Class Constructor:

A class constructor is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explains the concept of constructor:

```
#include <iostream>

class Line

{

  public:

    void setLength( double len );

    double getLength( void );

    Line();  // This is the constructor

  private:

    double length;

};

// Member functions definitions including constructor

Line::Line(void)

{

   cout << "Object is being created" << endl;

}

void Line::setLength( double len )

{

   length = len;

}
```

```
double Line::getLength( void )
{
   return length;
}
// Main function for the program
int main( )
{
  Line line;
  // set line length
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;
  return 0;
}
```

Parameterized Constructor:

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
#include <iostream>
class Line
{
  public:
    void setLength( double len );
    double getLength( void );
    Line(double len);  // This is the constructor
  private:
    double length;  };
```

```
// Member functions definitions including constructor
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}
void Line::setLength( double len )
{
    length = len;
}
double Line::getLength( void )
{
    return length;
}
// Main function for the program
int main( )
{
   Line line(10.0);
   // get initially set length.
   cout << "Length of line : " << line.getLength() <<endl;
   // set line length again
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;
   return 0;
}
```

Using Initialization Lists to Initialize Fields:

In case of parameterized constructor, you can use following syntax to initialize the fields:

```
Line::Line( double len): length(len)
{
   cout << "Object is being created, length = " << len << endl;
}
```

Above syntax is equal to the following syntax:

```
Line::Line( double len)
{
   cout << "Object is being created, length = " << len << endl;
   length = len;
}
```

If for a class C, you have multiple fields X, Y, Z, etc., to be initialized, then use can use same syntax and separate the fields by comma as follows:

```
C::C( double a, double b, double c): X(a), Y(b), Z(c)
{
  ....
}
```

Copy Constructor

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one.If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here:

```
classname (const classname &obj) {
   // body of constructor
}
```

Here, obj is a reference to an object that is being used to initialize another object.

```
#include <iostream>
class Line
{
   public:
      int getLength( void );
      Line( int len );             // simple constructor
      Line( const Line &obj);  // copy constructor
      ~Line();                     // destructor
   private:
      int *ptr;
};
// Member functions definitions including constructor
Line::Line(int len)
{
    cout << "Normal constructor allocating ptr" << endl;
    // allocate memory for the pointer;
    ptr = new int;
    *ptr = len;
}
```

```
Line::Line(const Line &obj)
{
    cout << "Copy constructor allocating ptr." << endl;
    ptr = new int;
   *ptr = *obj.ptr; // copy the value
}
Line::~Line(void)
{
    cout << "Freeing memory!" << endl;
    delete ptr;
}
int Line::getLength( void )
{
    return *ptr;
}
void display(Line obj)
{
   cout << "Length of line : " << obj.getLength() <<endl;
}
// Main function for the program
int main( )
{
   Line line(10);
   display(line);
   return 0; }
```

Let us see the same example but with a small change to create another object using existing object of the same type:

```
#include <iostream>

class Line

{

  public:

    int getLength( void );

    Line( int len );            // simple constructor

    Line( const Line &obj);  // copy constructor

    ~Line();                    // destructor

  private:

    int *ptr;

};

// Member functions definitions including constructor

Line::Line(int len)

{

   cout << "Normal constructor allocating ptr" << endl;

   // allocate memory for the pointer;

   ptr = new int;

   *ptr = len;

}

Line::Line(const Line &obj)

{

   cout << "Copy constructor allocating ptr." << endl;

   ptr = new int;
```

```
   *ptr = *obj.ptr; // copy the value
}
Line::~Line(void)
{
    cout << "Freeing memory!" << endl;
    delete ptr;
}
int Line::getLength( void )
{
    return *ptr;
}
void display(Line obj)
{
   cout << "Length of line : " << obj.getLength() <<endl;
}
// Main function for the program
int main( )
{
   Line line1(10);
   Line line2 = line1; // This also calls copy constructor
   display(line1);
   display(line2);
   return 0;
}
```