Pointers

This tutorial is my attempt at clarifying pointers for anyone still confused about them. Pointers are notoriously hard to grasp, so I thought I'd take a shot at explaining them. The more information out there on pointers, the better.

I'll be giving code demonstrations thoughout this tutorial and will also include a downloadable program that you can use to test different aspects of pointers. I'm not sure how much can be gained by copying out the code here line for line, but I do encourage you to play around with the code.

First, let's briefly talk about how memory is stored in a computer. You can think of memory as sequential rows of storage spaces, each with its own address. When you declare variables, your program stores the information in one of those spaces (or a group of spaces if it needs the extra room) and remembers where it is. If you have declared a variable and you want to know where your program put it, you can use the address-of operator to get the address.

```
int age = 21;                       // declare the variable on the
stack
cout << "age: " << age << endl;     // print the value stored in age
cout << "&age: " << &age << endl;   // print the location of the value
```

The Stack, Free Store, and Global Namespace

Now let's talk about the difference between the stack, the global namespace, and the free store.

The stack is part of the area in memory that is reserved for your program. Any local variables that you declare on the stack (ie. normal variable declarations like "int age;") stay there until your program is finished, at which point that area in memory will automatically be cleaned up. If you declare a variable in a function, loop, or conditional statement to be on the stack, it will only exist until the end of that statement. Afterwards, that part of the stack will be cleared and anything declared there will be gone.

Global variables exist in the global namespace and the memory they take up is automatically freed at the end of your program. They act just like variables declared on the stack with the exception that they can be accessed from anywhere in your program.

The memory in your computer that is not being taken up by the stack or the global namespace is called the free store (a.k.a. "the heap"). This area of memory is pretty much fair game for any program that wants to use it. Unlike the stack and the global namespace, variables declared on the freestore exist until you explicitly delete them. If you don't delete them by the end of your program, they will just sit there until you restart your computer. This situation is most commonly called a memory leak and is usually very difficult to detect because there's no easy way to see what is stored on the free store.

Before we discuss why we would want to store data on the free store, let's look at how you would declare a variable there. This is also a good time to download the source code that comes with this tutorial, read the comments, and see what you get when you run the program.

Here's some code that declares a value on the freestore:

```
// a variable has been declared but set to NULL, nothing has been put
on the freestore yet
int* width = NULL;
// now something is on the freestore (30), and width holds the
address that points to it
width = new int(30);
```

Don't worry, we're going to get to why we would do this in a minute. First, I want you to really understand what goes on when we declare a pointer. The variable width does not represent the value 30, it just stores the location in memory where 30 exists. If we had another pointer to that location in memory and changed the value there, width would then point to that new value. This is because width doesn't represent the value 30, it just points to whatever is sitting at that location on the free store.

Once again **please** check out the downloadable source for this tutorial.

Let's now look at some of the operations we can perform on pointers and then get to why we would use them.

```
// Assume that width has already been declared as a pointer
// This prints out whatever is stored in the width variable. In this
case, it's the location
// on the free store that the pointer points to, not the value it
points to
cout << "\nwidth: " << width << endl;

// This prints out the address of the variable width. This is just
like what we did with age
```

```
// above. It's the address of the variable, not the value at the
variable (which is also an
// address). The address we get from this is the address of the
location on the stack where
// we store the address to the value stored on the free store.
cout << "&width: " << &width << endl;

// The indirection operator (*) is used to print out the value a
pointer points to. In this
// case it points to the value 30.
cout << "*width: " << *width << endl;

// Now that we're done using our pointer, we MUST delete it!
delete width;
```

Using Pointers

Alright, so why use pointers? Remember when we talked about the stack and how local variables only exist where they're declared? For example, if we declare a variable on the stack within a function, it will only exist until the end of that. But what if we want our data to persist? This happens for all kinds of reasons. We might have some data representing the location of an enemy in a game. We'll need that data to be available in all kinds of functions (collision detection, rendering, AI processing, etc). So let's look at ways we can do this.

The first way would be to just declare the enemy data in a function and start passing it to other functions. It'll be a bit of a pain because all of our functions will need an extra parameter, but there's an even worse problem with this method. When you declare a variable on the stack in a function, it only lasts as long as that function. If you try passing that data to another function, the new function can't just use the same place on the stack where the data is stored because it could be erased at any time. Thus, the new function has to make its own room on the stack and copy the data. This process is called "passing by value".

Passing by Reference vs. Passing by Value

This brings me to the difference between passing by reference and passing by value. When you pass by value, an entirely new area in memory is created by the function you've passed the data to and the information is copied over. This is fine for something like an integer, but what about a data structure that consists of tons of data? You'll really slow your program down if it has to constantly copy large amounts of data.

Before I talk about passing by reference, let's look at the global variable solution to our problem. With global variables we can have all of our data stored in one place and it can be accessed from anywhere. So what's the problem with this approach?

Well, the problem with global variables is that we can have all of our data stored in one place and it can be accessed from anywhere (yes, I did just repeat myself). This is great for smaller programs. You don't have to create extra parameters in your functions just so you can access data from different parts of your program, and there's no penalty like with passing by value.

The problem comes when you try to write large scale applications. Think of the complexity of most games now. They have phyics engines, graphics engines, sound, input, networking...way too much stuff to be able to keep track of with a mess of global variables. Especially when you have a team of 30 people working on a project. Instead of expecting everyone to keep track of every variable in the program, a different method must be used.

So now we come to the pointer solution and the definition of passing by reference. When we pass by reference, we only pass the address of the data (the data is located on the free store) to the function that we want to access the data. The address is of fixed length (whatever the size of a pointer is on your machine), so we aren't slowing our program down by copying large amounts of data. We also don't need to worry about the data not being there because we are the ones who control its deletion. Note that we can still cause ourselves problems by deleting something on the free store and then trying to access it from somewhere else.

Let's now talk about reference variables and follow with example of passing by reference and passing by value. Reference variables are not pointer variables, but they are very similar. Both reference variables and pointer variables store the address of data on the free store. They differ in how we use them.

To declare a reference variable, you simply add a & after the variable like so:

```
int& ref;
```

Reference variables are good for passing the address of a value to a function. If you're wondering why having the function take a pointer isn't just as good, see the following code.

Here's a listing of three functions, one that takes the value of a variable, one that takes a pointer to a variable, and one that takes the address of the variable:

```
// Just takes the values of the parameters passed to it
// and copies them into the variables x and y
int AddByValue(int x, int y)
{
     return x + y;
}

// Takes pointers to the parameters passed to it
int AddByReference1(int* x, int* y)
{
    // You have to use the indirection operator because they're
pointers
    return (*x) + (*y);
}

// Takes the values by reference
int AddByReference2(int& x, int& y)
{
    return x + y; // ah, that's nice!
}

// Here's how you would call each function
int x = 4;
int y = 7;

// This one's pretty straight forward, just call it and
// it'll copy the values you pass to it
AddByValue(x, y);

// We have to send the addresses because the function takes pointers.
// Remember, a pointer just stores the address of the data, if we
// were allowed to just send in x and y (without the & operator) we'd

// be telling the variable x to point to the non-existent address 4
// and the variable y to point to the non-existent address 7
AddByReference1(&x, &y);

// Aren't reference parameters great? We just send in the variables
// and it's taken care of for us
AddByReference2(x, y);
```

As you can see, using reference variables as parameters allows us to use cleaner code than would be possible using pointers as parameters. When we used pointers, we had to pass the function parameters using the address-of operator. We also had to use the indirection operator when we added the two values (we had to use the indirection operator to get the actual value stored at the location the pointers pointed to, otherwise

we would have added their addresses. Try it out and see what you get!).

When we had the function take reference variables, everything was taken care of for us. So the moral of the story is, when you want to pass by reference, use reference variables as function parameters.

Conclusion

That concludes our discussion of pointers. Please let me know if you think I could make this any better. And be sure to download the source and play with it if you haven't already.

One final note: When people explain pointers they often make it sound like pointers are the only thing you should ever use. They do this because they are trying to convince you of their importance. However, don't think that you should only ever use pointers.

Click here to download the source code for this tutorial.