A Prelude to Pointers

Nitron

23 Jul 2003

Rate me: 4.42/5 (49 votes)

A quick reference to pointers and pointer operations

Download demo project - 7.8 KB

Introduction

Pointers are undoubtedly a daunting topic of discussion for people just starting out in the C++ lifestyle, which prompted me to write this brief article. Note that this is not a full-featured discussion on pointers, but a quick reference to commonly asked questions about their use and what they really mean. This is also a nice primer to Andrew Peace's[^] article: A Beginner's Guide to Pointers[^], where he presents an excellent discussion on the topic.

Philosophy

Like I previously stated, my intent is not to give the origin of the pointer universe, but to provide a brief example of how pointers are used, and in the process hopefully clear up some lingering questions posted in the forums and similar questions posted in response to the aforementioned article. Therefore, the philosophy behind this article is to demystify pointers by using them in a simple working example.

Terminology and Technique

When working with pointers, I employ the following terminology and techniques:

- **Value** - The actual value of the specified data type
- **Pointer To** - A pointer to the memory location of a specified data type
- **Address Of** - The physical memory address of a given data type
- **Reference To** - A reference to a given data type, NOT a copy of it (C++ only)
- **Dereferenced Value** - The value of the data type pointed to by a pointer

The subject of pointers deals heavily with the use of the symbols ***** and **&**, and to a beginner, this is often confusing; especially when different people put the modifers in different places (me and the VS6 class wizard for example :P). My technique is to keep the modifiers next to what they are modifying. For example:

Hide Shrink Copy Code

```
// Initialize a variable, 'x', of type 'int' to the 'value' 5.
int x = 5;

// Initialize a variable, 'px' of type 'pointer to int' to the value of
// 'the address of' x.
int* px = &x;
----      --
|         |
|         |- This is read "address of" x
```

```
|
|- Notice I put the * modifier after int, not before px (int* px not
   int *px).  The compiler doesn't care, however I am modifying the
   data type, not the variable. Thus I remain consistent to my technique.

// Now initialize a variable 'ref_to_x' as a reference to x.
int& ref_to_x = x;
----
|
|- Again, I keep the modifier with what I am modifying.

// Initialize a variable,'deref_px' of type 'int' to the 'dereferenced
// value' of px.
int deref_px = *px;
               ---
               |
               |- Notice in this case, the * is with px, because that is
                  what I'm modifying.
```

Ok, with that out of the way, we continue on.

Passing By Reference

There have been many a question floating around about: *passing values by reference*. Why would you pass by reference? Passing a value by reference speeds things up and slims things down, because you are not making a copy of the variable passed. This may not seem too important when making a single function call in the whole life of your application's instance, but when calling a function say 1800 times in a minute; performance becomes a major motivating factor. Also, when modifying a copy of a value, the original value remains unchanged (which may be desirable in some cases), where when passing by reference, the original variable is modified. The downloadable example program will illustrate why, but here is a brief example that should clear up the confusion:

Consider the following functions:

Hide Copy Code

```
void MyFunctionByCopy(CString x)
{
   CString y = "Test by copy.";
   x.Format("%s",static_cast<const char*>(y));

   return;
}

void MyFunctionByReference(CString& x)
{
   CString y = "Test by reference.";
   x.Format("%s",static_cast<const char*>(y));

   return;
}
```

Note: The first one will create a copy of the variable passed, where the latter one will modify the variable itself. Note that the reference to int is an automatically dereferenced pointer, thus pointer notation is not required. Check the sample program for a working example of this in action.

Given the two functions above, can you guess the values of the results in the following example?

Hide Copy Code

```
{
    CString szTestOne = "Test Me One";
    CString szTestTwo = "Test Me Two";
```

```
    MyFunctionByCopy(szTestOne);
    MyFunctionByReference(szTestTwo);

    printf("\nszTestOne = %s", static_cast<const char*>(szTestOne));
    printf("\nszTestTwo = %s", static_cast<const char*>(szTestTwo));
}
```

For those who can't handle the suspense, `szTestOne` = "Test Me One" and `szTestTwo` = "Test by reference.". If you don't believe me, try it.

Pointer to a Pointer

Another lingering concern is the *pointer to a pointer*. If you subscribe to my technique, a pointer is nothing more than an arbitrary data type, and thus a 'pointer to a pointer' is nothing more than a plain old pointer. Sure, it may appear daunting and somewhat foreign, but break it down to its bare essentials and there's really nothing to it. So in fact, you can also have 'pointers to pointers to pointers' and 'pointers to pointers to pointers to poin...' well, you get the idea.

For anyone interested in what such an implementation may look like, consider the lines below. You may try this on your own and see the results.

Hide Copy Code

```
int   x   = 7;
int*  px  = &x;
int** ppx = &px;

printf("\nx     = %d", x    );
printf("\n*px   = %d", *px  );
printf("\n**ppx = %d", **ppx);
```

Why Use Pointers at All?

I can remember when I first came across pointers when beginning C++, and after much confusion, I wondered: "why use pointers at all?" I can just instantiate explicit instances of all my variables and not worry about it. (Notice I was all fancy using words like *instantiate* and *explicit instance*, yea I knew C++... right...) Well, with experience comes humility and wisdom; and with CodeProject comes enlightenment and encouragement. Venturing off into applications beyond the command line scripts (I say scripts, because my entire program was inside `main()` and I made no function calls. My idea of code reuse was cut-and-paste) I was coding before, I found myself in a whole new world.

As it turns out, in real-world programming, you don't always have the privilege of knowing how many things you will be analyzing, or how big your arrays need to be. You don't know how much system resources your user's machine will have. You begin to worry about things like *optimization* and *passing by reference*. You start using built-in functions that take pointers by default, although you may not know why...

In my personal experience (all two years of it), the primary driving forces behind my exodus into the realm of pointers are the wonderful operators:

Hide Copy Code

```
new and delete
```

Although this is not a comprehensive guide to pointers, I feel you cannot talk about pointers without mentioning the aforementioned operators. Why are these operators so important? They are important not only because they allow you to dynamically allocate memory to and from your program, but they do it through the use of pointers. There is much information available on this topic so I will not go into vast detail, but I will cover the basic highlights. Again consider the following code:

Hide Copy Code

```
int* pNewInt = new int;
*pNewInt = 7;
printf("\nThe pointer, pNewInt, is located at memory address: 0x%X",
       &pNewInt);
printf("\npNewInt points to memory location 0x%X and contains the
       value %d.",
       pNewInt, *pNewInt);
delete pNewInt;
```

In this code, we created a new 'pointer to `int`' with the keyword `new`. Note that new returns a 'pointer to `int`'. An important thing to remember is to delete your new pointers after you are done with them. If you don't, you will have many a memory leak and risk utilizing all resources of the target machine. The details of this can be found elsewhere, but the point remains: new is a powerful friend and is worth the time to understand.

In Conclusion

Pointers may seem intimidating at first, but they are in fact a fundamental concept in C and C++. Reading about them will only take you so far; it is the experience in using them and the enlightenment of harnessing their power that will ultimately contribute to making the world a better place. So enjoy your pointers. Use them often, and take good care of them. But most importantly, use them with courage and confidence and you will be greatly rewarded when you *pass beyond that great exception in the sky*...(**Mark Conger - Death of a Coffee Pot**)