{
        test obj(a), obj(b), obj(c);
        cout<<”\n three object created”;
        cout<<”\n the main program ends here”;
        return 0;
}
```