**https://owlcation.com/stem/Copy-Constructor-shallow-copy-vs-deep-copy**

**1. Introduction**

When we pass standard data types (int, float, bool etc.,) to a function a copy from the calling piece of code to called function occurs. Now look at the below piece of code which performs a simple function call:

[?](https://owlcation.com/stem/Copy-Constructor-shallow-copy-vs-deep-copy)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int AddNumbers(int loc\_X, int loc\_Y)  {      return (loc\_X + loc\_Y);  }    void main  {      int x = 5;      int y = 3;      int result = AddNumbers(x, y);  } |

The copy I am taking occurs between x=>loc\_X and y=>loc\_Y. The content of variable x in main function scope is copied to the variable loc\_X, which is in the ***AddNumbers*** function scope. This holds true for the next parameter loc\_Y also. This copying is shown below:

Source

OK. This is good for standard data types. A class can have one or more data members. How the copy occurs between the data members is what we are going to deal with this hub. When the Hub progresses, I will explain ***Shallow Copy***, ***Deep Copy*** and the need for our own ***copy constructor***.

**2. ShalloC class**

To demonstrate the need for the copy constructor, we will first define an example class. This example class is ***ShalloC***. This class contains only one integer pointer as private data member as shown below:

[?](https://owlcation.com/stem/Copy-Constructor-shallow-copy-vs-deep-copy)

|  |  |
| --- | --- |
| 1  2  3 | //Sample 01: Private Data Member  private:      int \* x; |

The constructor will create a memory location in a heap and copy the passed in value m to the heap content. This code is shown below:

[?](https://owlcation.com/stem/Copy-Constructor-shallow-copy-vs-deep-copy)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //Sample 02: Constructor with single parameter  ShalloC(int m)  {      x = new int;      \*x = m;  } |

The Get and Set functions are used to get the heap memory content value and Set the heap memory content respectively. Below is the code that sets and gets the integer heap memory value:

[?](https://owlcation.com/stem/Copy-Constructor-shallow-copy-vs-deep-copy)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //Sample 03: Get and Set Functions  int GetX() const  {      return \*x;  }  void SetX(int m)  {      \*x = m;  } |

Finally, there is a function to print the heap content value in the console window. The function is shown below:

[?](https://owlcation.com/stem/Copy-Constructor-shallow-copy-vs-deep-copy)

|  |  |
| --- | --- |
| 1  2  3  4  5 | //Sample 04: Print Function  void PrintX()  {      cout << "Int X=" << \*x << endl;  } |

Now you may get the idea of what the ***ShalloC*** class will do. At present it has a constructor that creates a heap memory and in the destructor we clear the memory created as shown in the below code:

[?](https://owlcation.com/stem/Copy-Constructor-shallow-copy-vs-deep-copy)

|  |  |
| --- | --- |
| 1  2  3  4  5 | //Sample 05: DeAllocate the heap  ~ShalloC()  {      delete x;  } |

**3. Shallow Copy vs. Deep Copy**

In the Program main we created two Objects ob1 and ob2. The object ob2 is created using the copy constructor. How? And where is the "copy constructor".? If you look at the statement ***ShalloC ob2 = ob1 ;*** you clearly know that the ob2 is not yet created and in the mean time ob1 is already created. Hence, a copy constructor is invoked. Even though the copy constructor not implemented, the compiler will provide default copy constructor. Once both the objects are created we print the values in ob1 and ob2.

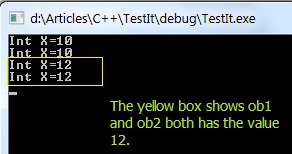
[?](https://owlcation.com/stem/Copy-Constructor-shallow-copy-vs-deep-copy)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //Sample 06: Create Object 1 and copy that to Object 2.  //           Print the data member for both Object 1 & 2.  ShalloC ob1(10);  ShalloC ob2 = ob1 ;  ob1.PrintX();  ob2.PrintX(); |

After printing the values in ob1 and ob2 we change the value of the object ob1’s data member pointed value to 12. Then both the values of ob1 and ob2 are printed. The code and its output are shown below:

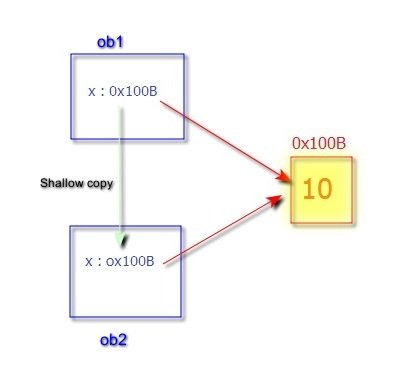
[?](https://owlcation.com/stem/Copy-Constructor-shallow-copy-vs-deep-copy)

|  |  |
| --- | --- |
| 1  2  3  4  5 | //Sample 07: Change the Data member value of Object 1  //           And print both Object 1 and Object 2  ob1.SetX(12);  ob1.PrintX();  ob2.PrintX(); |



Source

The output shows value 12 for both ob1 and ob2. Surprisingly, we modified the data member of the object ob1 only. Then, Why the changes are reflected on both the objects? This is what called ***shallow copy*** induced by the compiler provided default constructor. To understand this look at the below picture:



Source

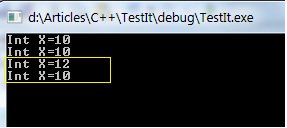
When object ob1 is created, the memory to store an integer is allocated in the heap. Let us assume the heap memory location address is 0x100B. This address is what stored in the x. Remember x is an integer pointer. The value stored in the pointer variable x is the address 0x100B and the content of the address 0x100B is value 10. In the example, we want to deal with the content of the address 0x100B we use the pointer ***de-referencing like \*x***. The compiler provided copy constructor copies the address stored in the ob1(x) to ob2 (x). After the copy, both pointers in ob1 and ob2 points to the same object. So changing the 0x100B through ob1.SetX(12) is reflected back in the ob2. Now you got how the result is printing 12 for both the objects ob1 and ob2.

How do we avoid the above-shown problem? We should perform the ***deep copy*** by implementing our own copy constructor. So a user defined copy constructor is required to avoid the problem of shallow copy. Below is the copy constructor:

[?](https://owlcation.com/stem/Copy-Constructor-shallow-copy-vs-deep-copy)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | //Sample 08: Introduce Copy Constructor and perform Deep Copy  ShalloC(const ShalloC& obj)  {      x = new int;      \*x = obj.GetX();  } |

Once we inject this copy constructor to the ShalloC class, the x pointer in the object ob2 will not point to the same heap location 0x100B. The statement ***x = new int;*** will create the new heap location and then copies the value of obj content to new heap location. The output of the program, after introducing our own copy constructor is shown below:



Source

The entire code is shown below:

[?](https://owlcation.com/stem/Copy-Constructor-shallow-copy-vs-deep-copy)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | // TestIt.cpp : Defines the entry point for the console application.  //    #include "stdafx.h"  #include <iostream>  using namespace std;    class ShalloC  {  //Sample 01: Private Data Member  private:      int \* x;    public:      //Sample 02: Constructor with single parameter      ShalloC(int m)      {          x = new int;          \*x = m;      }        //Sample 08: Introduce Copy Constructor and perform Deep Copy      ShalloC(const ShalloC& obj)      {          x = new int;          \*x = obj.GetX();      }        //Sample 03: Get and Set Functions      int GetX() const      {          return \*x;      }      void SetX(int m)      {          \*x = m;      }        //Sample 04: Print Function      void PrintX()      {          cout << "Int X=" << \*x << endl;      }        //Sample 05: DeAllocate the heap      ~ShalloC()      {          delete x;      }  };    int main()  {      //Sample 06: Create Object 1 and copy that to Object 2.      //           Print the data member for both Object 1 & 2.      ShalloC ob1(10);      ShalloC ob2 = ob1 ;      ob1.PrintX();      ob2.PrintX();        //Sample 07: Change the Data member value of Object 1      //           And print both Object 1 and Object 2      ob1.SetX(12);      ob1.PrintX();      ob2.PrintX();  } |

**4. Shallow copy vs Deep copy in c example:**

No. A shallow copy in this particular context means that you copy "references" (pointers, whatever) to objects, and the backing store of these references or pointers is identical, it's the very same object at the same memory location.

A deep copy, in contrast, means that you copy an entire object (struct). If it has members that can be copied shallow or deep, you also make a deep copy of them. Consider the following example:

typedef struct {

char \*name;

int value;

} Node;

Node n1, n2, n3;

char name[] = "This is the name";

n1 = (Node){ name, 1337 };

n2 = n1; // Shallow copy, n2.name points to the same string as n1.name

n3.value = n1.value;

n3.name = strdup(n1.name); // Deep copy - n3.name is identical to n1.name regarding

// its \*contents\* only, but it's not anymore the same pointer

# **Are array members deeply copied?**

In C/C++, we can assign a struct (or class in C++ only) variable to another variable of same type. When we assign a struct variable to another, all members of the variable are copied to the other struct variable. But what happens when the structure contains pointer to dynamically allocated memory and what if it contains an array?

In the following C++ program, struct variable st1 contains pointer to dynamically allocated memory. When we assign st1 to st2, str pointer of st2 also start pointing to same memory location. This kind of copying is called [Shallow Copy](http://en.wikipedia.org/wiki/Object_copy#Shallow_copy).

|  |
| --- |
| # include <iostream>  # include <string.h>    using namespace std;    struct test  {    char \*str;  };    int main()  {    struct test st1, st2;      st1.str = new char[20];    strcpy(st1.str, "GeeksforGeeks");      st2 = st1;      st1.str[0] = 'X';    st1.str[1] = 'Y';      /\* Since copy was shallow, both strings are same \*/    cout << "st1's str = " << st1.str << endl;    cout << "st2's str = " << st2.str << endl;      return 0;  } |

Run on IDE

Output:  
st1’s str = XYeksforGeeks  
st2’s str = XYeksforGeeks

Now, what about arrays? The point to note is that the array members are not shallow copied, compiler automatically performs [*Deep Copy*](http://en.wikipedia.org/wiki/Object_copy#Deep_copy) for array members.. In the following program, struct test contains array member str[]. When we assign st1 to st2, st2 has a new copy of the array. So st2 is not changed when we change str[] of st1.

|  |
| --- |
| # include <iostream>  # include <string.h>    using namespace std;    struct test  {    char str[20];  };    int main()  {    struct test st1, st2;      strcpy(st1.str, "GeeksforGeeks");      st2 = st1;      st1.str[0] = 'X';    st1.str[1] = 'Y';      /\* Since copy was Deep, both arrays are different \*/    cout << "st1's str = " << st1.str << endl;    cout << "st2's str = " << st2.str << endl;      return 0;  } |

Run on IDE

Output:  
st1’s str = XYeksforGeeks  
st2’s str = GeeksforGeeks

Therefore, for C++ classes, we don’t need to write our own copy constructor and assignment operator for array members as the default behavior is Deep copy for arrays.