Constructor

- A constructor is a special type of member function of a class which initializes objects of a class.
- In C++, Constructor is automatically called when object (instance of class) creates.
- It is special member function of the class because it does not have any return type.
- Constructor has same name as the class itself.
- It must be placed in public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for object expects no parameters and has an empty body.
- **Types of Constructors:**
  1. Default Constructor
  2. Parameterized Constructor
  3. Copy Constructor

1. Default Constructor:

- A constructor without any arguments or with default value for every argument, is said to be ***default constructor***.
- Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.
- Compiler defined default constructor is required to do certain initialization of class internals. It will not touch the data members or plain old data types (aggregates like an array, structures, etc.). However, the compiler generates code for default constructor based on the situation.
- Consider a class derived from another class with the default constructor, or a class containing another class object with default constructor. The compiler needs to insert code to call the default constructors of base class/embedded object.
- There are many scenarios in which compiler needs to insert code to ensure some necessary initialization as per language requirements.

2. Parameterized Constructor:

- It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created.
- When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function.
- The constructors can be called explicitly or implicitly.

```
Class Object = Class(arg1, arg2);   // Explicit call
Class(arg1, arg2);                  // Implicit call
```

- 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.

3. Copy Constructor:

- A copy constructor is a member function which initializes an object using another object of the same class.
- **Syntax:**

```
ClassName (const ClassName &old_obj):
```

- **Example:**
  - *Input:*

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

class String
{
   private:
       char *s;
       int size;
   public:
       String(const char *str = NULL);      // Constructor
       ~String() { delete [] s;  }          // Destructor
       String(const String&);               // Copy constructor
       void print() { cout << s << endl; }  // Function to print string
       void change(const char *);           // Function to change
};

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

void String::change(const char *str)
{
   delete [] s;
   size = strlen(str);
   s = new char[size+1];
   strcpy(s, str);
}

String::String(const String& old_str)
{
   size = old_str.size;
   s = new char[size+1];
   strcpy(s, old_str.s);
}
```

```
int main()
{
   String str1("Shivam");
   String str2 = str1;

   str1.print();
   str2.print();

   str2.change("Kumar");

   str1.print();
   str2.print();
   return 0;
}
```

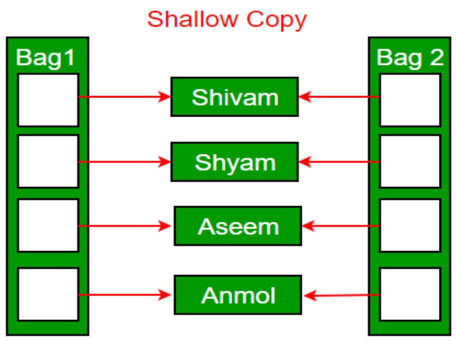
- *Output:*

```
Shivam
Shivam
Shivam
ShivamKumar
```

- Copy constructor can be called in the following ways:
  - Class copyObject(oldObject);
  - Class copyObject = oldObject;
- **Copy constructor vs Assignment Operator:**

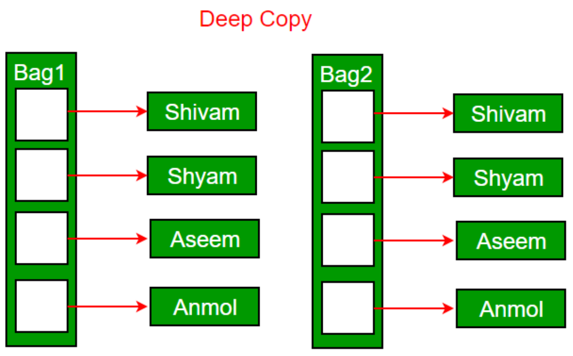
```
MyClass t1, t2;
MyClass t3 = t1;   // Copy Constructor
t2 = t1;           // Assignment Operator
```

  - Copy constructor is called when a new object is created from an existing object, as a copy of the existing object.
  - Assignment operator is called when an already initialized object is assigned a new value from another existing object.
- **There are two types of copy constructor:-**
  - **i. Default Copy Constructor / Shallow Copy:**
    - 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 runtime allocation of the resource like file handle, a network connection etc.
    - The default constructor creates the exact copy or shallow copy of the existing object. Thus, the pointer of both the objects point to the same memory location.
    - Therefore, when the memory of a field is freed, the memory of another field is also automatically freed as both the fields point to the same memory location.



ii. User Defined Copy Constructor / Deep Copy:

- In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.
- Deep copy dynamically allocates the memory for the copy and then copies the actual value, both the source and copy have distinct memory locations.
- In this way, both the source and copy are distinct and will not share the same memory location. Deep copy requires us to write the user-defined constructor.
- Deep copy does not create the copy of a reference type variable.



➢ Can a constructor be private?

- By default, constructors are defined in public section of class and Yes, Constructor can be defined in private section of class.
- If constructor defined in private section of class, then we can not make its object directly and any other class can not inherit it. To be Inherited the constructor should be defined in protected.
- Ways to use Constructors in private section are:
  - **Using Friend Class :** If we want that class should not be instantiated by anyone else but only by a friend class.
  - **Using Singleton design pattern:** When we want to design a singleton class. This means instead of creating several objects of class, the system is driven by a single object or a very limited number of objects.
  - **Named Constructor Idiom :** Since constructor has same name as of class, different constructors are differentiated by their parameter list, but if numbers of constructors are more, then implementation can become error prone. With the

Named Constructor Idiom, we can declare all the class's constructors in the private or protected sections, and then for accessing objects of class, we can create public static functions.

Does compiler create default constructor when we write our own?

- In C++, compiler by default creates default constructor for every class. But, if we define our own constructor, compiler doesn't create the default constructor.
- This is so because default constructor does not take any argument and if two default constructors are created, it is difficult for the compiler which default constructor should be called.

Can we make copy constructor private?

- Yes, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable.
- This is particularly useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our own copy constructor like above String example or make a private copy constructor so that users get compiler errors rather than surprises at runtime.

Why argument to a copy constructor must be passed as a reference?

- A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So, if we pass an argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore, compiler doesn't allow parameters to be passed by value.
- In other words, If an object is passed as value to the Copy Constructor then its copy constructor would call itself, to copy the actual parameter to the formal parameter. Thus, an endless chain of call to the copy constructor will be initiated. This process would go on until the system run out of memory.

Why copy constructor argument should be const?

- When we create our own copy constructor, we pass an object by reference and we generally pass it as a const reference.
- It ensures that we don't accidentally damage or modify the original when making the copy, this is a good thing, because we don't really want our original object to be changed when making a copy of it.
- Copy constructors are sometimes used with arguments that are *temporaries* (Example: return from function). And non-const references to temporaries don't fly. Sometimes compiler created temporary objects cannot be bound to non-const references and the original program tries to do that. It doesn't make sense to modify compiler created temporary objects as they can die any moment.

➢ When should we write our own copy constructor?

- C++ compilers provide default copy constructor (and assignment operator) with class. When we don't provide implementation of copy constructor (and assignment operator) and tries to initialize object with already initialized object of same class then copy constructor gets called and copies members of class one by one in target object.
- The problem with default copy constructor (and assignment operator) is when we have members which dynamically gets initialized at run time, default copy constructor copies this member with address of dynamically allocated memory and not real copy of this memory. Now both the objects point to the same memory and changes in one reflects in another object, Further the main disastrous effect is, when we delete one of this object other object still points to same memory, which will be dangling pointer, and memory leak is also possible problem with this approach. Hence, in such cases, we should always write our own copy constructor (and assignment operator).

➢ When is copy constructor called?

- In C++, a Copy Constructor may be called in following cases:
  - When an object of the class is returned by value.
  - When an object of the class is passed (to a function) by value as an argument.
  - When an object is constructed based on another object of the same class.
  - When compiler generates a temporary object.
- It is however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases

➢ What happens when we write only a copy constructor – does compiler create default constructor?

- Compiler doesn't create a default constructor if we write any constructor even if it is copy constructor.

Destructor

- Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.
- The thing to be noted is that a destructor doesn't destroy an object.
- **Properties of Destructor:**
  - Destructor function is automatically invoked when the objects are destroyed.
  - It cannot be declared static or const.
  - The destructor does not have arguments.
  - It has no return type not even void.
  - An object of a class with a Destructor cannot become a member of the union.
  - A destructor should be declared in the public section of the class.
  - The programmer cannot access the address of destructor.

- **When is destructor called?**
  - A destructor function is called automatically when the object goes out of scope:
    - the function ends
    - the program ends
    - a block containing local variables ends
    - a delete operator is called

- **Can there be more than one destructor in a class?**
  - No, there can only one destructor in a class with class-name 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, and it is always a good idea to make destructors virtual in base class when we have a virtual function.

- Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.
- Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.
- Note: Any time we have a virtual function in a class, we should immediately add a virtual destructor (even if it does nothing).

- **Can a destructor be private in C++ ?**
  - The compiler notices that the local variable ‘objectName’ cannot be destructed because the destructor is private, it will give compile time error.
    ```
    className objectName;
    ```
  - When something is created using dynamic memory allocation, it is programmer’s responsibility to delete it. So, compiler doesn’t bother. However, the below program fails in compilation. When we call delete, destructor is called.
    ```
    className* objectName = new className;
    ```

- **What is the use of private destructor?**
  - Whenever we want to control destruction of objects of a class, we make the destructor private.
  - For dynamically created objects, it may happen that we pass a pointer to the object to a function and the function deletes the object. If the object is referred after the function call, the reference will become dangling.

- **Is it possible to call constructor and destructor explicitly?**
  - Yes, it is possible to call special member functions explicitly by programmer.
  - **Example:**

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

    class Test
    {
        public:
            Test()  { cout << "Constructor is executed\n"; }
             ~Test() { cout << "Destructor is executed\n";  }
    };

    int main()
    {
    ```

```
    Test();         // Explicit call to constructor
    Test t;         // Local object
    t.~Test();   // Explicit call to destructor
    return 0;
}
```

Output:

```
Constructor is executed
Destructor is executed
Constructor is executed
Destructor is executed
Destructor is executed
```

- When the constructor is called explicitly the compiler creates a nameless temporary object and it is immediately destroyed. That's why 2nd line in the output is call to destructor.

❖ Initialization of data members:

- In C++, class variables are initialized in the same order as they appear in the class declaration.
- If they are not initialized in same order, then garbage value is obtained.

```
class Test
{
   private:
       int x;
       int y;
   public:
       Test() : x(10), y(x + 10) {}
};
```

❖ When are static objects destroyed?

- **What is static keyword in C++?**
    - static keyword can be applied to local variables, functions, class' data members and objects in C++.
    - static local variable retains their values between function call and initialized only once.
    - static function can be directly called using the scope resolution operator preceded by class name.

- **What are static objects in C++?**
    - An object become static when static keyword is used in its declaration.
    - Example:

    ```
    class obj1;             // Stack based object
    static class obj2;     // Static object
    ```

    - First statement when executes creates object on stack means storage is allocated on stack. Stack based objects are also called automatic objects or local objects.
    - static object are initialized only once and live until the program terminates. static objects are allocated storage in static storage area. static object is destroyed at the termination of program.
    - **Example:**

    Input:

    ```
    #include <iostream>
    class Test
    {
       public:
           Test()
           {
              std::cout << "Constructor is executed\n";
           }
    ```

```
    ~Test()
    {
        std::cout << "Destructor is executed\n";
    }
};

void myfunc()
{
    static Test obj;
}   // Object obj is still not destroyed because it is static

int main()
{
    std::cout << "main() starts\n";
    myfunc();       // Destructor will not be called here
    std::cout << "main() terminates\n";
    return 0;
}
```

Output:

```
main() starts
Constructor is executed
main() terminates
Destructor is executed
```

Explanation: If we observe the output of this program closely, we can see that the destructor for the local object named obj is not called after it's constructor is executed because the local object is static so it has scope till the lifetime of program so it's destructor will be called when main() terminates.

❖ When do we use Initializer List in C++?

- Initializer List is used in initializing the data members of a class.
- The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon.
- **Example:**
  Input:

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

class Point
{
  private:
     int x;
     int y;
```

```
    public:
        Point(int i = 0, int j = 0):x(i), y(j) {}
    /*  The above use of Initializer list is optional as the
        constructor can also be written as:
        Point(int i = 0, int j = 0)
       {
            x = i;
            y = j;
        }
    */

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

int main()
{
    Point t1(10, 15);
    cout << "x = " << t1.getX() << ", ";
    cout << "y = " << t1.getY();
    return 0;
}
```

Output:

```
x = 10, y = 15
```

- The above code is just an example for syntax of the Initializer list. In the above code, x and y can also be easily initialed inside the constructor.
- But there are situations where initialization of data members inside constructor doesn't work and Initializer List must be used.
- Following are such cases:
  - For initialization of non-static const data members.
  - For initialization of reference members.
  - For initialization of member objects which do not have default constructor.
  - For initialization of base class members.
  - When constructor's parameter name is same as data member.