CS 11 Data Structures and Algorithms

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);
}
```

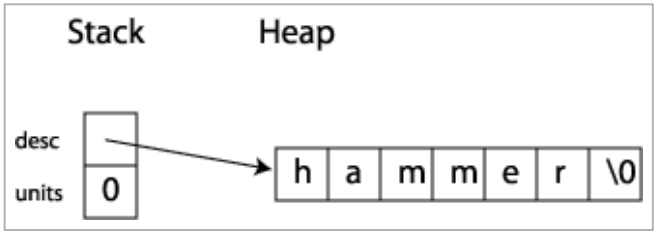
It is very important that you understand what is wrong with this code. It is very likely that if I ran the program using this code I would get an unhandled exception. Why? Because I have tried to copy information into a C-string pointed to by description, but I have not yet allocated any memory for it! 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).

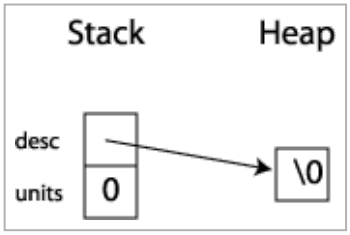
```
InventoryItem::InventoryItem(const char *inDesc)
{
    units = 0;
    description = new char[strlen(inDesc) + 1];
```

```
    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:

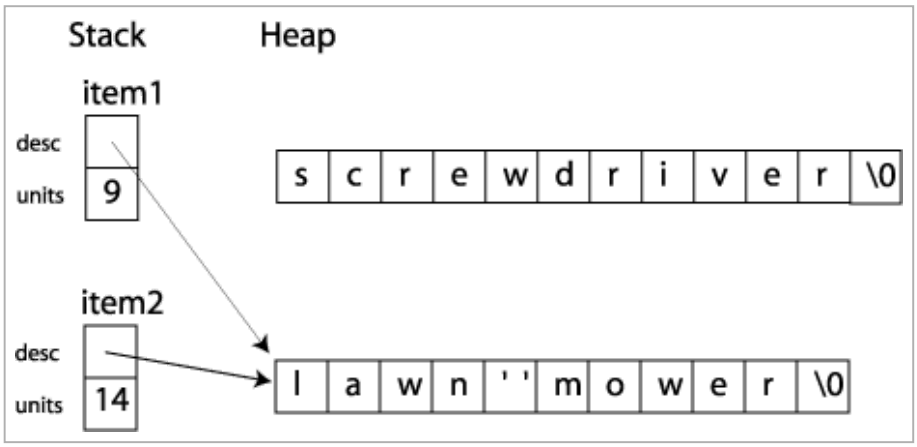
```
Actual Output:

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)
```