CIS 278 (CS1) Programming Methods: C++

Lesson 17: Using Pointers in Classes

Skip to Main Content

[Note: The inventory item example below was originally inspired by Tony Gaddis in "Starting Out with C++".]

Section 17.1: Introduction to the InventoryItem class

Several issues come up when one of the data members in a class is a pointer. In order to illustrate these issues, and also to practice our use of pointers and dynamic memory, particularly dynamic arrays and, even more particularly, C-strings, we will be working on an extended example, an inventory item class which we will name InventoryItem. Our class objects will each store data for only a single item in the store. (if someone wanted to use our class to store the inventory for an entire store, which is our intention, they would have to create a list of our inventory items.) In order to keep things simple, we will store only 2 pieces of data for each item: a description of the item (for example, "hammer"), and the quantity of this item currently on hand in the store.

This naturally leads us to an implementation with 2 private data members: a member we will name description that will be a C-string representing the description of the item, and a member we will name units that will be an int representing the quantity of the item. (It would probably be better to use a string object to represent the description of the item, but using a C-string instead of a string will give us an opportunity to work with C-strings and dynamic data.)

To get us started, we will provide 5 member functions: 2 constructors (a default constructor and a constructor that takes a C-string argument), a setInfo member function that sets the description and the units, a setUnits member function that sets only the units and leaves the description unchanged, and an insertion operator so that we can see whether our other member functions are working properly.

Here is a client program that we will use to test these first 5 member functions, along with the desired output the program should produce.

```
#include <iostream>
#include "invitem.h"
using namespace std;

int main()
{
    InventoryItem item1;
    InventoryItem item2("hammer");

    cout << "item1 is " << item1 << endl;
    cout << "item2 is " << item2 << endl;

    item1.setInfo("screwdriver", 5);
    item2.setUnits(9);

    cout << "item1 is " << item1 << endl;
    cout << "item2 is " << item2 << endl << endl;
}
```

```
Desired Output:

item1 is 0
item2 is 0 hammer
item1 is 5 screwdriver
item2 is 9 hammer
```

Here is our initial header file:

```
#ifndef INVITEM_H
#define INVITEM_H
#include <iostream>

class InventoryItem {
    public:
        InventoryItem();
        InventoryItem(const char *inDesc);
        void setInfo(const char *inDesc, int inUnits);
        void setUnits(int inUnits);
        friend std::ostream& operator<<(std::ostream& out, const InventoryItem& printMe);
    private:
        char *description;
        int units;
};

#endif
```

There is one thing I've done in this class declaration that may be new to you. When a pointer is passed to a function the data that it is pointing to does not get modified inside the function, the reserved word const should be used in the parameter list, exactly like we would use it if we were passing an array.

Now we need to write each of the 5 member functions. This would be a good time to stop reading and see if you can implement these 5 functions yourself before continuing.

Note: as we write each member function for this class, we will show just the new member function instead of showing the entire implementation file repeatedly. For your reference the entire implementation file can be found at the end of this document.

Let's start with the constructor that has a C-string argument. In this constructor, we will set the units data member to 0 and the description data member to the parameter inDesc. Because assignment of C-strings with the assignment (=) operator is not allowed in C++, we will use the strcpy function (see Savitch section 9.1). Here is a first attempt at the code:

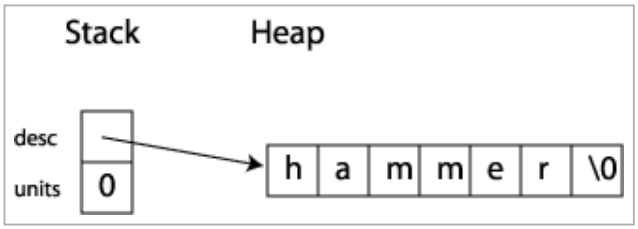
```
InventoryItem::InventoryItem(const char *inDesc)
{
    units = 0;
    strcpy(description, inDesc);
}
```

It is very important that you understand what is wrong with this code. It is very likely that if I ran the program using this code I would get an unhandled exception. Why? Because I have tried to copy information into a C-string pointed to by description, but I have not yet allocated any memory for it! Put another way: description is currently an uninitialized pointer. Before I can call strcpy I must use new to allocate memory for a C-string.

When I allocate memory for a dynamic array to store the C-string, I must specify how big to make the array. In this case, we want the array to be just big enough to store the C-string inDesc. We will use the strlen function to determine how many characters are in the C-string inDesc, then we will add one to this figure because we need to include enough space for the '\0' that comes at the end of every C-string.

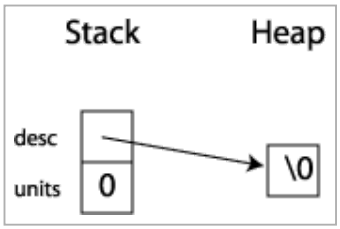
Here is the correct code, followed by a picture of the situation after the code is executed (using the declaration InventoryItem item2("hammer") from the client program).

```
InventoryItem::InventoryItem(const char *inDesc)
{
    units = 0;
    description = new char[strlen(inDesc) + 1];
    strcpy(description, inDesc);
}
```



Next let's write our default constructor. In this constructor, we will set the units data member to 0. What should we set the description data member to? One possible approach would be to set it to NULL. This could work, but then everytime we wanted to do an operation using the description data member that assumed that it was a C-string we would have to use an if statement to determine whether description was NULL or whether it was a valid C-string. A better approach would be to have a class invariant that the description data member always stores a valid C-string. This approach will make the rest of our work much more convenient. In order to satisfy this class invariant, instead of simply assigning description the value NULL, we will create a null string (that is, a string with 0 characters) and assign it to description. Here is the correct code, followed by a picture of the situation after the code is executed:

```
InventoryItem::InventoryItem()
{
    units = 0;
    description = new char[1];
    strcpy(description, "");
}
```



There are several variations in how we could have written this code. For example, the second statement in the function could have been written description = new char;, which may seem a bit simpler. The third statement is equivalent to the statement description[0] = '\0'; or *description = '\0';. We chose the statements as they appear in the function above because these seemed to be the most consistent with the statements in the other constructor.

Next we will write the setInfo member function. This function will be very similar to our constructors. An important difference is that in our constructors we were constructing an InventoryItem object from nothing, starting with an uninitialized pointer in the description data member. With our setInfo member function we will be dealing with a description data member that already has some value and we will be reassigning it. Does this mean we don't have to allocate memory for it (since it already has memory allocated for it)? Making this assumption might result in the following code:

```
void InventoryItem::setInfo(const char *inDesc, int inUnits)
{
    units = inUnits;
    strcpy(description, inDesc);
}
```

The problem with this code is that we don't know whether description is the right size to hold the C-string stored in inDesc. description might be too big, in which case we would be okay but we would be using our memory inefficiently. Worse, description might be too small, in which case the call to strcpy would overwrite some memory which has not been allocated. This has unpredictable results, but often the result is program termination with an unhandled exception error message. Not good.

So is there some way to resize the description data member so that it is the right size to be able to store the C-string stored in inDesc? Not in one step. The only way to handle this situation is to completely deallocate the

memory that description is currently pointing to and the reallocate it to be the right size. Here is the correct code:

```
void InventoryItem::setInfo(const char *inDesc, int inUnits)
{
    units = inUnits;
    delete [] description;
    description = new char[strlen(inDesc) + 1];
    strcpy(description, inDesc);
}
```

Notice that this code turns out to be very similar to the code for our constructor except that we have added the delete statement. Be careful not to have a delete statement in your constructor. You don't have to delete anything in your constructor because the pointers are already uninitialized. Having an extraneous delete statement in your constructor is for novices a very common and difficult to track down error. It often results in an unhandled exception, but the unhandled exception does not happen at the point of the delete, making this error very difficult to isolate.

The setUnits member function and the insertion operator are trivial. The code is given here to complete our discussion of our first 5 functions. With these 5 functions in place we get the desired output shown with the client program at the beginning of this section.

```
void InventoryItem::setUnits(int inUnits)
{
    units = inUnits;
}

ostream& operator<<(ostream& out, const InventoryItem& source)
{
    out << source.units << " " << source.description;
    return out;
}
```

Section 17.2: The Assignment Operator

We will now proceed to extend the client program given at the beginning of section 1, making sure that everything in our class works as it should. In the process we will discover that **there are 3 member functions that must be included in any class that uses dynamic memory**. These three functions are commonly referred to as the "big-3".

Let's begin by adding 8 lines to the client program of section 1, as illustrated here along with the expected output:

```
#include <iostream>
#include "invitem.h"
using namespace std;

int main()
{
    InventoryItem item1;
    InventoryItem item2("hammer");

    cout << "item1 is " << item1 << endl;
    cout << "item2 is " << item2 << endl;

    item1.setInfo("screwdriver", 5);
```

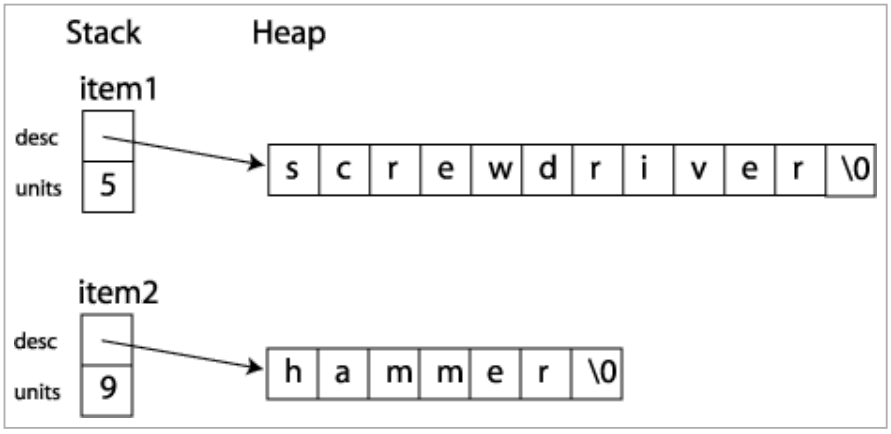
```
    item2.setUnits(9);

    cout << "item1 is " << item1 << endl;
    cout << "item2 is " << item2 << endl << endl;

    item1 = item2;
    cout << "after item1 = item2, " << endl;
    cout << "item1 is " << item1 << endl;
    cout << "item2 is " << item2 << endl << endl;

    item2.setInfo("lawn mower", 14);
    cout << "after item2.setInfo(\"lawn mower\", 14), " << endl;
    cout << "item1 is " << item1 << endl;
    cout << "item2 is " << item2 << endl << endl;
}
```

```
Expected Output:

item1 is 0
item2 is 0 hammer
item1 is 5 screwdriver
item2 is 9 hammer

after item1 = item2,
item1 is 9 hammer
item2 is 9 hammer

after item2.setInfo("lawn mower", 14),
item1 is 9 hammer
item2 is 14 lawn mower
```

This all seems straightforward, but watch what happens when I test this client program to make sure I get the expected output:

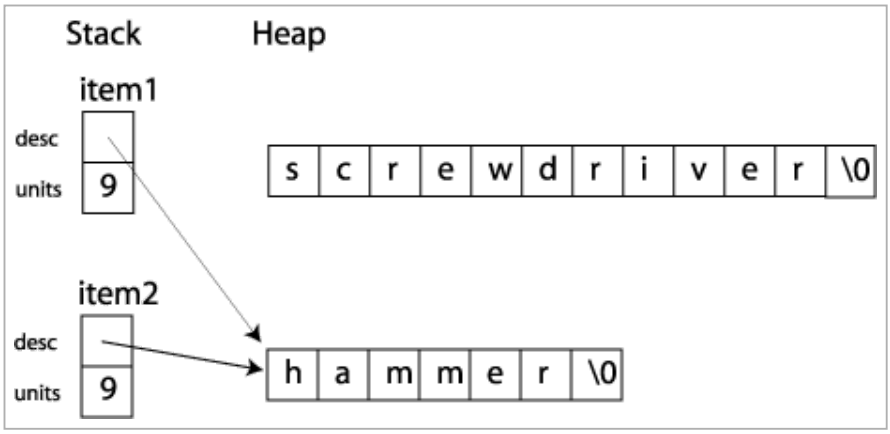
```
Actual Output:

item1 is 0
item2 is 0 hammer
item1 is 5 screwdriver
item2 is 9 hammer

after item1 = item2,
item1 is 9 hammer
item2 is 9 hammer

after item2.setInfo("lawn mower", 14),
item1 is 9 lawn mower
item2 is 14 lawn mower
```

Look carefully at this output. Somehow when we called setInfo to change the value of item2, the value of item1 was modified! Let's draw a picture of our situation and see if we can figure out how this happened. Here is the situation right before we execute the assignment statement:

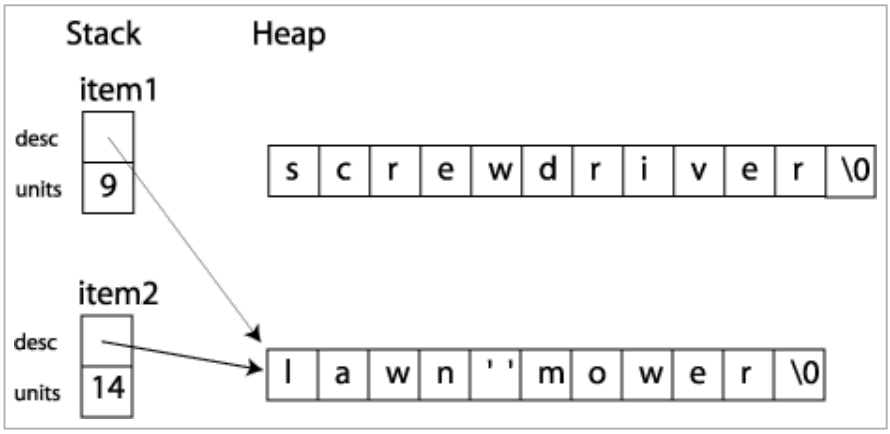


When we assign one object to another C++ performs what is called a "memberwise assignment". In other words, each data member in the object on the left is assigned to the corresponding data member in the object on the right. The description data member is a pointer. As we stated in lesson 4.2.4, when we assign one pointer to another no data changes. The only change is in where the pointer is pointing. So one result of the assignment statement is that the description data member of item1 now points to the variable

that the description data member of item2 points to. Here is the situation right after the assignment statement is executed:



This situation is bad because we now have a memory leak: that C-string that item1.description used to be pointing to is no longer accessible, and so we have no way of either using that value or of returning the memory to the heap for later use. But the next two cout statements work fine, since item1.description and item2.description both point to the C-string "hammer". But then what happens when we execute the call to setInfo? Here is the situation after the call to setInfo:



When we use setInfo to change the value of item2, we are also changing the value of item1, because item1.description and item2.description are both pointing to the same piece of memory. The problem here is that when we do a member-wise assignment (which is the default in C++) we are doing what is called a "shallow copy". It is called a shallow copy because we are copying only the pointers, instead of also copying the variables that the pointers are pointing to. What we need to do is tell C++ to have the assignment operator perform a deep copy instead of a shallow copy. We do this by overloading the assignment operator. When we overload the assignment operator C++ uses the function we have provided instead of its default assignment operator that does a shallow copy.

This would be a good time to stop reading and see if you can write the assignment operator on your own before continuing.

Here is our first attempt at overloading the assignment operator:

```
InventoryItem InventoryItem::operator=(const InventoryItem& right)
{
    units = right.units;
    description = right.description;
}
```

This code won't do us much good. In fact this code is essentially what C++ does automatically if we fail to provide an assignment operator: a shallow copy. We need to copy not the pointer right.description, but the variable that the pointer right.description is pointing to. Here is another attempt:

```
InventoryItem InventoryItem::operator=(const InventoryItem& right)
{
    units = right.units;
    description = new char[strlen(right.description) + 1];
```

```
    strcpy(description, right.description);
}
```

By now these three lines of code should be looking familiar. They are very similar to what we saw in the constructors and the setInfo function. There are still two issues to resolve. First, when this function begins execution description is pointing at something on the heap. We need to use the delete operator to return this memory to the heap for future use. Second, although you may not have realized it, the assignment operator in C++ returns a value. It returns whatever value was assigned. This means that, although it would be bad practice to do so, we could use the assignment operator like this:

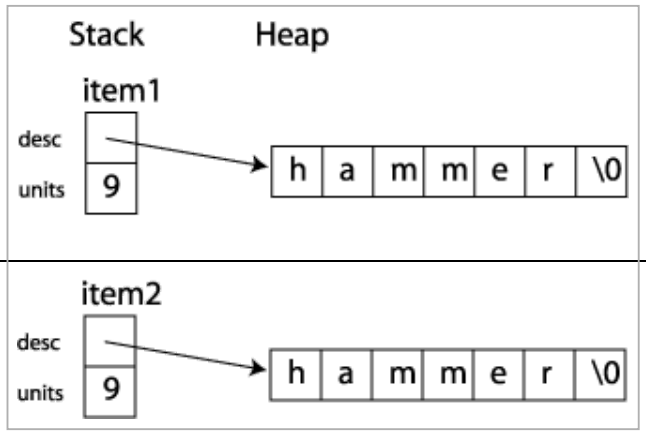
```
int i;
cout << (i = 7) << endl;
```

In this case, the number 7 would get printed on the screen, because the assignment operator returns the value that was assigned (7 in this case). To summarize: we must have a return statement in our assignment operator that returns the value that was assigned. Because the variable that we are assigning the value to is the calling object, the easiest way to return the value that was assigned is to return the calling object (*this).

These two additional issues are resolved in this version of our assignment operator:

```
InventoryItem InventoryItem::operator=(const InventoryItem& right)
{
    units = right.units;
    delete [] description;
    description = new char[strlen(right.description) + 1];
    strcpy(description, right.description);
    return *this;
}
```

If we add this function to our implementation file, add its prototype to our header file, and run the client program from the beginning of section 2, we will now get the expected output. Here is a picture of the situation after the assignment statement is executed. You should compare this to the situation we had at the same point when we used a shallow copy. (In that picture, item1.description and item2.description both pointed to the same C-string, "hammer".) You can see that now when we change item2.description to "lawn mower" item1.description will not be affected.



We have one more case to test on our assignment operator. Even though it seems unusual, our assignment operator should certainly be able to handle a statement like "item1 = item1;". This is called self-assignment. Let's add the following 3 statements to our client program to check that our assignment operator handles this situation correctly:

```
item1 = item1;
cout << "after item1 = item1, ";
cout << "item1 is " << item1 << endl << endl;
```

Here is the output I get when I run the client program, including these last three statements, with the header file and implementation file we have created so far:

```
item1 is 0
```

```
item2 is 0 hammer
item1 is 5 screwdriver
item2 is 9 hammer

after item1 = item2,
item1 is 9 hammer
item2 is 9 hammer

after item2.setInfo("lawn mower", 14),
item1 is 9 hammer
item2 is 14 lawn mower

after item1 = item1, item1 is 9 Â \er
```

The value of item1 got destroyed! Recall that our assignment operator first deletes the memory location that the description data member of the calling object is pointing to, then assigns to it the value of right.description. If the calling object and right are really the same object, we destroyed right when we delete the calling object. We need to modify our function so that in the case of a self-assignment, when the calling object and right are really the same object, the value of the calling object does not get destryoed. There are several ways to do this, but the easiest is to add an if statement so that the assignment is only done if the calling object and right are not the same object. Our first attempt at this might be to say

```
if (*this != right){....
```

but the problem with this condition is that we have not defined a not-equal-to operator for InventoryItem objects. In addition, the issue here is not whether the two objects are equal, but whether the two objects are actually the SAME object. For these two reasons, instead of comparing the two objects for equality, we will compare the addresses of the two objects for equality. If the addresses are equal, that means that we have a self-assignment situation. Here is the correct and final (at last!) code for the assignment operator.

```
InventoryItem InventoryItem::operator=(const InventoryItem& right)
{
    if (this != &right){
        units = right.units;
        delete [] description;
        description = new char[strlen(right.description) + 1];
        strcpy(description, right.description);
    }
    return *this;
}
```

Section 17.3: The Copy Constructor

In section 2 we learned that **there are 3 member functions that must be included in any class that uses dynamic memory**, and that one of those 3 is the **assignment operator**. Next I want to make sure that pass-by-value parameter passing works correctly for InventoryItem objects. So I will add a function to my client program and pass an InventoryItem object to the function using pass-by-value. My function will change its local copy of the object -- but this change should have no effect on the object's value in main because it was passed by value. Here is my new client program which includes this new function and also 3 lines of code in main to see if the results are correct. Also included is the expected output.

```
#include <iostream>
#include "invitem.h"
using namespace std;

void f(InventoryItem item1);

int main()
{
    InventoryItem item1;
```

```
    InventoryItem item2("hammer");

    cout << "item1 is " << item1 << endl;
    cout << "item2 is " << item2 << endl;

    item1.setInfo("screwdriver", 5);
    item2.setUnits(9);

    cout << "item1 is " << item1 << endl;
    cout << "item2 is " << item2 << endl << endl;

    item1 = item2;
    cout << "after item1 = item2, " << endl;
    cout << "item1 is " << item1 << endl;
    cout << "item2 is " << item2 << endl << endl;

    item2.setInfo("lawn mower", 14);
    cout << "after item2.setInfo(\"lawn mower\", 14), " << endl;
    cout << "item1 is " << item1 << endl;
    cout << "item2 is " << item2 << endl << endl;

    item1 = item1;
    cout << "after item1 = item1, ";
    cout << "item1 is " << item1 << endl << endl;

    f(item1);
    cout << "after being used as an argument to pass-by-value parameter, ";
    cout << endl << "item1 is still " << item1 << endl << endl;
}

void f(InventoryItem item1)
{
    item1.setInfo("pizza", 67);
}
```

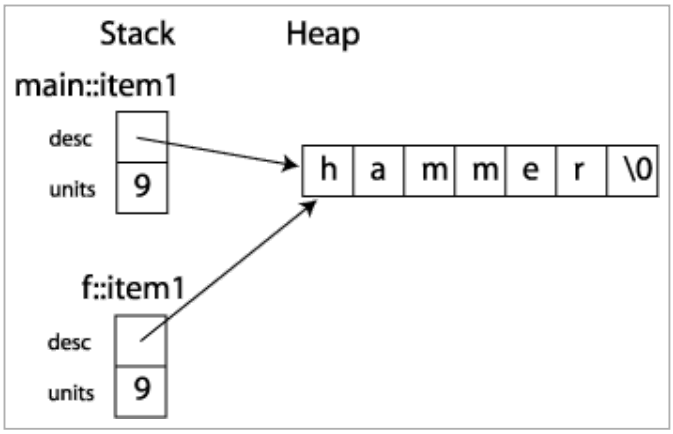
```
Expected Output:

item1 is 0
item2 is 0 hammer
item1 is 5 screwdriver
item2 is 9 hammer

after item1 = item2,
item1 is 9 hammer
item2 is 9 hammer

after item2.setInfo("lawn mower", 14),
item1 is 9 hammer
item2 is 14 lawn mower

after item1 = item1, item1 is 9 hammer
after being used as an argument to pass-by-value parameter,
item1 is still 9 hammer
```

As you may have guessed, this expected output is not what we get. Here is the actual output:

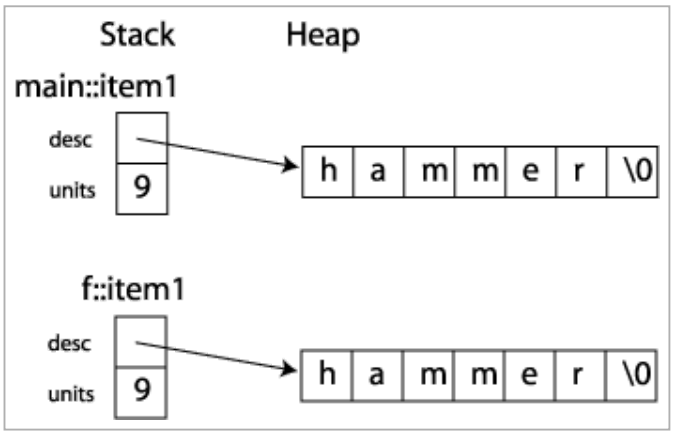
```
Actual Output:

item1 is 0
item2 is 0 hammer
item1 is 5 screwdriver
item2 is 9 hammer

after item1 = item2,
item1 is 9 hammer
item2 is 9 hammer

after item2.setInfo("lawn mower", 14),
item1 is 9 hammer
item2 is 14 lawn mower

after item1 = item1, item1 is 9 hammer
after being used as an argument to pass-by-value parameter,
item1 is still 9 pizza
```

What's happening here is very similar to what happened with the assignment operator before we overloaded it. When we use pass-by-value, a copy is made of the argument and used as a local variable in the function. When

C++ makes the copy, the default is to use a shallow copy. Here is a picture of the situation after the parameters have been properly set up but before any statements inside the function f have been executed:



You can see that when we now proceed to set f's item1.description to pizza, we are inadvertently changing main's item1.description as well. We would like our picture to look like this:



Fortunately, C++ provides a way for us to override C++'s default copy mechanism (which does a shallow copy). If we provide a constructor that takes a single InventoryItem argument, C++ will use that constructor to make a copy instead of using the default memberwise copy. This special constructor is called a copy constructor.

There may be some confusion at this point about when we need an assignment operator and when we need a copy constructor, since the two situations are essentially the same. The difference is that the overloaded assignment operator is used exclusively for situations where the actual assignment operator is used in a client program. The copy constructor is used when C++ has to make a copy of an object in situations where the assignment operator is not being used.

The code for the copy constructor is going to look an awful lot like code you have seen in several places before. In fact, one way of thinking about the copy constructor is that it is exactly the same as the assignment operator except (1) you don't have to delete anything, because we are creating an object from nothing rather than modifying an existing object, (2) since you don't have to delete anything you don't have to check for self-assignments, and (3) you don't have to return anything. Removing these lines from our assignment operator results in the following code for the copy constructor. This time I've given you the correct version from the start!

```
InventoryItem::InventoryItem(const InventoryItem& right)
{
    units = right.units;
    description = new char[strlen(right.description) + 1];
    strcpy(description, right.description);
}
```

Let me reemphasize that your copy constructor does not require a delete statement! Countless hours have been spent by students trying to debug their classes when the only error was including a delete statement in their copy constructor!

When we include this copy constructor in our implementation file (and add its prototype to the class declaration) we get the expected output from the client program given at the beginning of section 3. We also get the picture that we desired (see the last picture).

There are three situations in which C++ uses the copy constructor to make a copy if one is available. The first is pass-by-value parameters, which we have just discussed. The second is the return statement. When a return statement is executed in C++, a copy of the return value is made and placed among the local variables of the calling function. The third situation is initialization. It is important to understand that C++ makes a distinction between assignment and initialization. When you initialize a variable at declaration, the assignment operator is not invoked. So the declaration statement

int x = 7;

is not considered an assignment, it is considered an initialization. It is also important to understand that there is an alternative syntax for this statement. The statement above is exactly equivalent to the statement

int x(7);

Let me illustrate by adding several lines to our client program:

```
InventoryItem item3 = item1;
// this is exactly equivalent to: InventoryItem item3(item1);
cout << "after InventoryItem item3 = item1;" << endl;
cout << "item1 is " << item1 << endl;
cout << "item3 is " << item3 << endl << endl;

item3.setInfo("ice cream", 962);
cout << "after item3.setInfo(\"ice cream\", 962);" << endl;
cout << "item1 is " << item1 << endl;
cout << "item3 is " << item3 << endl << endl;
```

The expected output of these lines of code is as follows:

```
after InventoryItem item3 = item1;
item1 is 9 hammer
item3 is 9 hammer

after item3.setInfo("ice cream", 962);
item1 is 9 hammer
item3 is 962 ice cream
```

However, if we add this code to our client program but run it without the copy constructor present, our output would be as follows:

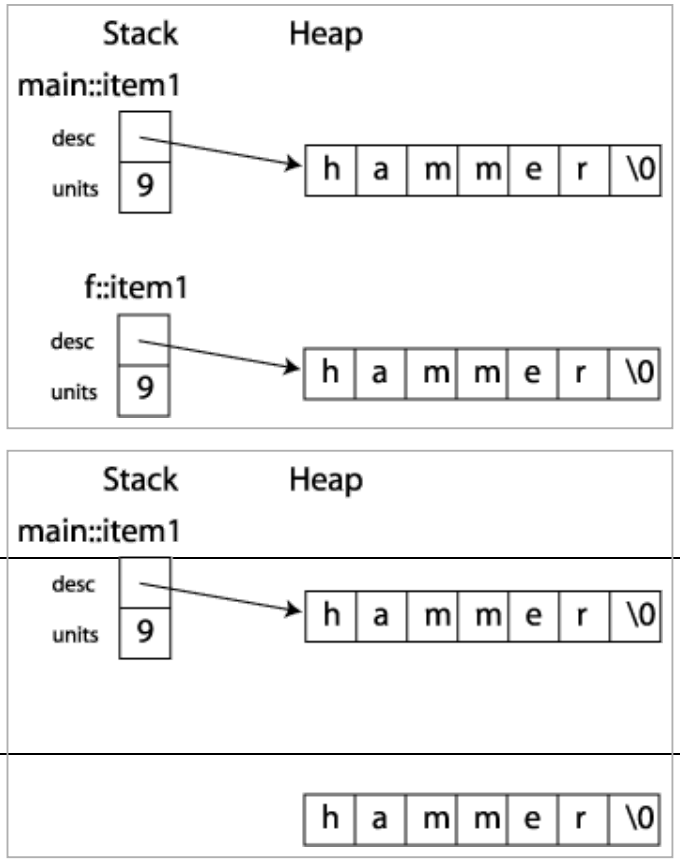
```
after InventoryItem item3 = item1;
item1 is 9 hammer
item3 is 9 hammer

after item3.setInfo("ice cream", 962);
item1 is 9 ice cream
item3 is 962 ice cream
```

This incorrect output is generated if we omit the copy constructor from our class, EVEN IF THE ASSIGNMENT OPERATOR HAS BEEN OVERLOADED. You might want to try copying the final version of our class and experimenting with this a bit to see for yourself what happens when the copy constructor is missing.

Section 17.4: The Destructor

We have now encountered 2 of the 3 member functions that must be included in any class that uses dynamic memory: the **assignment operator** and the **copy constructor**. To discover the remaining member of the big-3, let's take a closer look at what happens to the local variable item1 in the function f in the client code above. Because we have a copy constructor, memory has been allocated on the heap for the description data member of this variable. Here is a picture of the situation:

What happens to that memory that has been allocated on the heap when execution reaches the end of the function f? Recall that all local variables on the stack (the automatic variables) are deallocated at this time, so the variable item1 in function f will be properly deallocated. But the variable that item1.description was pointing



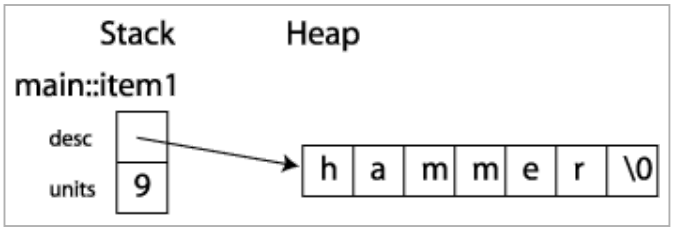
to on the heap does not get automatically deallocated, and so it is now inaccessible and we have a memory leak, as this picture illustrates:

To fix this problem, C++ provides a function called a destructor that is called automatically when an object on the stack is deallocated (this normally happens when execution reaches the end of a function). We just need to write a destructor that will deallocate any memory on the heap that the object is pointing to. Destructors must always have the same name as the class, except that a tilda (~) precedes the name. Destructors, just like constructors, do not have return types. And destructors never have arguments. Here is the destructor for our class:

```
InventoryItem::~InventoryItem()
{
    delete [] description;
}
```

Of course, including a destructor does not usually affect the output that our program produces, so it is easy to forget.

When we add this destructor to our implementation file and add its prototype to our class declaration, both the local variable item1 and the variable that it is pointing to on the heap are deallocated when execution reaches the end of the function, as in this picture:



Section 17.5: The Default Constructor

We have now encountered all three of the big-3 member functions that must be included in any class that uses dynamic memory: the **assignment operator**, the **copy constructor**, and the **destructor**. In addition, you'll need a default constructor (a constructor that takes no arguments). Here's why: suppose you declare an InventoryItem object without providing any arguments, and that no default constructor is present. For one thing, C++ will not let you do this: you will get a syntax error. But even if this were not the case there would still be a problem. When execution reaches the end of the function in which the object was declared, your destructor will be called. The destructor will execute a delete statement on a pointer that was never initialized, and you will get an unhandled exception. Since our class already has a default constructor, we have no further code to add to our class.

Section 17.6: The square brackets ([]) operator

We will now extend our class so that we can use square brackets to extract a single character of our description. So, for example, we would like to be able to include the statement

cout << item1[1] << item1[2] << item1[3] << endl;

in our client program to print out the characters of the item1 description that are in positions 1, 2, and 3. The square brackets operator is a bit unusual in that the operator is split into two parts. Instead of a single character appearing between its operands, with the square brackets operator you have the left bracket appearing between the two operands, and a right bracket appearing AFTER the right operand. So in the expression item1[1], the left operand (or calling object) is item1 and the right operand is 1.

In addition to returning the appropriate character, our square brackets operator should do some range checking. We will use an assert statement to make sure that the right-side operand (the integer index) is in the appropriate range; that is, between 0 and one less than the length of the description. Then we simply return the appropriate character in the description. Here is our function:

```
char InventoryItem::operator[](int index) const
{
    assert(index >= 0 && index < strlen(description));
    return description[index];
}
```

This code will work fine for the example statement given above, but what if we want to use the square brackets on the left side of an assignment statement? For example, what if we add to our client program the statements

```
item1[1] = 'c';
item1[2] = 'i';
item1[3] = 's';
cout << item1[1] << item1[2] << item1[3] << endl;
```

If we add these lines to our client program, we will get a syntax error such as "not an lvalue". "lvalue" stands for "left value". The reason for the error message is that in order for something to be on the left side of an assignment operator, it has to be a memory address, not a value. It doesn't make sense, for example, to say 19 = count + 1;, because the thing on the left of the assignment operator is a value, not a memory location. x = count + 1;, on the other hand, is fine, because the thing on the left side of the assignment operator is a memory address (variable, in this case).

The way to fix this problem is to have our overloaded square-brackets operator return a REFERENCE to a variable (which is actually the memory address of a variable). The way our square brackets operator is defined now, it returns a value, which cannot be an lvalue. Let's put an & after the word "char" so that the function returns a REFERENCE to (or memory address of) a variable instead of a copy of the variable. This way, the calling function will be able to use it as an lvalue, and the code above will work. Here is our current version of the function:

```
char& InventoryItem::operator[](int index) const
{
    assert(index >= 0 && index < strlen(description));
    return description[index];
}
```

We still have one more problem. Let's now add this code:

```
const InventoryItem item4("chair");
cout << item4[1];
item4[1] = 'z';
cout << item4[1] << endl;
```

If we have written our class correctly, the compiler will forbid the third of these four statements, because item4 is a const and should not be allowed to change. Unfortunately, the square brackets operator we have written will allow this code and the letter 'z' will be output, because our function returns a REFERENCE to a character. (The fact that operator[] is a const function only prevents the function itself from changing the calling object; it does not prevent the calling function in the client program from changing the object.) We need some way to make it so that if the object in the client program is a const, the operator[] function returns a value, but if the object in

the client program is not a const, it returns a reference. We will do this by providing two operator[] functions instead of just one. One version will return a value, the other a reference. We will force the C++ to call the version that returns a value when the calling object is const by making that version a const member function. The other version, which returns a reference, will not be const, so C++ will call it when the calling object is not const. Here are the two versions of the function:

```
char InventoryItem::operator[](int index) const
{
    assert(index >= 0 && index < strlen(description));
    return description[index];
}

char& InventoryItem::operator[](int index)
{
    assert(index >= 0 && index < strlen(description));
    return description[index];
}
```

Section 17.7: The Extraction Operator

We would like to be able to read InventoryItem objects from a stream using the extraction operator in the usual way. For example, we'd like to be able to include code like this in our client program:

```
cout << "enter two inventory items: ";
cin >> item1 >> item2;
cout << "you entered " << item1 << " and " << item2 << endl << endl;
```

Before we write our extraction operator we need to decide on a format in which we will expect the input to appear. Our extraction operator will skip leading whitespace, since this is the usual behavior of the extraction operator. With the exception of this leading whitespace, we will require the description of the item to be the first thing in the input stream. This description will be terminated with a colon (:). After the colon there will be an integer representing the number of units. There may or may not be whitespace after the colon before the integer. As is usual for the extraction operator, if any of these requirements are not met, the stream should go into an error state. Here is a first attempt at an extraction operator:

```
istream& operator>>(istream& in, InventoryItem& target)
{
    while (isspace(in.peek())){
        in.ignore();
    }

    in.getline(target.description, 127, ':');
    in >> target.units;
}
```

In case you need a review of the getline function: the first argument is the C-string to which you want the input to go, the second argument is a maximum number of characters to read, and the third argument is the delimiting character (so here we want to stop reading when we get to a colon).

Unfortunately, our function is fraught with problems. First, let's talk about the maximum number of characters. Why don't we write the function so that there is no maximum number of characters? It turns out that this would be very difficult and very inefficient. Since we can't know how many characters there are until after we read them all, we can't create our array ahead of time. The only way to make the size of the descriptions we read unlimited would be to read one character at a time, deleting and reallocating our array each time a the array becomes full. Instead, we will place a limit on the size of the descriptions we read. We will arbitrarily set a limit of 127. Note that this does NOT mean that descriptions can NEVER have more than 127 characters. We could, for

example, use a CONSTRUCTOR to create a description with more than 127 characters. We just can't READ descriptions that are longer than 127 characters.

The second problem is much more serious. At the beginning of this function the parameter target has some value. It may be an InventoryItem with a very short description or a very long description; we don't know. If, for example, the description that we read is longer than target's description, we are likely to get an unhandled exception when we execute the getline function, because there is not enough space in target.description to store the description that appears in the input.

To fix this problem we need to delete target.description and reallocate it. But the next problem is: how big should we make it when we reallocate it? One solution would be to allocate enough memory to hold a 127 character description. This, however, would be extremely inefficient. We don't want to allocate a 128 element array to hold the description "hammer" which has only 6 letters! Our solution will be as follows (compare these steps to the code shown below): (1) Declare a temporary C-string variable with a size of 128 (one extra character for the terminating null ('\0') character); (2) Read the description into this temporary variable; (3) Delete target.description; (4) reallocate memory for target.description based on the size of the string stored in this temporary variable; (5) Copy the temporary variable into target.description.

In order to accomplish this, we'll also add a constant to the class to store the maximum input size of 127. Constants that are members of classes must be preceded by the word "static," which makes it so that only one copy of the constant is stored and shared by all objects of the class, instead of having a separate copy of the constant for each object of the class. So, our constant declaration will look like this:

```
static const int MAX_INPUT_SIZE = 127;
```

The third problem with our original code is that we need to have a return value. As always, the extraction operator returns its left operand, which in our case is in. Here is the final version of our extraction operator:

```
istream& operator>>(istream& in, InventoryItem& target)
{
    while (isspace(in.peek())){
        in.ignore();
    }

    char temp[InventoryItem::MAX_INPUT_SIZE + 1];
    in.getline(temp, InventoryItem::MAX_INPUT_SIZE, ':');
    delete [] target.description;
    target.description = new char[strlen(temp) + 1];
    strcpy(target.description, temp);
    in >> target.units;

    return in;
}
```

Section 17.8: The Complete InventoryItem Code

```
// This is the file "invitemtest.cpp"

#include <iostream>
#include "invitem.h"
using namespace std;

void f(InventoryItem item1);

int main()
{
    InventoryItem item1;
    InventoryItem item2("hammer");

    cout << "item1 is " << item1 << endl;
    cout << "item2 is " << item2 << endl;

    item1.setInfo("screwdriver", 5);
    item2.setUnits(9);

    cout << "item1 is " << item1 << endl;
```

```
cout << "item2 is " << item2 << endl << endl;

// There are 4 issues that come up when you use
// a pointer as a data member.  You must include
// in your class:
// (1) ASSIGNMENT OPERATOR
// (2) COPY CONSTRUCTOR
// (3) DESTRUCTOR
// (4) DEFAULT CONSTRUCTOR



// (1) ASSIGNMENT OPERATOR

// If you don't have an overloaded assignment
// operator, the next 8 lines will give you
// incorrect results.  Try it!

item1 = item2;
cout << "after item1 = item2, " << endl;
cout << "item1 is " << item1 << endl;
cout << "item2 is " << item2 << endl << endl;

item2.setInfo("lawn mower", 14);
cout << "after item2.setInfo(\"lawn mower\", 14), " << endl;
cout << "item1 is " << item1 << endl;
cout << "item2 is " << item2 << endl << endl;



// The next 3 lines illustrate the need for the
// if statement in the assignment operator.  Take
// the if statement out and see what happens.

item1 = item1;
cout << "after item1 = item1, ";
cout << "item1 is " << item1 << endl << endl;




// (2) COPY CONSTRUCTOR

// There are three situations in which C++ makes a copy:
// 1) pass-by-value, 2) return, and 3) initialization.
// You can supply a copy constructor to override
// C++'s default copy operation.


// First pass-by-value.  If you take out the copy constructor,
// the call to f will change the value of item1, even though
// pass-by-value was used.

f(item1);
cout << "after being used as an argument to pass-by-value parameter, ";
cout << endl << "item1 is still " << item1 << endl << endl;


// Now initialization.  C++ makes a distinction between
// assignment and initialization.  Therefore, if you take
// out the copy constructor, the next 8 lines will give
// incorrect results, even if = is overloaded.

InventoryItem item3 = item1;
// this is exactly equivalent to: InventoryItem item3(item1);

cout << "after InventoryItem item3 = item1;" << endl;
cout << "item1 is " << item1 << endl;
cout << "item3 is " << item3 << endl << endl;

item3.setInfo("ice cream", 962);
cout << "after item3.setInfo(\"ice cream\", 962);" << endl;
cout << "item1 is " << item1 << endl;
cout << "item3 is " << item3 << endl << endl;




// OVERLOADING THE SQUARE BRACKETS

cout << item1[1] << item1[2] << item1[3] << endl;
item1[1] = 'c';
item1[2] = 'i';
item1[3] = 's';
```

```
    cout << item1[1] << item1[2] << item1[3] << endl;
    const InventoryItem item4("chair");
    cout << item4[0] << endl;

    /* should cause syntax error:

    item4[1] = 'z';
    cout << item4[1] << endl;

    */




    // OVERLOADING THE EXTRACTION OPERATOR

    cout << "enter two inventory items: ";
    cin >> item1 >> item2;
    cout << "you entered " << item1 << " and " << item2 << endl << endl;

}

void f(InventoryItem item1)
{
    item1.setInfo("pizza", 67);
}
```

```
// This is the output:

item1 is 0
item2 is 0 hammer
item1 is 5 screwdriver
item2 is 9 hammer

after item1 = item2,
item1 is 9 hammer
item2 is 9 hammer

after item2.setInfo("lawn mower", 14),
item1 is 9 hammer
item2 is 14 lawn mower

after item1 = item1, item1 is 9 hammer
after being used as an argument to pass-by-value parameter,
item1 is still 9 hammer

after InventoryItem item3 = item1;
item1 is 9 hammer
item3 is 9 hammer

after item3.setInfo("ice cream", 962);
item1 is 9 hammer
item3 is 962 ice cream
```

```
// This is the file "invitem.h"

#ifndef INVITEM_H
#define INVITEM_H
#include <iostream>

class InventoryItem {
    public:
            static const int MAX_INPUT_SIZE = 127;
        InventoryItem();
        InventoryItem(const char *inDesc);
        InventoryItem(const InventoryItem& right);
        ~InventoryItem();
        InventoryItem operator=(const InventoryItem& right);
        void setInfo(const char *inDesc, int inUnits);
        void setUnits(int inUnits);
        friend std::ostream& operator<<(std::ostream& out, const InventoryItem& source);
        friend std::istream& operator>>(std::istream& in, InventoryItem& target);
        char operator[](int index) const;
        char& operator[](int index);
    private:
        char *description;
        int units;
};

#endif
```

```
// This is the file "invitem.cpp"

#include <iostream>
#include <cassert>
#include <cstring>
#include "invitem.h"
using namespace std;

InventoryItem::InventoryItem()
{
    units = 0;
    description = new char[1];
    strcpy(description, "");
}

InventoryItem::InventoryItem(const char *inDesc)
{
    units = 0;
    description = new char[strlen(inDesc) + 1];
    strcpy(description, inDesc);
}

InventoryItem::InventoryItem(const InventoryItem& right)
{
    units = right.units;
    description = new char[strlen(right.description) + 1];
    strcpy(description, right.description);
}

InventoryItem::~InventoryItem()
{
    delete [] description;
}

InventoryItem InventoryItem::operator=(const InventoryItem& right)
{
    if (this != &right){
        units = right.units;
        delete [] description;
        description = new char[strlen(right.description) + 1];
        strcpy(description, right.description);
    }
    return *this;
}

void InventoryItem::setInfo(const char *inDesc, int inUnits)
{
    units = inUnits;
    delete [] description;
    description = new char[strlen(inDesc) + 1];
    strcpy(description, inDesc);
}

void InventoryItem::setUnits(int inUnits)
{
    units = inUnits;
}

ostream& operator<<(ostream& out, const InventoryItem& source)
{
    out << source.units << " " << source.description;
    return out;
}

istream& operator>>(istream& in, InventoryItem& target)
{
    while (isspace(in.peek())){
        in.ignore();
    }

    char temp[InventoryItem::MAX_INPUT_SIZE + 1];
    in.getline(temp, InventoryItem::MAX_INPUT_SIZE, ':');
    delete [] target.description;
    target.description = new char[strlen(temp) + 1];
    strcpy(target.description, temp);
    in >> target.units;

    return in;
}

char InventoryItem::operator[](int index) const
{
    assert(index >= 0 && index < strlen(description));
    return description[index];
}

char& InventoryItem::operator[](int index)
```

```
    {
        assert(index >= 0 && index < strlen(description));
        return description[index];
    }
```