Some Notes on Copying Objects*

Jeffrey D. Oldham

2000 Mar 01

Contents

Caveat

These terse notes are *not* a substitute for reading the textbook, particularly Section 4.3. The textbook provides another example and a different explanation. In contrast, these notes are a rough draft.

1 Instances of Copying Objects

In C++, there are at least four different types of syntax to copy an object's contents. Consider the piece of code.

```
#include <iostream>
#include <cstdlib>

// WARNING!  THIS IS INCOMPLETE CODE!
class arrayHolder {
public:
  arrayHolder(unsigned int sz) {
    size = sz;
    array = new bool[sz];
  }
```

```
  bool& operator[](unsigned int pos) { return array[pos]; }
  ~arrayHolder(void) { delete [] array; }
private:
  bool * array;
  unsigned int size;
};
// WARNING!  THIS IS INCOMPLETE CODE!

class twoer {
public:
  int x;
  double y;
  // t is copied, not passed by reference.  This is silly to do, but
  // it illustrates copying function arguments.
  friend ostream& operator<<(ostream& out, const twoer t) {
    return out << t.x << ' ' << t.y;
  }
};


int main(void) {
  arrayHolder a(3);
  a[1] = true;   a[2] = false;
  arrayHolder b(a);            // Construct b by copying a.
  b[1] = false;
  arrayHolder c = b;           // Construct c by copying b.

  a = b;                       // Change a to be same as b.
  a = a;                       // Copy a to itself.  Silly but permissible.

  twoer t;
  t.x = 3;  t.y = -2.3e-14;
  operator<<(cout, t);         // Copy t as function argument.

  return EXIT_SUCCESS;
}
```

The first copy of an object occurs when `b` is initialized with `a`'s value. In C++ parlance, this is called a *copy construction*, i.e., an object is created and its contents should be the same as the given parameter. The second copy is just alternate syntax for a copy construction.

The last copy occurs in the `operator<<` function call. The function's second argument is a `twoer` object. This is also a copy construction. When it is constructed, the object's initial value is the same as the given parameter.

The third and fourth copies are *assignments*. The first assignment copies the second object's contents to the first argument. The second assignment of an object to itself is silly but legal.

2 Copy Constructors

A *copy constructor* is a class member function constructing an object and ensuring its contents are the same as the constructor's argument's contents. One common use is assigning values to function parameter variables.

A copy constructor for a class named `foo` would have the class member function prototype

```
foo(const foo &f)
```

i.e., it would take a reference to a `foo` object. (In the rare case that the constructor argument needs to be changed, omit the `const` modifier.)

The compiler defines a default copy constructor for a class not explicitly defining one. For example, the compiler automatically constructs a `twoer` copy constructor like

```
twoer(const twoer & t) {
  x = t.x;
  y = t.y;
}
```

The value of each data member of the argument is copied to the object being constructed.

For `arrayHolder`, the default copy constructor just copies the `array` pointer, not the object that is pointed to. After all, how could the compiler know the length of the dynamically allocated piece of memory? When the constructor finishes, both objects' `array` would point to the same piece of memory. Thus, if one object changed the value stored in, say, the array's first position, the value stored in the other object's array's first position would also be changed. If this is not desirable, an explicit copy constructor should be written.

```
arrayHolder(const arrayHolder & ah) {
  size = ah.size;
  array = new bool[size];
  // Copy values from ah.array to array.
}
```

Thus, one should write an explicit copy constructor when the default copying of members' values is not desired. The only example we have seen so far is when an object holds dynamically-allocated memory.

3 Assignments

An assignment operator for a class named `foo` would have the class member function prototype

```
foo& operator=(const foo &f)
```

(In the rare case that the constructor argument needs to be changed, omit the `const` modifier.)

Just as for copy constructors, the compiler defines a default assignment operator that copies the values of each data member. For example,

```
twoer& operator=(const twoer &t) {
  x = t.x;
  y = t.y;
  return *this;
}
```

The definitions of copy constructors and assignment operators are frequently very similar because they do a very similar things: They both copy values from one object into another. They differ in that

- an assignment returns the current object `*this`. Returning a value permits chaining assignments such as `x = y = z`.
- an assignment modifies an object that already has values assigned to its members.

One should write an explicit assignment operator when the default copying of members' values is not desired. The only example we have seen so far is when an object holds dynamically-allocated memory. For example, consider an assignment operator for `arrayHolder`.

```
arrayHolder& operator=(const arrayHolder & ah) {
  if (this != &ah) {
    delete [] array;
    size = ah.size;
    array = new bool[size];
    // Copy values from ah.array to array.
  }
```

```
  return *this;
}
```

Since the assignment is to an object with existing values for its data members and `array` points to dynamically-allocated memory, that memory must be `delete`d before copying `ah`'s values.

The other thing to remember when writing an assignment operator is to handle *self-assignment*, e.g., `x = x`. The `if` condition ensures the heart of the function does not occur for self-assignment. Consider what would happen under self-assignment without the `if` statement: The array would be `delete`d and all its information would be lost.

The `if` conditional checks for self-assignment. `*this` is the current object so `this` is the storage location for the current object. `&` yields the storage location of its argument `ah`. This, if `this == &ah`, self-assignment is occurring.