COPY CONSTRUCTORS IN CPP

Copy Constructors
One of the more important forms of an overloaded constructor is the copy constructor. Defining a copy constructor can help you prevent problems that might occur when one object is used to initialize another. Let's begin by restating the problem that the copy constructor is designed to solve. By default, when one object is used to initialize another, C++ performs a bitwise copy. That is, an identical copy of the initializing object is created in the target object. Although this is perfectly adequate for many cases—and generally exactly what you want to happen—there are situations in which a bitwise copy should not be used. One of the most common is when an object allocates memory when it is created.

For example, assume a class called MyClass that allocates memory for each object when it is created, and an object A of that class. This means that A has already allocated its memory. Further, assume that A is used to initialize B,
as shown here: MyClass B = A;
If a bitwise copy is performed, then B will be an exact copy of A. This means that B will be using the same piece of allocated memory that A is using, instead of allocating its own. Clearly, this is not the desired outcome. For example, if MyClass includes a destructor that frees the memory, then the same piece of memory will be freed twice when A and B are destroyed! The same type of problem can occur in two additional ways: first, when a copy of an object is made when it is passed as an argument to a function; second, when a temporary object is created as a return value from a function. Remember, temporary objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances. To solve the type of problem just described, C++ allows you to create a copy constructor, which the compiler uses when one object initializes another. Thus, your copy constructor bypasses the default bitwise copy.

The most common general form
of a copy constructor is
classname (const classname &o) {
    // body of constructor
}

Here, o is a reference to the object on the right side of the initialization. It is permissible for a

copy constructor to have additional parameters as long as they have default arguments defined

for them. However, in all cases the first parameter must be a reference to the object doing the
initializing. It is important to understand that C++ defines two distinct types of situations in

which the value of one object is given to another. The first is assignment. The second is
initialization, which can occur any of three ways:

- When one object explicitly initializes another, such as in a declaration
- When a copy of an object is made to be passed to a function
- When a temporary object is generated (most commonly, as a return value)

The copy constructor applies only to initializations. For example, assuming a class called

myclass, and that y is an object of type myclass, each of the following statements involves
initialization.

myclass x = y; // y explicitly initializing x
func(y); // y passed as a parameter
y = func(); // y receiving a temporary, return object

Following is an example where an explicit copy constructor is needed. This program creates a

very limited "safe" integer array type that prevents array boundaries from being overrun. (Chapter 15 shows a better way to create a safe array that uses overloaded operators.) Storage
for each array is allocated by the use of new, and a pointer to the memory is maintained within
each array object.

/* This program creates a "safe" array class. Since space for the array is allocated using new, a
copy constructor is provided to allocate memory when one array object is used to initialize
another. */

#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;
class array {
int *p; int
size; public:
array(int sz) {
try {
p = new int[sz];
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
exit(EXIT_FAILURE);
}
size = sz;
}
~array() { delete [] p; }
// copy constructor
array(const array &a);
void put(int i, int j) {
if(i>=0 & & i<size) p[i] = j;
} 
int get(int i) {
return p[i];
}
};
// Copy Constructor
array::array(const array &a) {
int i;
try {
p = new int[a.size];
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
exit(EXIT_FAILURE);
}
for(i=0; i<a.size; i++) p[i] = a.p[i];
} 
int main()
{
array num(10);
}
int i;
for(i=0; i<10; i++) num.put(i, i);
for(i=9; i>=0; i--) cout << num.get(i);
cout << "\n";
// create another array and initialize with num
array x(num); // invokes copy constructor
for(i=0; i<10; i++) cout << x.get(i);
return 0;
}

Let's look closely at what happens when num is used to initialize x in the statement array
x(num); // invokes copy constructor The copy constructor is called, memory for the new array
is allocated and stored in x.p, and the contents of num are copied to x's array. In this way, x
and num have arrays that contain the same values, but each array is separate and distinct. (That
is, num.p and x.p do not point to the same piece of memory.) If the copy constructor had not
been created, the default bitwise initialization would have resulted in x and num sharing the
same memory for their arrays. (That is, num.p and x.p would have indeed pointed to the same
location.) Remember that the copy constructor is called only for initializations.

For example, this sequence does not call the copy constructor defined in the preceding
program:
array a(10);
// ...
array b(10);
b = a; // does not call copy constructor

In this case, b = a performs the assignment operation. If = is not overloaded (as it is not here),
a bitwise copy will be made. Therefore, in some cases, you may need to overload the =
operator as well as create a copy constructor to avoid certain types of problems.