A copy constructor for data type X takes a const reference to X as its parameter:

```cpp
A a1;
// populate a1
A *a2 = new A(a1);
```

**Question 1.** What happens in the following code if our copy constructor does a shallow copy and the destructor is implemented as indicated?

```cpp
IntArray::~IntArray () { delete [] data; data = NULL; }
int main () {
    IntArray a1;
    IntArray a2 (a1);
    return 0;
}
```

In order to avoid this problem, you will need to do a deep copy:

```cpp
IntArray::IntArray (const IntArray &a) {
    size = a.size;
    data = new int[size];
    for (int i = 0; i < size; i++) data[i] = a.data[i];
}
```

The **Rule of Three** states that if you need implement one of the following three functions, then you should implement all three of them:

- Destructor
- Copy Constructor
- Assignment operator

**Using the STL**

We’ve talked about creating your own data structures. When you are doing general programming in C++, you will usually use the STL instead of creating your own for various common data structures. You may use them in CSCI 104L as well, unless we explicitly disallow the usage (in which case, you may not use them).

Today, we’ll see an STL implementation of the map concept (discussed in lecture 3), a pair class (which represents, oddly, a pair of values which may or may not be the same type) and iterator (which allows you to read through all the items in a data structure).
STL's map class

#include <map>
#include "student.h"

int main() {
    map<string, Student> slist1;
    Student s1("Tommy", 86328);
    Student s2("Jill", 54982);
    ...  
slist1["Tommy"] = s1; //associate the string Tommy with his student record.
    string myname = "Jill";
    slist1[myname] = s2;
    ...
    Student s3 = slist1["Tommy"];
    slist1.erase("Jill");
    return 0;
}

STL's pair class

Using Pairs:

std::pair<string, int> mypair("Bill", 1);
cout << mypair.first << " " << mypair.second << endl;
std::pair<char, double> p2(’c’, 2.3);

STL's iterator class

map<int, string> m;
...
map<int, string>::iterator it;
for (it = m.begin(); it != m.end(); ++it) {
    cout << it->second << endl;
}

it = m.find(42);
if (it != m.end()) cout << "meaning of life found: " << it->second << endl;

- The data structure has two public functions: begin(), which returns an iterator at the start of the data, and end() which returns an iterator at the end of the data.
- The iterator is a custom class defined within the scope of the data structure it iterates over. It overloads operator== and operator!= (so you can check if your iterator is at the end), operator++ (so you can get to the next piece of data), and operator* (so you can look at the value the iterator is currently sitting at).
- Every iterator in the STL is implemented in the same manner, so that you can always use an iterator for a data structure, even though you may not understand how the data structure works.
- Think of it like a pointer (it is not a pointer, but it has overloaded operator* to act like one).
- Also, think of the end() function as returning one PAST the end of the data structure, so the above for-loop works properly.