The copy constructor is a special kind of constructor which creates a new object which is a copy of an existing one, and does it efficiently.  
The copy constructor receives an object of its own class as an argument, and allows to create a new object which is copy of another without building it from scratch.  
  
  
Here below is a simple declaration of a copy constructor

class string

{

string();

~string();

string(const string &s)

{

copy(s.m\_str);

}

};

Now you can use it as follow:

*// create an object which is copy of another object*

string s1("hello");

string s2(s1); *// copy constructor activated*

*// create an object as a copy of a temporary object*

string s3(string("abc"));

string s4 = s1;

*// object s4 does not activate the constructor, but its copy constructor to make only a copy of s1, rather than building a new object*

you have to use const in the argument at the copy constructor to create an object as a copy of a temporary object: e.g. string(const string &s).

To make things clear, you can create a new object as copy of a different object without using a copy constructor, like this:

string s4;

s4.set(s1);

this is an example of inefficient code. Since s4 first call its constructor to build a new object and then it make a bit-wise copy of s1. The whole process of calling the constructor to build an object which next is being rewritten, is wasteful, takes time and resources. **Copy constructor allow you to prevents this inefficiency**. **<<- IMP**

Default copy constructor

If the programmer did not declare the copy constructor for a class, the compiler will add its own default copy constructor for the objects derived from that class.  
Default copy constructor does a very simple operation, they will do a bit-wise (member-wise) copy of an object, which means that the object will be copied bit by bit.

string s1("hello");

string s2(s1);

string s2 = s1; *//the same as above*

There is a danger in copying an object bit by bit, if the object contains pointers since the pointer address will be copied in the process resulting in two different objects that share the same memory buffer. You can imagine what will happen when two copies of an object calls their destructors one after the other. The first object that call its destructor will have no problems, since it will try to deallocate the pointer and succeed, but the second objects destructor try to deallocate a pointer which does not exist anymore and the whole application crashes!  
  
**IMP**  
**For a situation where an object contain pointers we have to write our own copy constructor that will prevent situations like those mentioned above (Shallow Copy)**.

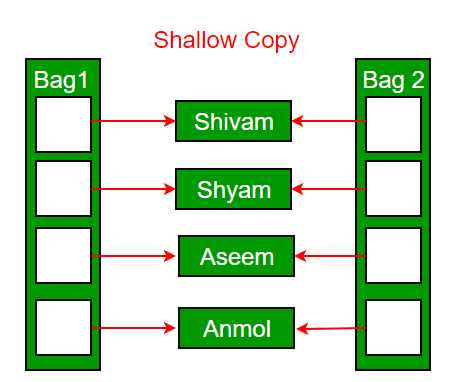
Copy constructor & passing objects as arguments to methods

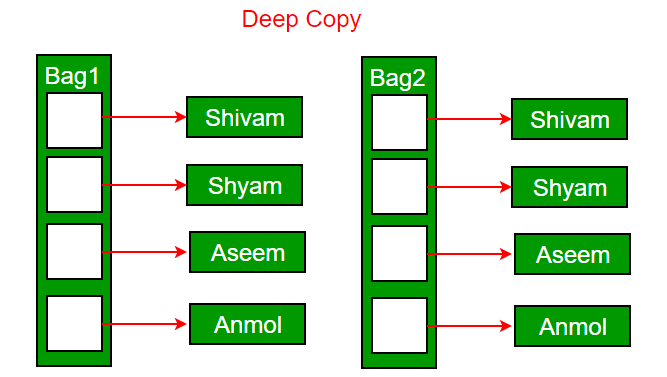
There are 3 situations in which the copy constructor is called:

1. When we make copy of an object.
2. When we pass an object as an argument by value to a method.
3. When we return an object from a method by value.
4. When compiler generates a temporary object.

we saw the first scenario above, and we will now look at the other two scenarios.   
  
When objects are passed to a function as arguments by value, a bit wise copy of the object will be passed to the function and placed on the stack, therefore the constructor of the object will not be called. It make sense if you think of it, you want to pass an object in a certain state containing the data you need for the function to process, if you wanted it in the initialization state with its default data, you could just create it inside the function instead of passing it as an argument.  
  
  
When the function end, the destructor of the object will be called. This also make sense since the object was passed by value and its data will not be needed outside the function scope.  
**IMP 🡪** Now pay attention to this, the situation in which only the object destructor is called can make great deal of troubles. Think what will happen when the object holds a pointer to some address in the memory. When this object is passed as argument to a function the pointer in the new temporary created object will hold the same address as the original object, since it’s a bit wise copy. When the function ends, the destructor will free the address pointed by the pointer. From this point, if the destructor of the original object will be called it will try to free an address which already free, and we all know what it the consequences of that.

**When is 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 file handle, a network connection..etc.

***Default constructor does only shallow copy.***  
[](https://media.geeksforgeeks.org/wp-content/uploads/copy-constructor.png)

***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.  
[](https://media.geeksforgeeks.org/wp-content/uploads/copy-constructor1.png)

**Copy constructor vs Assignment Operator**

Let’s look at an example to make things clear.  
Here we define string class which holds a char\* pointer:

Hide   Copy Code

class stringClass

{

public:

// constructor

stringClass(const char\* aStr)

{

str = new char[sizeof(aStr)];

strcpy (str,aStr);

}

// destructor

~stringClass()

{

delete str;

}

char\* getChars(){ return str; }

private:

**char\* str;**

};

now we will write a function that receive a string object as an argument.

Hide   Copy Code

void function (stringClass strObj)

{

cout<<strObj.str;

}

Lets look what will happen when we call this function :

Hide   Copy Code

void main ()

{

stringClass str("hello");

cout<<str.getChars()<<endl;

function(str); *// program crush*

}

Destructor will be called for “strObj” when function call ends and str pointer will be deleted. Then on end on main() destructor will be called for object “str” and tries to delete str pointer again and program crashes.  
the input argument on the stack is destroyed, and its destructor will be called, and delete the pointer.    
  
The copy constructor come to help us solve this kind of problems. Here is a solution:

#include <iostream>

#include <string.h>

using namespace std;

class stringClass

{

public:

// constructor

stringClass(const char\* aStr)

{

str = new char[sizeof(aStr)];

strcpy (str,aStr);

}

**// Copy Constructor**

**stringClass(const stringClass &obj){**

**str = new char[sizeof(obj.str)];**

**strcpy (str, obj.str);**

**}**

// destructor

~stringClass()

{

delete str;

}

char\* getChars(){ return str; }

private:

char\* str;

};

void function (stringClass str)

{

cout<<str.str;

}

int main( )

{

stringClass str("hello");

cout<<str.getChars()<<endl;

function(str); // Now program does not crush as user-defined copy constructor (Deep Copy) has replaced default copy constructor (shallow Copy)

return 0;

}

The same way we can handle the third scenario where a method returns an object:

class string

{

public:

string(char\* aStr)

{

str = new char[ strlen(aStr) + 1 ];

strcpy (str,aStr);

}

string(string &strObj)

{

str = new char[ strObj.str ];

strcpy( str, strObj.str );

}

string &string::operator=(const string &s)

{

string temp( s );

std::swap( temp.str, str );

return \*this;

}

*// destructor*

~string()

{

delete[] str;

}

private:

char\* str;

};

**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.

**Why argument to a copy constructor should be const?**

// version #1

Complx Maximum(const Complx &c1;, const Complx &c2;) {

if (c1.size() > c2.size())

return c1;

else

return c2;

}

// version #2 : This is our choice

const Complx & Maximum(const Complx &c1, const Complx &c2;) {

if (c1.size() > c2.size())

return c1;

else

return c2;

}

There are three points that can be emphasized:

1. Returning an object invokes the **copy constructor** while returning a reference doesn't. So, the version #2 does less work and is more efficient.

#include <iostream>

#include <string.h>

using namespace std;

class stringClass

{

public:

// constructor

stringClass(){

id++;

objId = id;

cout<<"default Ctor is called - "<<objId<<endl;

}

stringClass(const char\* aStr)

{

str = new char[sizeof(aStr)];

strcpy (str,aStr);

id++;

objId = id;

cout<<"Param Ctor is called - "<<objId<<endl;

}

// Copy Constructor

stringClass(const stringClass &obj){

str = new char[sizeof(obj.str)];

strcpy (str, obj.str);

id++;

objId = id;

cout<<"Copy Ctor is called - "<<objId<<endl;

}

//Assignment Operator

stringClass& operator = (const stringClass &obj){

str = new char[sizeof(obj.str)];

strcpy (str, obj.str);

cout<<"Assignment operator is called!!!"<<endl;

return \*this;

}

// destructor

~stringClass()

{

cout<<"Dtor is called for "<<this->objId<<endl;

delete str;

}

char\* getChars(){ return str; }

private:

static int id;

int objId;

char\* str;

};

int stringClass::id = 0;

void function (stringClass str) // No Copy constructor is called with pass by value

{

cout<<"function() - "<<str.getChars()<<endl;

}

stringClass& functionDuplicate (stringClass &str) // No Copy constructor is called with pass by reference

{

return str; // No Copy constructor is called with return by reference

}

stringClass functionDuplicate2 ()

{

stringClass strObj("Cat");

return strObj; // Copy constructor can be called. Complete discretion of Compiler. compiler creates temporary object.

}

int main( )

{

cout<<"----------------------------"<<endl;

stringClass str("hello");

cout<<str.getChars()<<endl;

function(str); // Now program does not crush as user-defined copy constructor (Deep Copy) has replaced default copy constructor (shallow Copy)

cout<<"----------------------------"<<endl;

stringClass str1;

str1 = functionDuplicate(str); // This will cause crash if user defined assignment operator is not defined.

cout<<str1.getChars()<<endl;

cout<<"----------------------------"<<endl;

stringClass str2;

str2 = functionDuplicate2(); // This will cause crash if user defined assignment operator is not defined.

cout<<str2.getChars()<<endl;

cout<<"----------------------------"<<endl;

return 0;

}