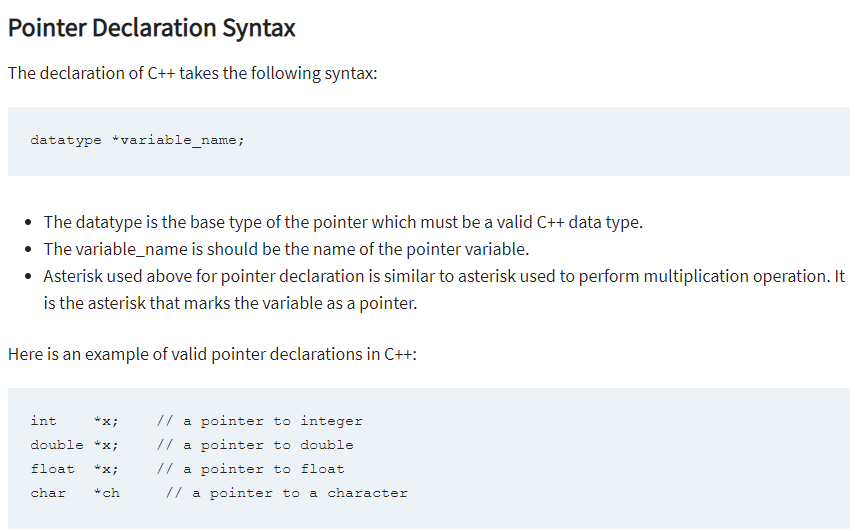
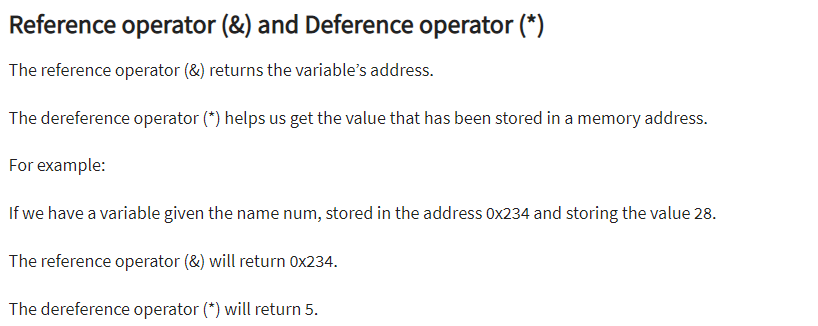
**Lecture -3**

**Last Class Question**

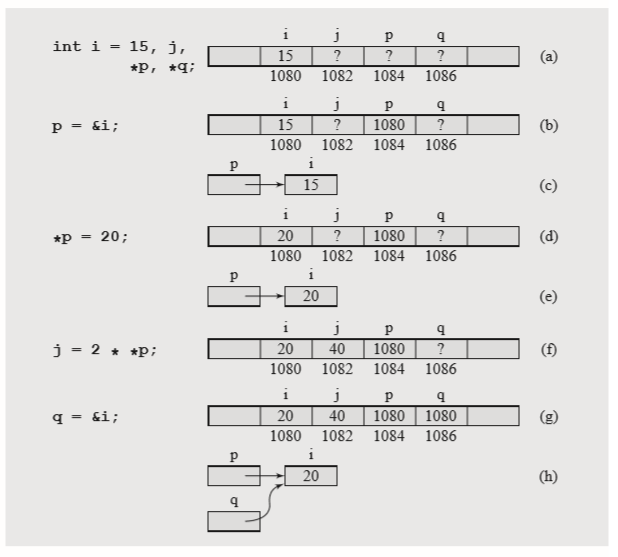
**Difference between Pointer and Reference**

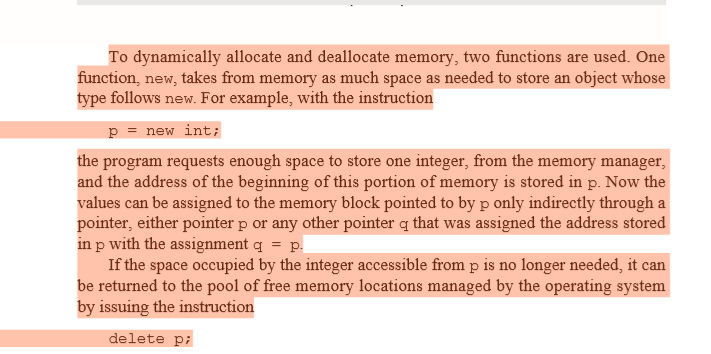
**Pointers and their uses**





Reference Picture from Book





include <iostream>

using namespace std;

int main() {

int x = 27;

int \*ip;

ip = &x;

cout << "Value of x is : ";

cout << x << endl;

cout << "Value of ip is : ";

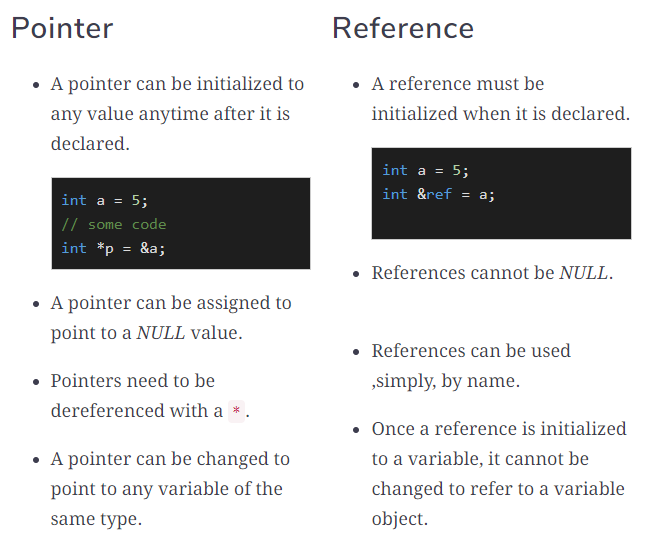
cout << ip<< endl;

cout << "Value of \*ip is : ";

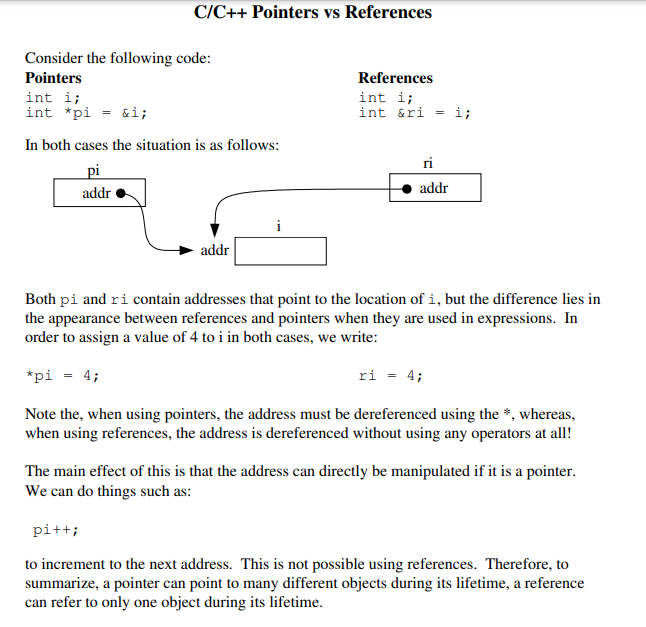
cout << \*ip << endl;

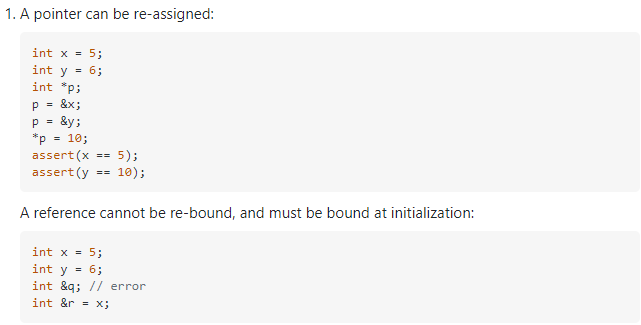
return 0;

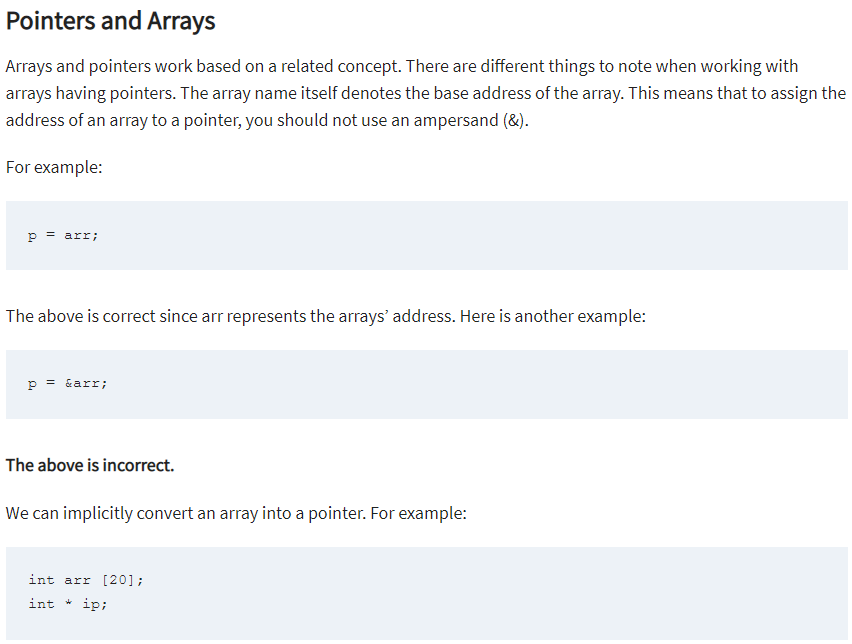
}

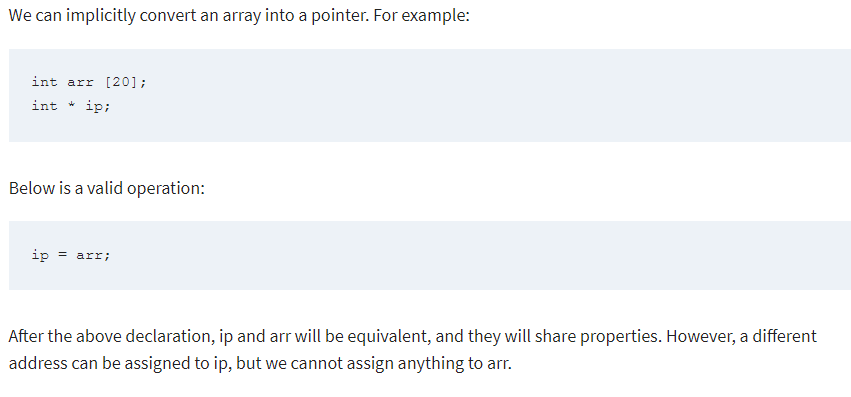
****

**Another Example**

****

**Example-4**





#include <iostream>

using namespace std;

int main() {

int \*ip;

int arr[] = { 10, 34, 13, 76, 5, 46 };

ip = arr;

for (int x = 0; x < 6; x++) {

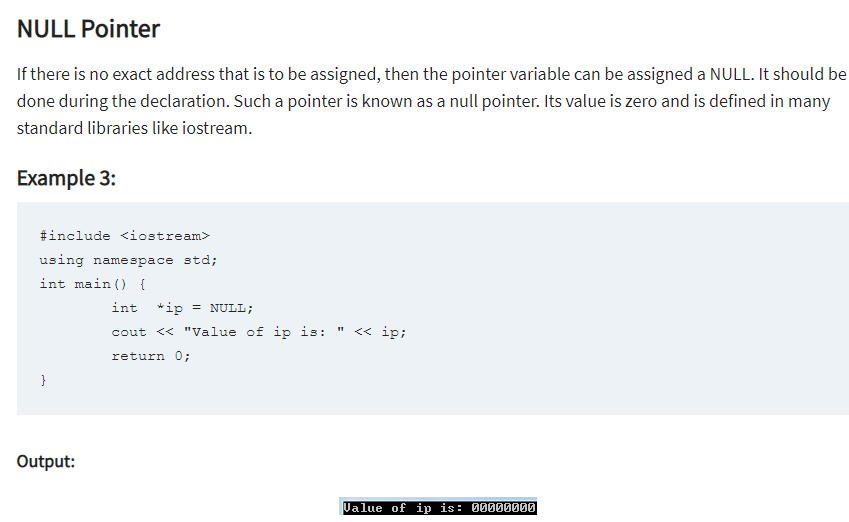
cout << \*ip << endl;

ip++;

}

return 0;

}



**Pointers of Variables**

With C++, you can manipulate data directly from the computer’s memory.

The memory space can be assigned or re-assigned as one wishes. This is made possible by Pointer variables.

Pointer variables point to a specific address in the computer’s memory pointed to by another variable.

It can be declared as follows:

int \*p;

Or,

int\* p;

In the you example, we have declared the pointer variable p.

It will hold a memory address.

The asterisk is the dereference operator that means a pointer to.

The pointer p is pointing to an integer value in the memory address.

#include <iostream>

using namespace std;

int main() {

int \*p, x = 30;

p = &x;

cout << "Value of x is: " << \*p;

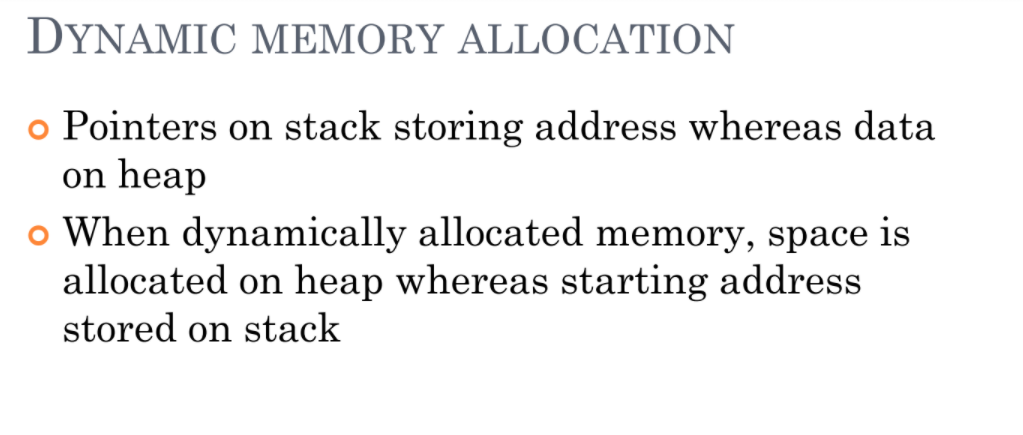
return 0;

}

Session Three Topics Start

**Lecture -3**

**Dynamic memory management**

****

**Code Snippets**

C:\Users\Anabia\source\repos\Project4 {4 Codes A, B, C}

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

## new and delete Operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

new data-type;

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new**operator with the following statements −

double\* pvalue = NULL; // Pointer initialized with null

pvalue = new double; // Request memory for the variable

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below −

**Important Point**

double\* pvalue = NULL;

if( !(pvalue = new double )) {

cout << "Error: out of memory." <<endl;

exit(1);

}

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the ‘delete’ operator as follows −

delete pvalue; // Release memory pointed to by pvalue

#include <iostream>

using namespace std;

int main () {

double\* pvalue = NULL; // Pointer initialized with null

pvalue = new double; // Request memory for the variable

\*pvalue = 29494.99; // Store value at allocated address

cout << "Value of pvalue : " << \*pvalue << endl;

delete pvalue; // free up the memory.

return 0;

}

Value of pvalue : 29495

Dynamic Memory Allocation for Arrays

**Code Snippets**

C:\Users\Anabia\source\repos\Dynamic Memory{A, B Notes {**Code B is Important**}}

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

char\* pvalue = NULL; // Pointer initialized with null

pvalue = new char[20]; // Request memory for the variable

To remove the array that we have just created the statement would look like this −

delete [] pvalue; // Delete array pointed to by pvalue

Following the similar generic syntax of new operator, you can allocate for a multi-dimensional array as follows −

double\*\* pvalue = NULL; // Pointer initialized with null

pvalue = new double [3][4]; // Allocate memory for a 3x4 array

However, the syntax to release the memory for multi-dimensional array will still remain same as above −

delete [] pvalue; // Delete array pointed to by pvalue

Dynamic Memory Allocation for Objects

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept –

#include <iostream>

using namespace std;

class Box {

public:

Box() {

cout << "Constructor called!" <<endl;

}

~Box() {

cout << "Destructor called!" <<endl;

}

};

int main() {

Box\* myBoxArray = new Box[4];

delete [] myBoxArray; // Delete array

return 0;

}

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result −

Constructor called!

Constructor called!

Constructor called!

Constructor called!

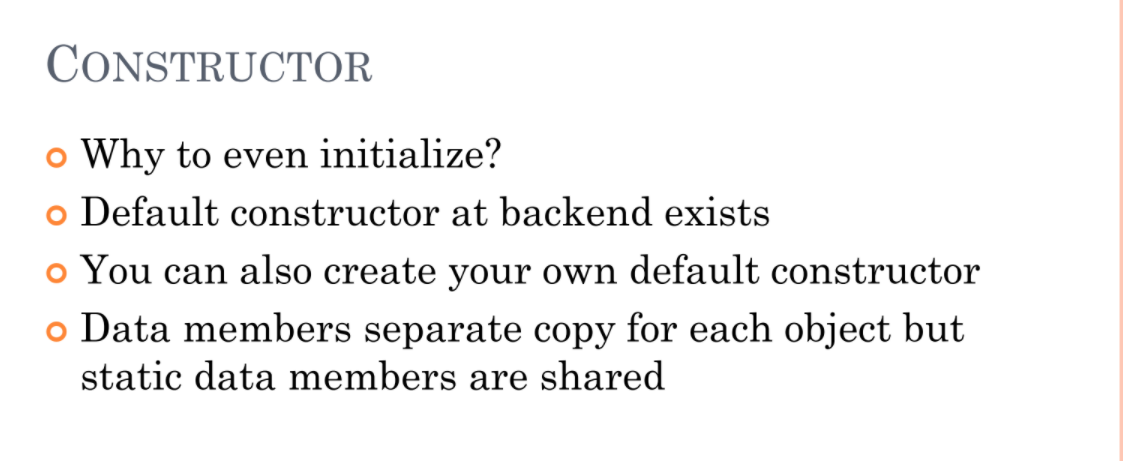
Destructor called!

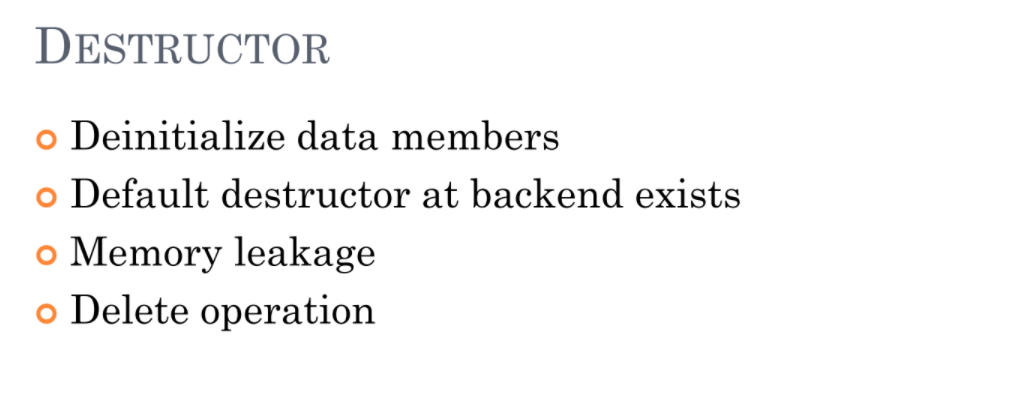
Destructor called!

Destructor called!

Destructor called!

**Constructors, destructors, Copy Constructor**

****

****

**Three Code Discuss For Constructor**

* **Default Constructor**
* **Constructor with Variable**
* **Constructor with out of the class**

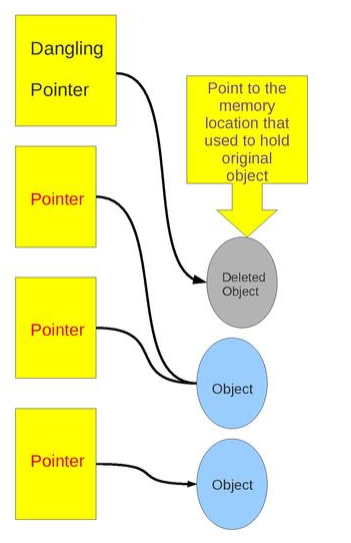
**What is a copy constructor?**  
A copy constructor is a member function that initializes an object using another object of the same class. A copy constructor has the following general function prototype:

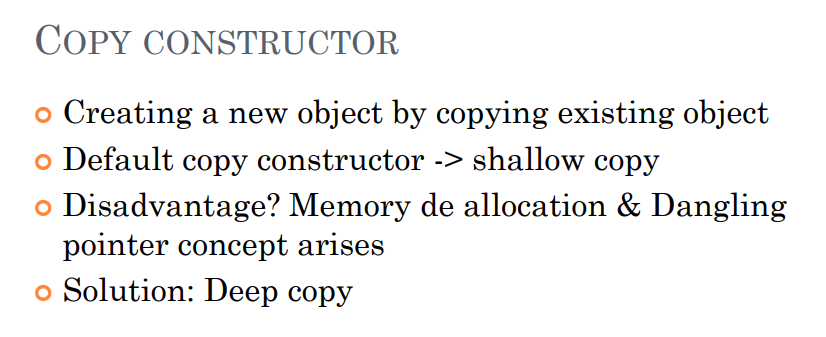
ClassName (const ClassName &old\_obj);

**Important Point**

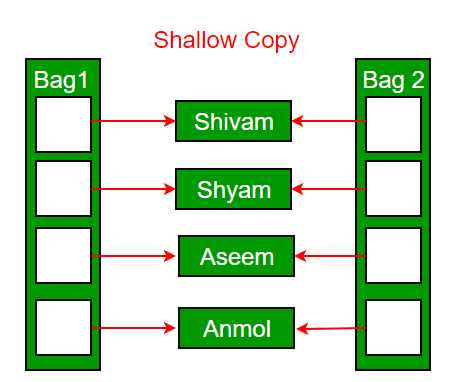
**When is** **copy constructor called?**   
In C++, a Copy Constructor may be called in the following cases:   
1. When an object of the class is returned by value.   
2. When an object of the class is passed (to a function) by value as an argument.   
3. When an object is constructed based on another object of the same class.   
4. When the compiler generates a temporary object.

Dangling Pointer Concept

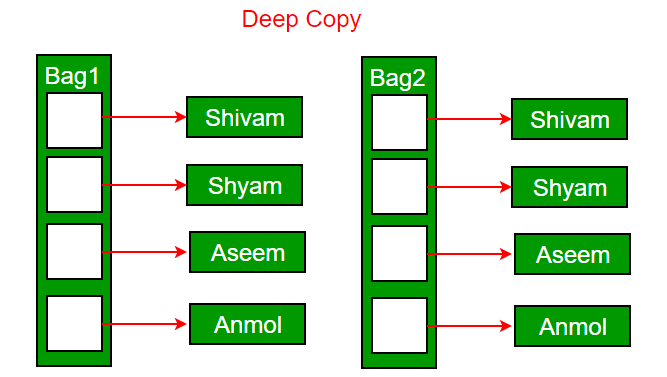
****

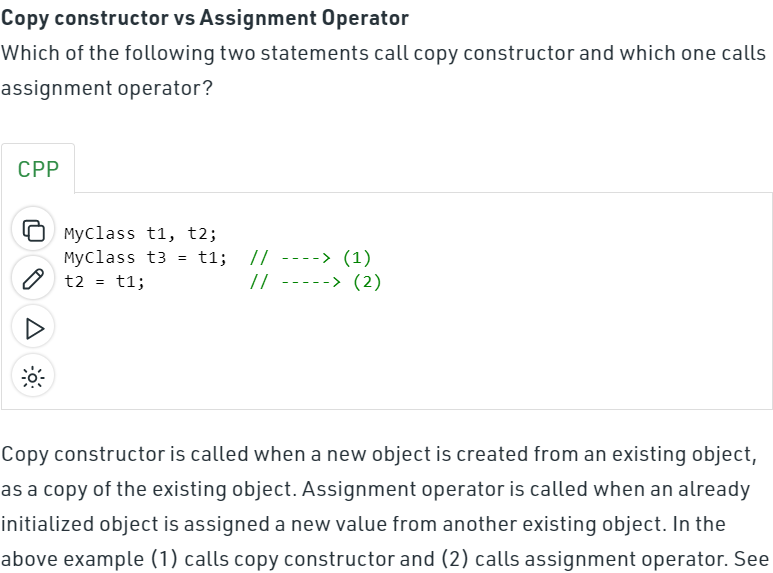
****

**When is**a **user-defined copy constructor needed?**   
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 filehandle, a network connection..etc.  
The default***constructor does only shallow copy.***



***Deep copy is possible only with user defined copy constructor.*** In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.



****

**Following is a complete C++ program to demonstrate use of Copy constructor. In the following String class, we must write copy constructor.**

**#**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("GeeksQuiz");

    String str2 = str1;

    str1.print(); // what is printed ?

    str2.print();

    str2.change("GeeksforGeeks");

    str1.print(); // what is printed now ?

    str2.print();

    return 0;

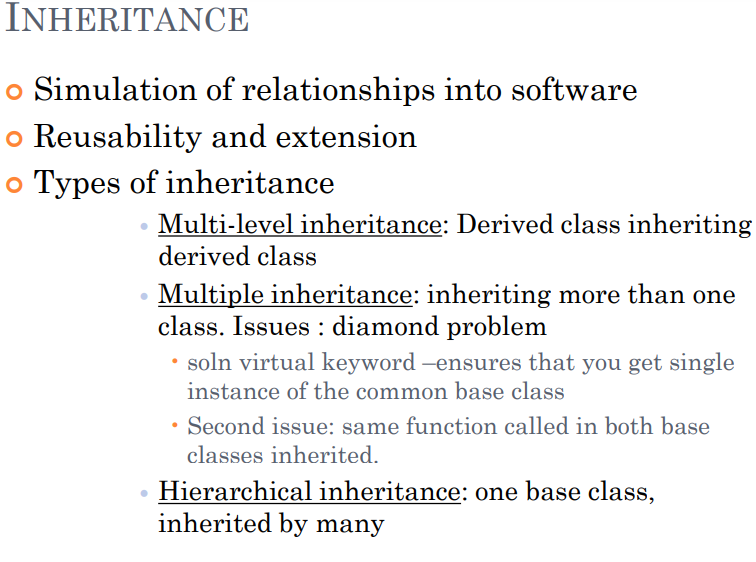
}

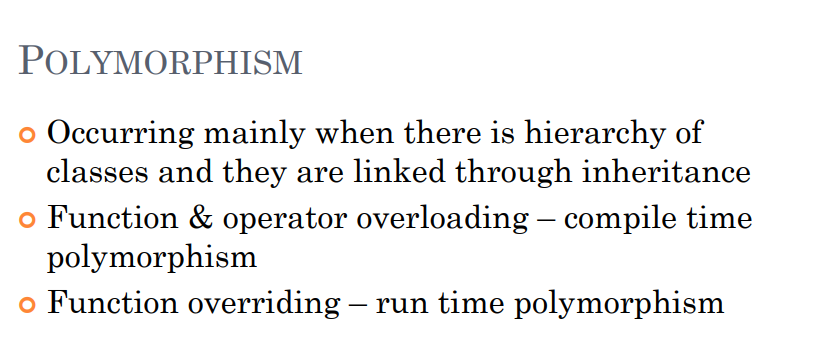
**Can we make copy constructor private?**

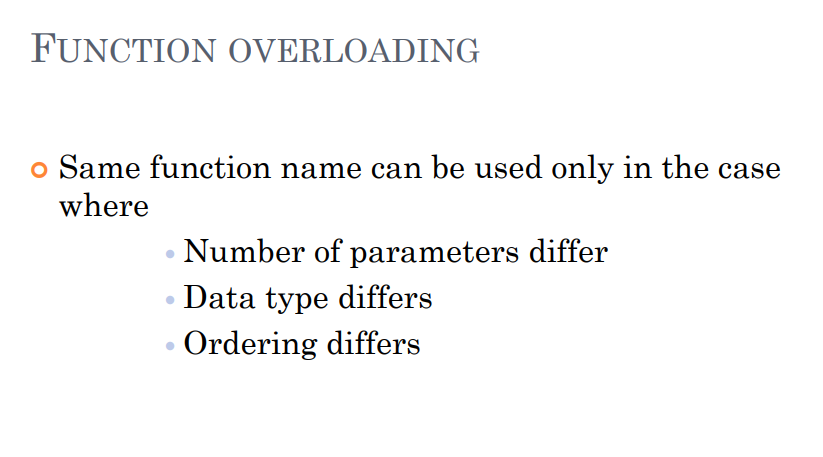
Yes, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable. This is particularly useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our own copy constructor like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

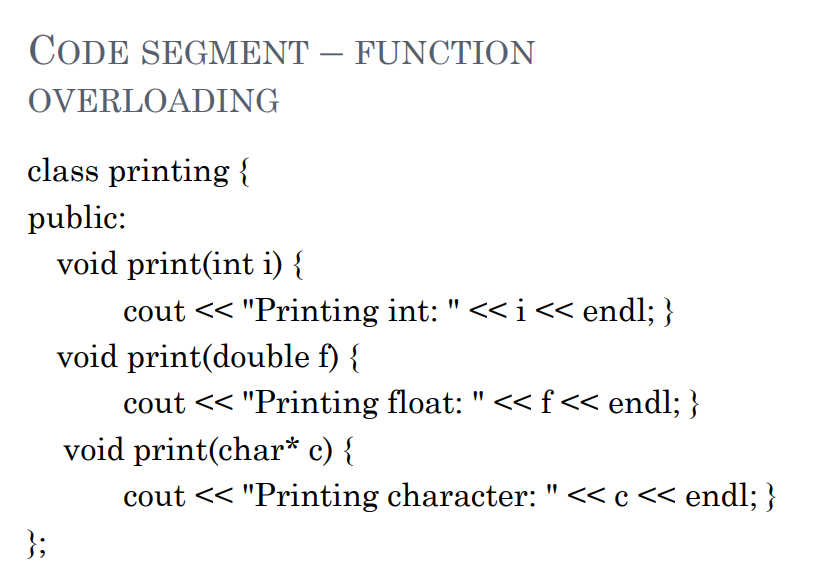
**Why argument to a copy constructor must be passed as a reference?**   
A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass an argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn’t allow parameters to be passed by value.

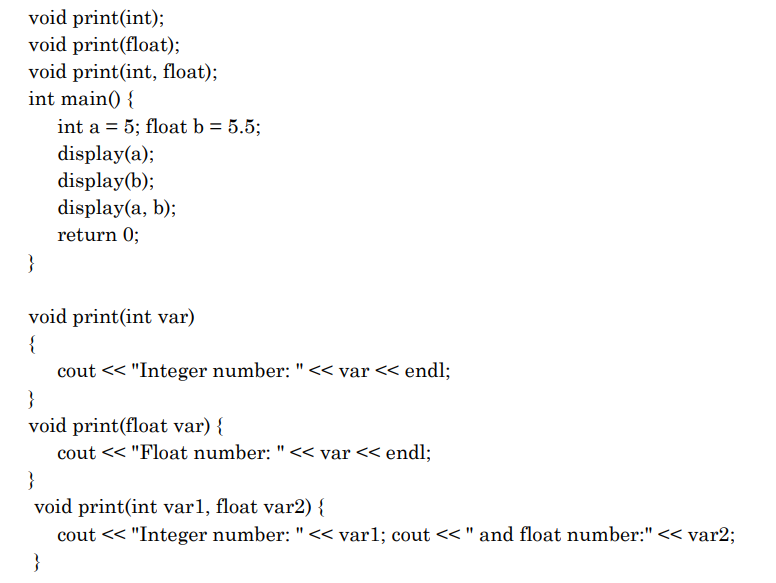
**polymorphic functions - overloading and overriding**

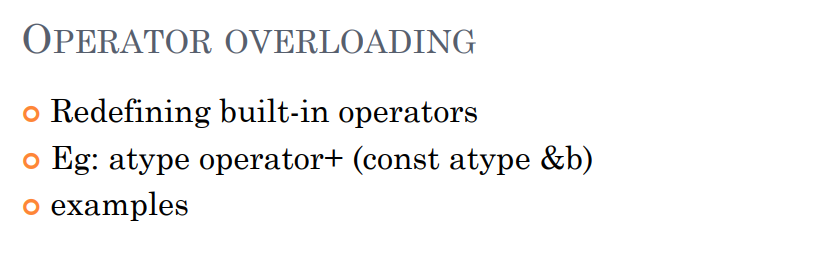
****

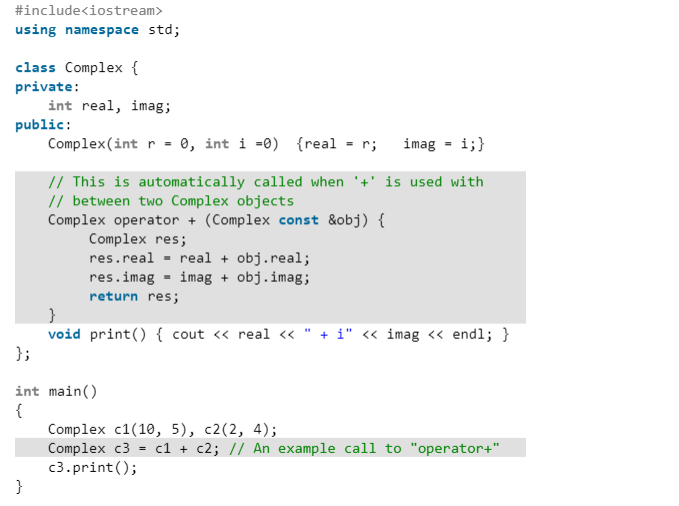
****

****

****

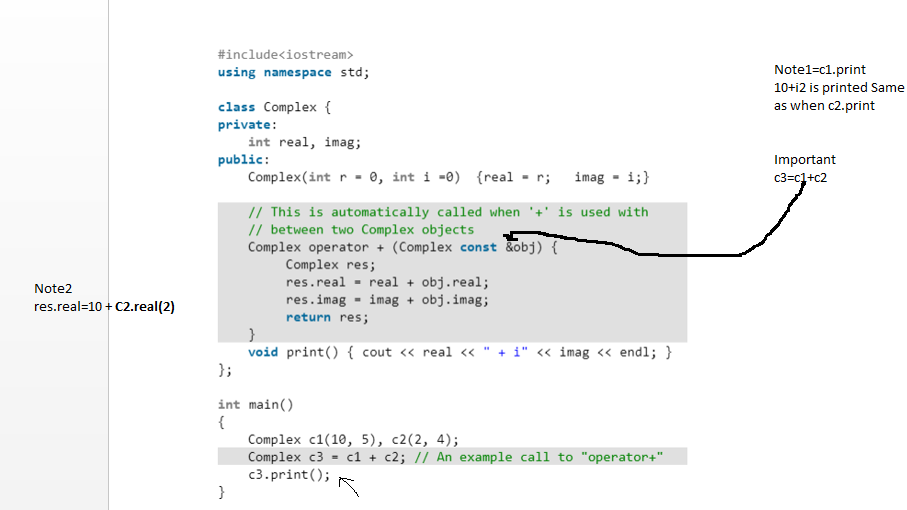
****

****

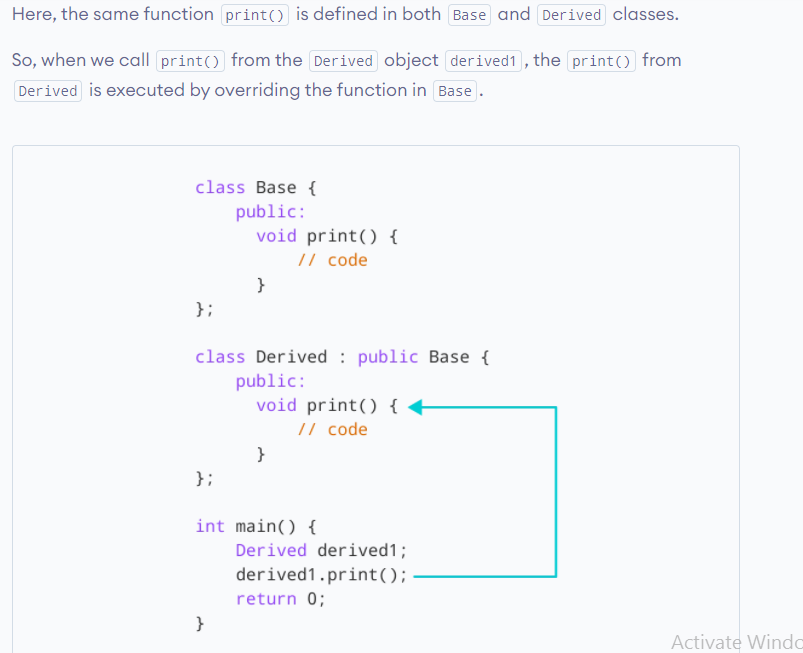
****

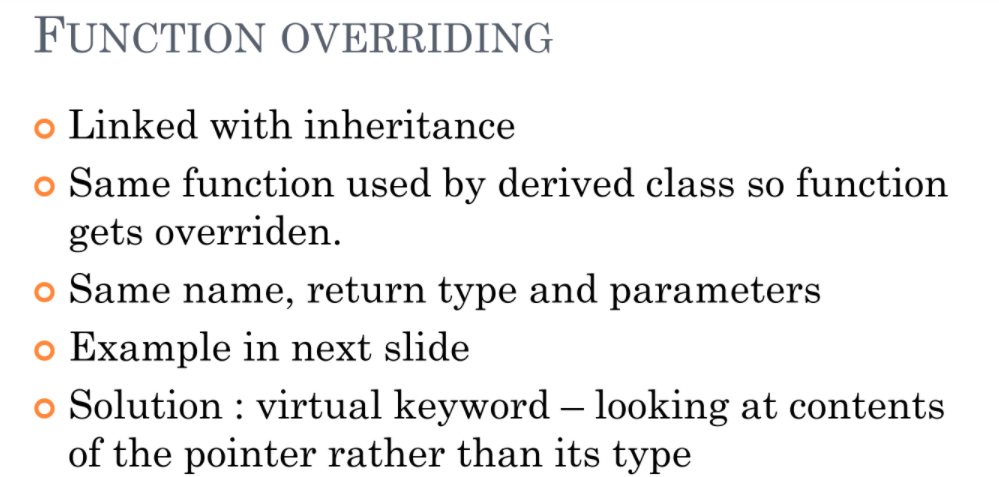
[**https://www.geeksforgeeks.org/operator-overloading-c/**](https://www.geeksforgeeks.org/operator-overloading-c/)

**Important Discuss the Output of that code**

****

**Function overridding Example**

****

****

**Code Example**

#include <iostream>

using namespace std;

class base {

public:

    virtual void print()

    {

        cout << "print base class" << endl;

    }

    void show()

    {

        cout << "show base class" << endl;

    }

};

class derived : public base {

public:

    void print()

    {

        cout << "print derived class" << endl;

    }

    void show()

    {

        cout << "show derived class" << endl;

    }

};

int main()

{

    base\* bptr;

    derived d;

    bptr = &d;

    // virtual function, binded at runtime

    bptr->print();

    // Non-virtual function, binded at compile time

    bptr->show();

}

**Rule of three** [**https://www.geeksforgeeks.org/rule-of-three-in-cpp/**](https://www.geeksforgeeks.org/rule-of-three-in-cpp/)