University American College Skopje

SINCE 2005

OBJECT PROGRAMMING

- LECTURE 4 -

(1ST YEAR OF STUDY)

Skopje, 2023/24

Contents

4.1. Introduction

- A **constructor** is a special member function of a class which initializes objects of a class.
- In C++, constructor is automatically called when object of a class is created.
- By default, constructors are defined in *public* section of class.
- So, question is can a constructor be defined in *private* section of class?
- Answer: Yes, Constructor can be defined in *private* section of class.

4.1. Introduction

- How to use Constructors in private section?

  1. Using Friend Class: If we want that class should not be instantiated by anyone else but only by a friend class.

  - If you comment the line **friend class B,** you will encounter an error.

```cpp
// CPP program to demonstrate usage of
// private constructor
#include <iostream>
using namespace std;

// class A
class A{
private:
    A(){
       cout << "constructor of A\n";
    }
    friend class B;
};

// class B, friend of class A
class B{
public:
    B(){
        A a1;
        cout << "constructor of B\n";
    }
};

// Driver program
int main(){
    B b1;
    return 0;
}
```

Output:

```
constructor of A
constructor of B
```

4.1. Introduction

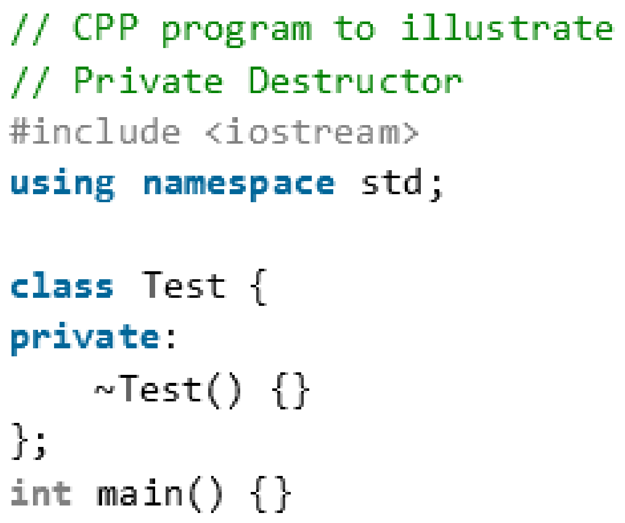
2. Using Singleton design pattern: When we want to design a singleton class. This means instead of creating several objects of class, the system is driven by a single object, or a very limited number of objects.

3. Named Constructor Idiom: Since constructor has same name as of class, different constructors are differentiated by their parameter list, but if numbers of constructors is more, then implementation can become error prone. With the *Named Constructor Idiom*, you declare all the class's constructors in the *private* or *protected* sections, and then for accessing objects of class, you create *public* static functions.

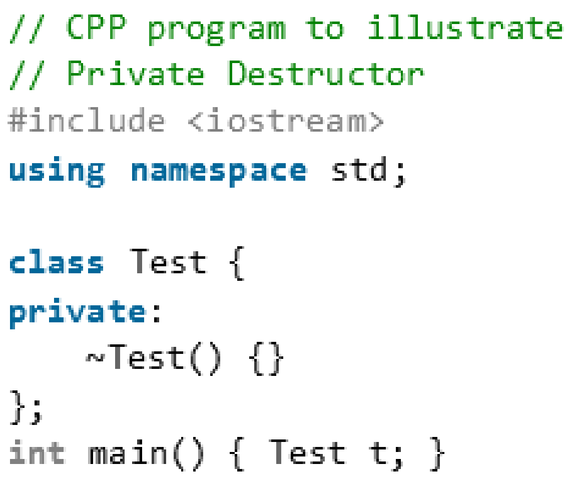
4.1. Introduction

- Destructors with the ***access modifier*** as *private* are known as **Private Destructors.**
- Whenever we want to prevent the destruction of an object, we can make the destructor private.
- For dynamically created objects, it may happen that you 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.

4.1. Introduction



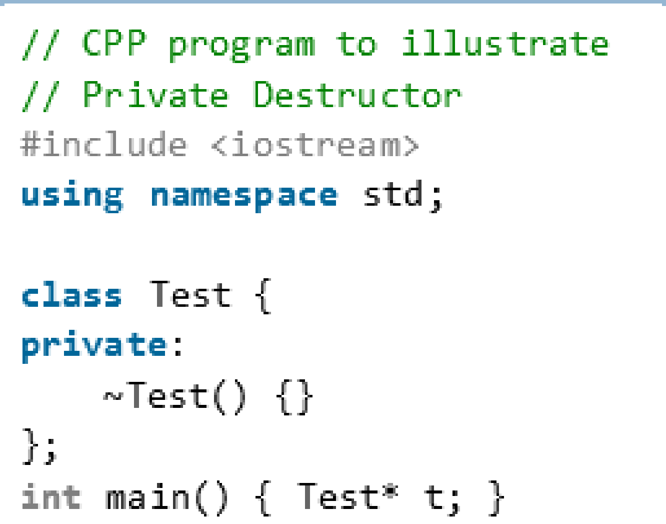
```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};
int main() {}
```



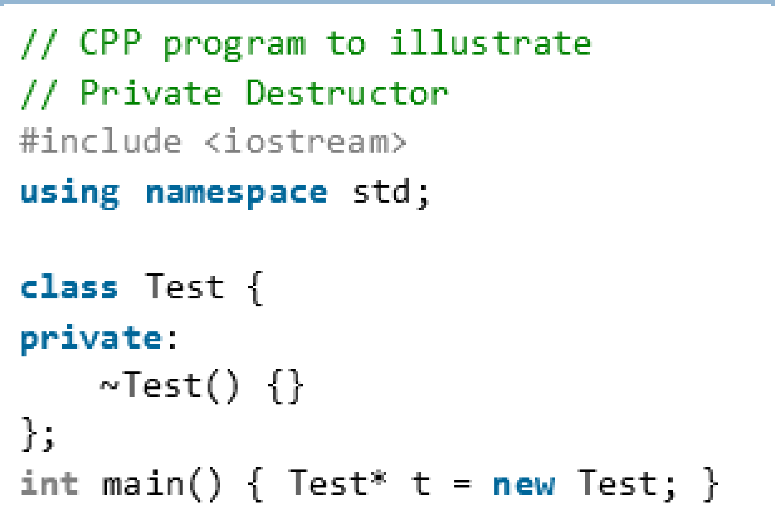
```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};
int main() { Test t; }
```

- The 1st program compiles and runs fine. Hence, we can say that: It is **not** a compiler error to create private destructors.
- The 2nd program fails in the compilation. The compiler notices that the local variable ‘t’ cannot be destructed because the destructor is private.

4.1. Introduction



```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};
int main() { Test* t; }
```

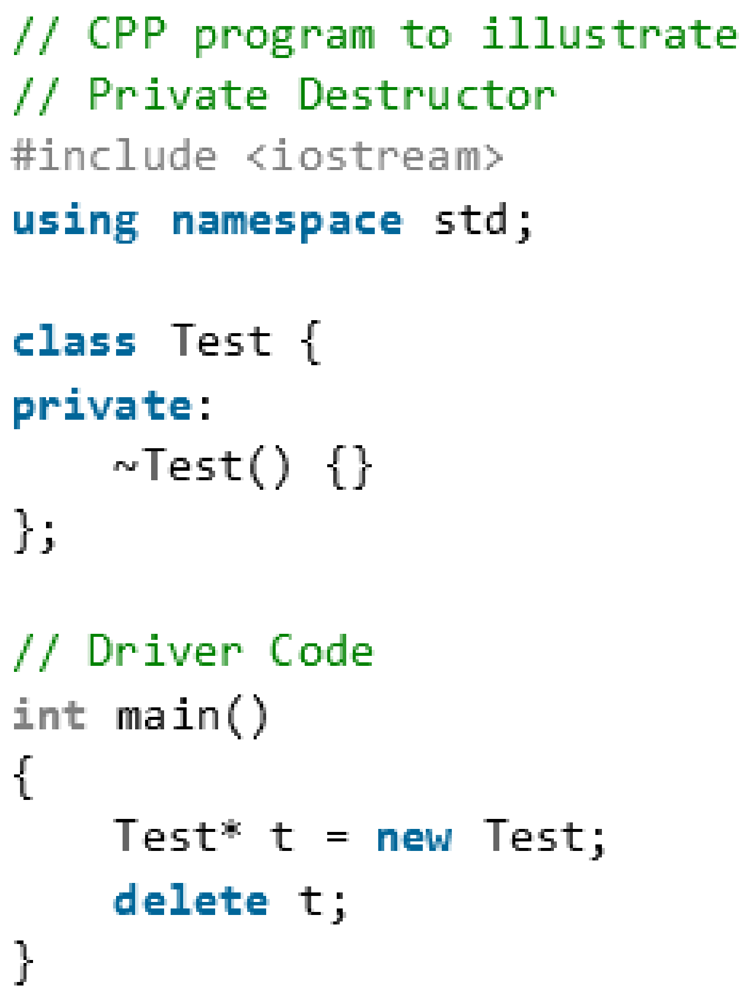


```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};
int main() { Test* t = new Test; }
```

- The 1st program works fine. There is no object being constructed, the program just creates a pointer of type “Test*”, so nothing is destructed.
- The 2nd program also works fine. When something is created using dynamic memory allocation, it is the programmer’s responsibility to delete it. So compiler doesn’t bother.

4.1. Introduction

- However, the below program *fails* in the compilation.
- When we call delete, destructor is called.
- We noticed in the above programs when a class has a private destructor, only dynamic objects of that class can be created.



```cpp
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};

// Driver Code
int main()
{
    Test* t = new Test;
    delete t;
}
```

4.1. Introduction

- A ***copy constructor*** is a member function that initializes an object using another object of the same class. The copy constructor is called mainly when a new object is created from an existing object, as a copy of the existing object.
- In C++, a *copy constructor* may be called for the cases:

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.

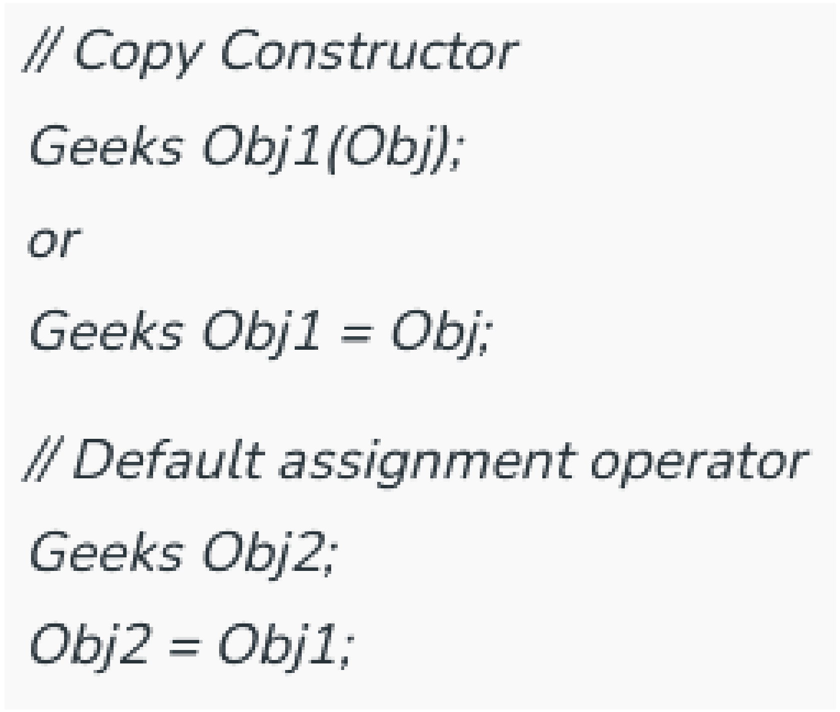
4) When the compiler generates a temporary object.

4.1. Introduction

- 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, one example being the ***Return Value Optimization*** (sometimes referred to as ***RVO***).
- **Note:** C++ compiler implicitly provides a copy constructor, if no copy constructor is defined in the class.

4.2. Shallow Copy and Deep Copy in C++

- In general, creating a copy of an object means to create an exact replica of the object having the same *literal value*, *data type*, and *resources*.



```
// Copy Constructor
Geeks Obj1(Obj);
or
Geeks Obj1 = Obj;

// Default assignment operator
Geeks Obj2;
Obj2 = Obj1;
```

4.2. Shallow Copy and Deep Copy in C++

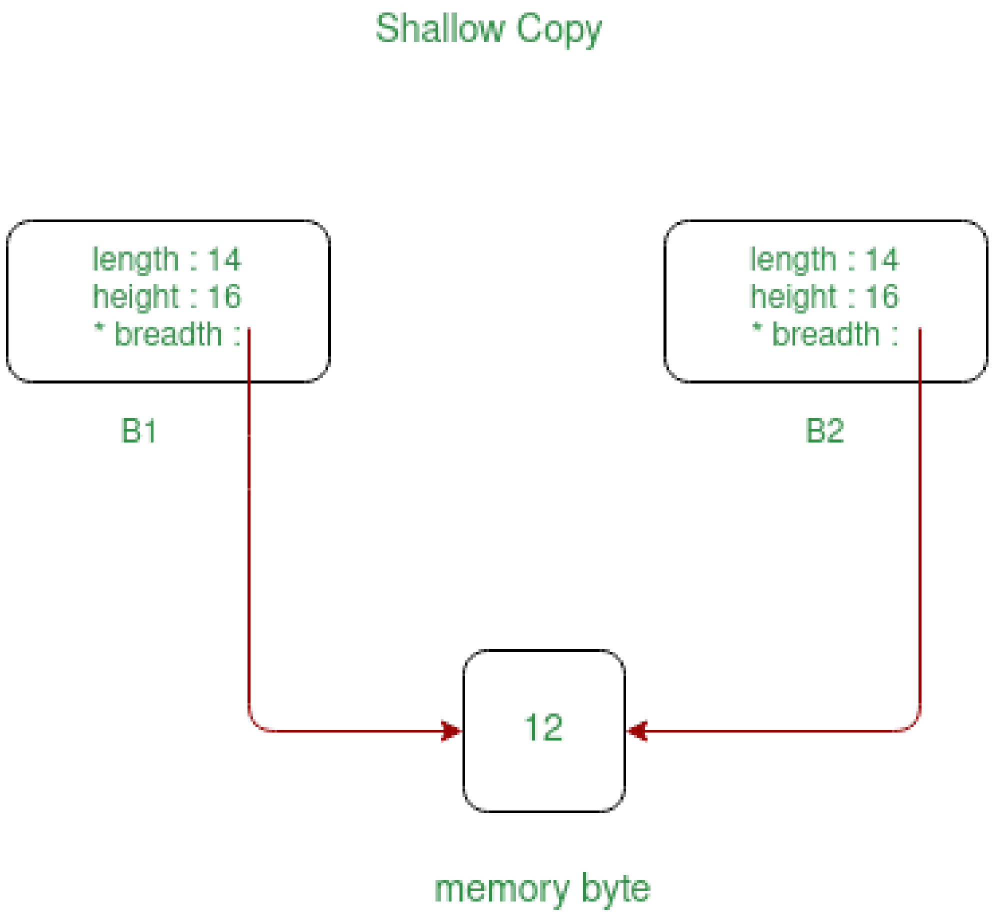
- In **shallow copy**, an object is created by simply copying the data of all variables of the original object.
- This works well if none of the variables of the object are defined in the *heap section of memory*. If some variables are dynamically allocated memory from heap section, then the copied object variable will also reference the same memory location.

4.2. Shallow Copy and Deep Copy in C++

- This will create ambiguity and run-time errors, dangling pointer. Since both objects will reference to the same memory location, then change made by one will reflect those change in another object as well. Since we wanted to create a replica of the object, this purpose will not be filled by Shallow copy.
- **Note:** C++ compiler implicitly creates a ***copy constructor*** and ***overloads assignment operator*** in order to perform shallow copy at compile time.

4.2. Shallow Copy and Deep Copy in C++

- **Shallow Copy** of object if some variables are defined in heap memory, then:



4.2. Shallow Copy and Deep Copy in C++

```cpp
// C++ program for the above approach
#include <iostream>
using namespace std;

// Box Class
class box {
private:
    int length;
    int breadth;
    int height;

public:
    // Function that sets the dimensions
    void set_dimensions(int length1, int breadth1,
                        int height1)
    {
        length = length1;
        breadth = breadth1;
        height = height1;
    }

    // Function to display the dimensions
    // of the Box object
    void show_data()
    {
        cout << " Length = " << length
             << "\n Breadth = " << breadth
             << "\n Height = " << height
    }        << endl;
};
```

```cpp
// Driver Code
int main()
{
    // Object of class Box
    box B1, B3;

    // Set dimensions of Box B1
    B1.set_dimensions(14, 12, 16);
    B1.show_data();

    // When copying the data of object
    // at the time of initialization
    // then copy is made through
    // COPY CONSTRUCTOR
    box B2 = B1;
    B2.show_data();

    // When copying the data of object
    // after initialization then the
    // copy is done through DEFAULT
    // ASSIGNMENT OPERATOR
    B3 = B1;
    B3.show_data();

    return 0;
}
```

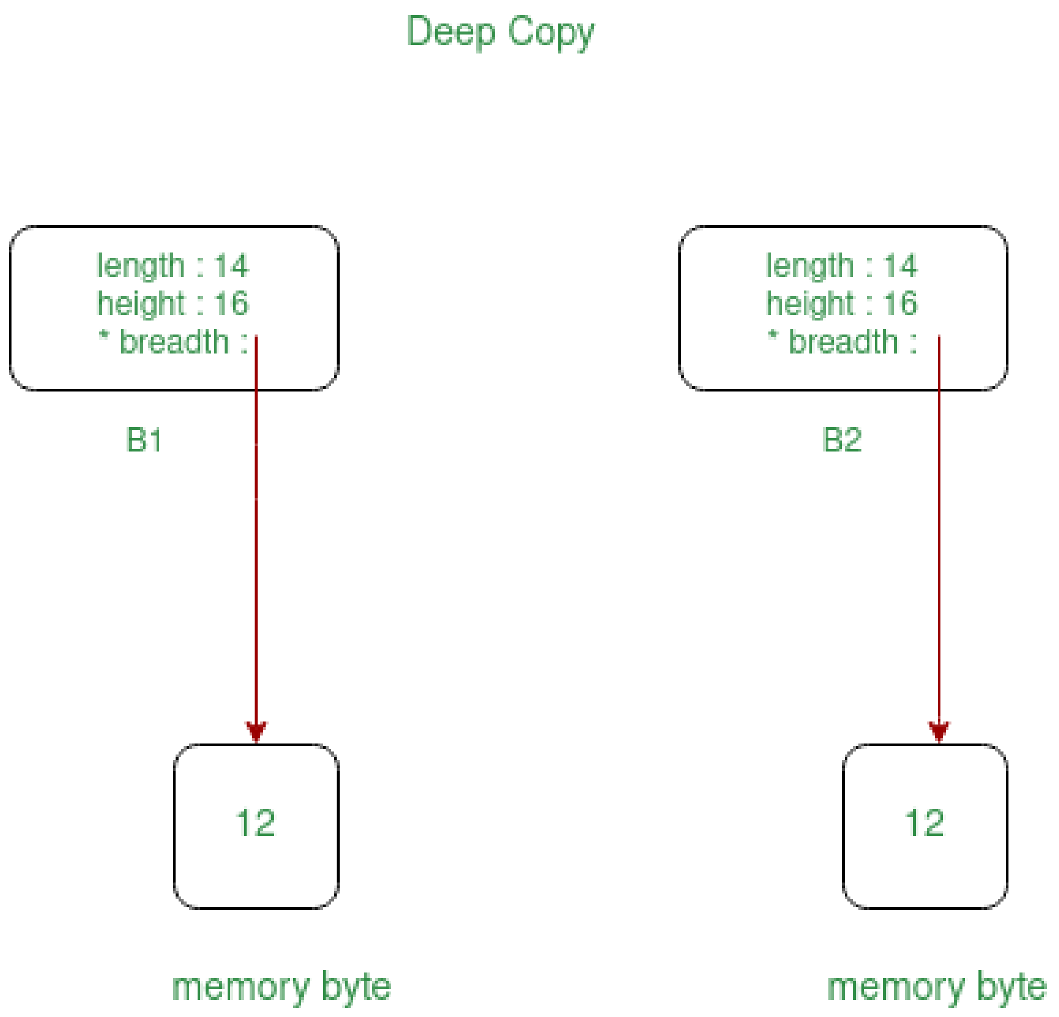
Output:

```
 Length = 14
  Breadth = 12
  Height = 16
  Length = 14
  Breadth = 12
  Height = 16
  Length = 14
  Breadth = 12
  Height = 16
```

4.2. Shallow Copy and Deep Copy in C++

- In **deep copy**, an object is created by copying data of all variables, and it also allocates similar memory resources with the same value to the object.
- In order to perform Deep copy, we need to explicitly define the copy constructor and assign dynamic memory as well, if required.
- Also, it is required to dynamically allocate memory to the variables in the other constructors, as well.

4.2. Shallow Copy and Deep Copy in C++



4.2. Shallow Copy and Deep Copy in C++

```cpp
// C++ program to implement the
// deep copy
#include <iostream>
using namespace std;

// Box Class
class box {
private:
    int length;
    int* breadth;
    int height;

public:
    // Constructor
    box()
    {
        breadth = new int;
    }

    // Function to set the dimensions
    // of the Box
    void set_dimension(int len, int brea,
                       int heig)
    {
        length = len;
        *breadth = brea;
        height = heig;
    }
```

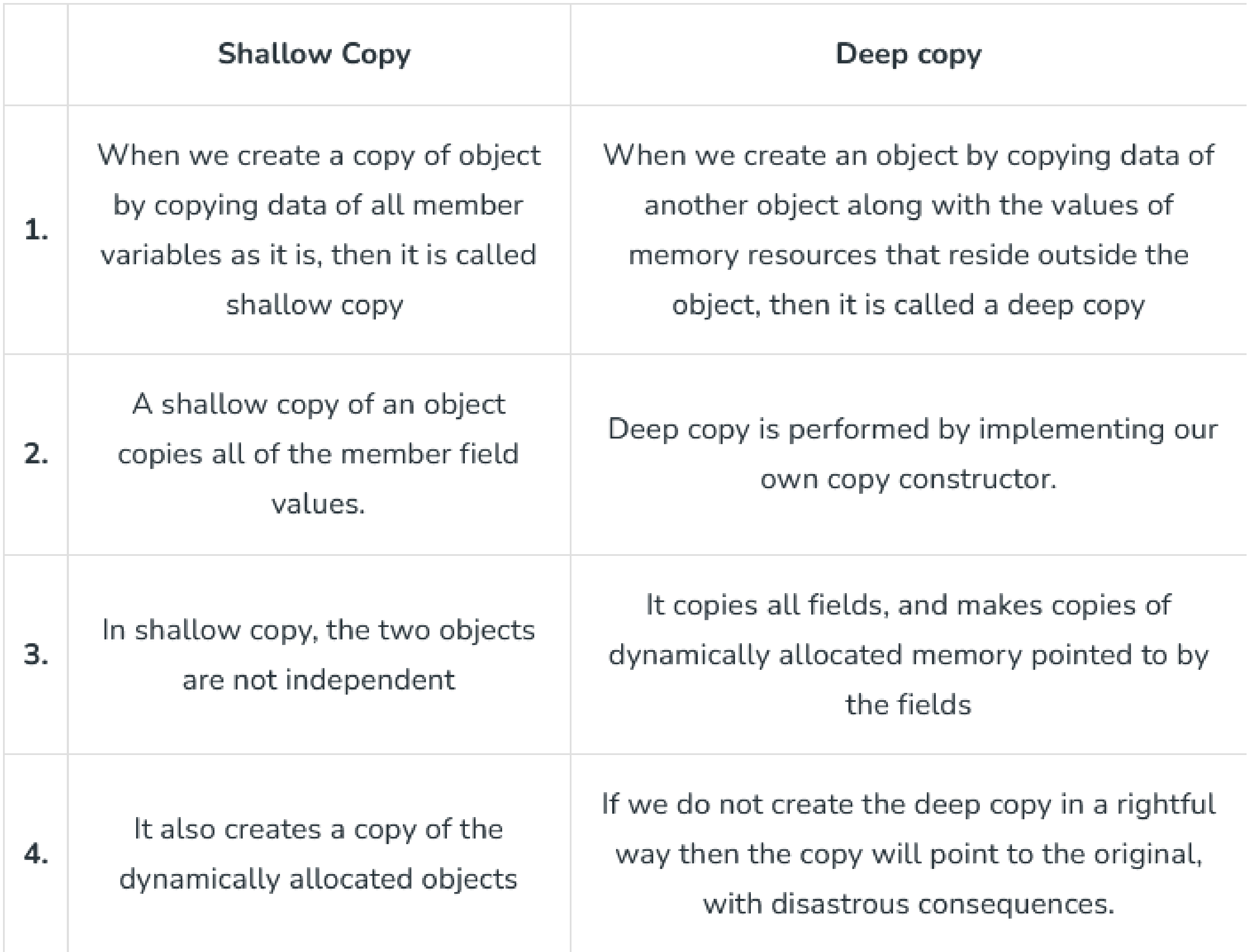
```cpp
// Parameterized Constructors for
// for implementing deep copy
box(box& sample)
{
    length = sample.length;
    breadth = new int;
    *breadth = *(sample.breadth);
    height = sample.height;
}

// Destructors
~box()
{
    delete breadth;
}
};
```

Output:

```
Length = 12
 Breadth = 14
 Height = 16
 Length = 12
 Breadth = 14
 Height = 16
```

```cpp
// Driver Code
int main()
{
    // Object of class first
    box first;

    // Set the dimensions
    first.set_dimension(12, 14, 16);

    // Display the dimensions
    first.show_data();

    // When the data will be copied then
    // all the resources will also get
    // allocated to the new object
    box second = first;

    // Display the dimensions
    second.show_data();

    return 0;
}
```

4.2. Shallow Copy and Deep Copy in C++



	Shallow Copy	Deep copy
1.	When we create a copy of object by copying data of all member variables as it is, then it is called shallow copy	When we create an object by copying data of another object along with the values of memory resources that reside outside the object, then it is called a deep copy
2.	A shallow copy of an object copies all of the member field values.	Deep copy is performed by implementing our own copy constructor.
3.	In shallow copy, the two objects are not independent	It copies all fields, and makes copies of dynamically allocated memory pointed to by the fields
4.	It also creates a copy of the dynamically allocated objects	If we do not create the deep copy in a rightful way then the copy will point to the original, with disastrous consequences.

4.2. Shallow Copy and Deep Copy in C++

- A copy constructor is a member function that initializes an object using another object of the same class.
- C++ compiler provides a default copy constructor (and assignment operator) with class. When we don’t provide an implementation of the copy constructor (and assignment operator) and try to initialize an object with the already initialized object of the same class, then the copy constructor gets called and copies members of the class one by one in the target object.

4.2. Shallow Copy and Deep Copy in C++

- But the problem with the **default copy constructor** (and **assignment operator**) is:
  - When we have members that dynamically get initialized at run time, the default copy constructor copies this member with the address of dynamically allocated memory, and not a real copy of this memory.
  - Now, both the objects point to the same memory, and changes in one reflect in another object.
  - Further, the main disastrous effect is, that when we delete one of these objects another object still points to the same memory, which will be a dangling pointer, and memory leak is also a possible problem with this approach.

4.2. Shallow Copy and Deep Copy in C++

- So, we need to define our own copy constructor only if an object has pointers, or any runtime allocation of the resource like: a file handle, a network connection, etc.
- Another use case of a copy constructor is when we want to copy only specific data members of the class from one object to another object, and keep the remaining data members identical for each object, we can write our own copy constructor.

4.2. Shallow Copy and Deep Copy in C++

- The *default constructor* does only *shallow* copy.
- *Deep copy* is possible only with a *user-defined copy constructor*.
- In a user-defined copy constructor, we make sure that pointers (or references) of copied objects point to new memory locations.
- Hence, in such cases, we should always write our own copy constructor (and assignment operator).

4.2. Shallow Copy and Deep Copy in C++

- The C++ compiler doesn't create a default constructor when we initialize our own.
- The compiler by default creates a default constructor for every class; but, if we define our own constructor, the compiler doesn't create the default constructor.
- This is so because the 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.

4.3. Static Members in C++

- **Static data members** are class members that are declared using **static** keyword. A static member has certain special characteristics which are as follows:
  - Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
  - It is initialized before any object of this class is created, even before the main starts.
  - It is visible only within the class, but its lifetime is the entire program.

4.3. Static Members in C++

Syntax:

```
static data_type data_member_name;
```

```
// C++ Program to demonstrate
// the working of static data member
#include <iostream>
using namespace std;

class A {
public:
    A()
    {
      cout << "A's Constructor Called " <<
               endl;
    }
};
```

Output

```
B's Constructor Called
```

```
class B {
    static A a;

public:
    B()
    {
      cout << "B's Constructor Called " <<
               endl;
    }
};

// Driver code
int main()
{
    B b;
    return 0;
}
```

4.3. Static Members in C++

- **Explanation:** The above program calls only B’s constructor, it doesn’t call A’s constructor.
- The reason is that Static members are only declared in a class declaration, not defined. They must be explicitly defined outside the class using the scope resolution operator.
- Below is the C++ program to show when a static member ‘a’ is accessed without explicit definition:

4.3. Static Members in C++

```
// C++ Program to demonstrate
// the Compilation Error occurred
// due to violation of Static
// Data Memeber Rule
#include <iostream>
using namespace std;

class A {
    int x;

public:
    A()
    {
      cout << "A's constructor called " <<
                endl;
    }
};
```

Output

```
Compiler Error: undefined reference to `B::a'
```

```
class B {
    static A a;

public:
    B()
    {
      cout << "B's constructor called " <<
                endl;
    }
    static A getA()
    {
      return a;
    }
};

// Driver code
int main()
{
    B b;
    A a = b.getA();
    return 0;
}
```

4.3. Static Members in C++

- **Explanation:** Here static member ‘a’ is accessed without explicit definition. If we add the definition, the program will work fine and call A’s constructor.
- To access the static data member of any class we have to define it first.
- Below is the C++ program to show how to resolve the above error:

4.3. Static Members in C++

```cpp
// C++ program to access static data
// member with explicit definition
#include <iostream>
using namespace std;

class A {
    int x;

public:
    A()
    {
      cout << "A's constructor called " <<
               endl;
    }
};
```

```cpp
class B {
    static A a;

public:
    B()
    {
      cout << "B's constructor called " <<
               endl;
    }
    static A getA()
    {
      return a;
    }
};

// Definition of a
A B::a;

// Driver code
int main()
{
  B b1, b2, b3;
  A a = b1.getA();

  return 0;
}
```

Output

```
A's constructor called
B's constructor called
B's constructor called
B's constructor called
```

4.3. Static Members in C++

- **Explanation:** The above program calls B’s constructor 3 times for 3 objects (b1, b2, and b3), but calls A’s constructor only once.
- The reason is that the *static members* are shared among all objects. That is why they are also known as *class members* or *class fields*. Also, static members can be accessed without any object.
  - **NOTE**: Static data members can only be defined globally in C++. The only exception to this are static const data members of integral type which can be initialized in the class declaration.

4.3. Static Members in C++

- We can access *any static member* without any object by using the *scope resolution operator* directly with the *class name.*

```cpp
// C++ Program to demonstrate
// static members can be accessed
// without any object
#include <iostream>
using namespace std;

class A {
    int x;

public:
    A()
    {
      cout << "A's constructor called " <<
                endl;
    }
};
```

```cpp
class B {
    static A a;

public:
    B()
    {
      cout << "B's constructor called " <<
                endl;
    }
    static A getA()
    {
      return a;
    }
};

// Definition of a
A B::a;
```

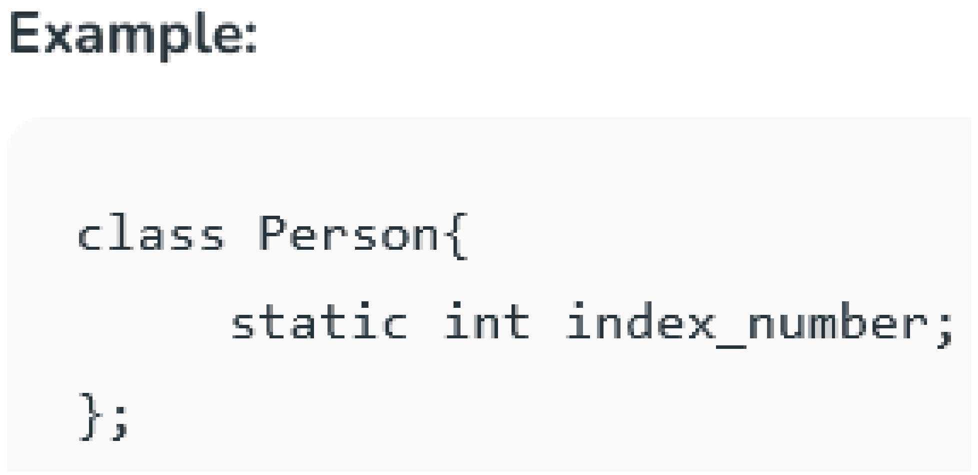
```cpp
// Driver code
int main()
{
  // static member 'a' is accessed
  // without any object of B
  A a = B::getA();

  return 0;
}
```

Output

```
A's constructor called
```

4.3. Static Members in C++

- **Static members** of a class are not associated with the objects of the class.
- Just like a static variable once declared is allocated with memory that can’t be changed, every object points to the same memory.



Example:

```
class Person{
     static int index_number;
};
```

4.3. Static Members in C++

- Once a static member is declared, it will be treated as same for all the objects associated with the class.

```cpp
// C++ Program to demonstrate
// Static member in a class
#include <iostream>
using namespace std;

class Student {
public:
    // static member
    static int total;

    // Constructor called
    Student() { total += 1; }
};

int Student::total = 0;
```

```cpp
int main()
{
    // Student 1 declared
    Student s1;
    cout << "Number of students:" << s1.total << endl;

    // Student 2 declared
    Student s2;
    cout << "Number of students:" << s2.total << endl;

    // Student 3 declared
    Student s3;
    cout << "Number of students:" << s3.total << endl;
    return 0;
}
```

Output

```
Number of students:1
Number of students:2
Number of students:3
```

4.3. Static Members in C++

- **Static Member Function** in a class is the function that is declared as static because of which function attains certain properties as defined below:
  - A static member function is independent of any class object.
  - A static member function can be called even if no objects exist.
  - A static member function can also be accessed using the class name through the scope resolution operator.
  - A static member function can access static data members and static member functions inside or outside of the class.
  - Static member functions have a scope inside the class and cannot access the current object pointer.
  - You can also use a static member function to determine how many objects of the class have been created.

4.3. Static Members in C++

- **The reason we need Static member function:**
  - Static members are frequently used to store information that is shared by all objects in a class.
  - For instance, you may keep track of the quantity of newly generated objects of a specific class type using a static data member as a counter. This static data member can be increased each time an object is generated to keep track of the overall number of objects.

4.3. Static Members in C++

```cpp
// C++ Program to show the working of
// static member functions
#include <iostream>
using namespace std;

class Box
{
    private:
    static int length;
    static int breadth;
    static int height;

    public:

    static void print()
    {
        cout << "The value of the length is: " << length << endl;
        cout << "The value of the breadth is: " << breadth << endl;
        cout << "The value of the height is: " << height << endl;
    }
};

// initialize the static data members

int Box :: length = 10;
int Box :: breadth = 20;
int Box :: height = 30;
```

```cpp
// Driver Code

int main()
{

    Box b;

    cout << "Static member function is called through Object name: \n" << endl;
    b.print();

    cout << "\nStatic member function is called through Class name: \n" << endl;
    Box::print();

    return 0;
}
```

Output

```
Static member function is called through Object name:

The value of the length is: 10
The value of the breadth is: 20
The value of the height is: 30

Static member function is called through Class name:

The value of the length is: 10
The value of the breadth is: 20
The value of the height is: 30
```