More Than You Ever Wanted to Know About Pointers

Kyle Lemons

October 30, 2006

Contents

1 Introduction **2**

2 The Basics **2**
2.1 What Is a Pointer? 2
2.2 Why Use a Pointer? 3
2.3 Compiler Speak: Syntax 4

3 Using Pointers **6**
3.1 What's Behind Door #3: Dereferencing a Pointer 6
3.2 Where are you? The Address-Of Operator 7
3.3 I Swear It's Really A Pointer: Arrays 8
3.4 Fun With Strings (C-style) 8
3.5 A Walk in the Park: Traversing Arrays and Strings 8

4 Advanced Usage **8**
4.1 Here We Go Again: Arrays of Strings 8
4.2 Kill Me Now: Multidimensional Arrays 8
4.3 Pointers to Nowhere: Handles 8
4.4 Back to Kindergarten: Pointer Arithmetic 8

5 Memory Management **8**
5.1 The usual suspects: `malloc` and `free` 8
5.2 Array Management: `calloc`, `realloc`, and the `mem*` Family . . . 8

1 Introduction

This article is written at the request of the Fall 2006 CS1372 class at the Georgia Institute of Technology. The purpose of this document is to supplement the readings and to assist in the understanding of pointers. The motivation behind it is that pointers are such an integral part of and a large component of the power behind programming in C and C++ that not understanding them would cripple the student in the course and impede their effectiveness as a programmer. This document will cover pointers from the very most basic concepts up to the most advanced usage that the author can come up with, with the hope that the reader will work slowly through it to understand each concept before moving on to more difficult ones.

All code examples contained in this document should work in any standards-compliant C or C++ compiler. At the moment, the author does not intend to cover the `new` and `delete` C++ operators, so most of the text should apply equally to C and C++.

2 The Basics

2.1 What Is a Pointer?

At its most basic, a pointer is just a number. In fact, pointers are really just `unsigned int`egers. In most operating systems (32-bit) a pointer is 32 bits wide. Not coincidentally, this is the same size as an `unsigned int`eger. In fact, as you can see in the example below[1], you can print out a pointer using `printf` and it can look exactly like an integer. Note, however, that there are also special format specifiers which can print out pointers in a more useful manner (`%p`, `%x` and `%X`). Granted, they are very large integers, but they are integers just the same. Understand that this is a contrived example and is not very useful, so don't go using `char`acter pointers for math instead of `int`egers..

```
char *a = "This is a test,";
char *b = "This is only a test.";
printf("a = %d,\tb = %d\n", a, b); // See, they're only numbers.
printf("a is %p,\tb is 0x%X\n", a, b); // Useful for debugging!
// %p - Print as a pointer. Automatically includes 0x
// %x - print in hexadecimal, %X capitalizes the hex letters
/* Example Output:
  a = 4196956,    b = 4196972
  a is 0x400a5c,  b is 0x400A6C
*/
```

Now that you know what a pointer is, you need to know what a pointer means. You can get some idea of this by looking at the word "pointer." A pointer "points to" someting in memory. A pointer is a number that represents

[1] The syntax will be covered later. Assume for now that a and b are pointers to a character.

where in your computer's memory where the aforementioned something resides. This number called an address. Within memory, each byte (set of 8 bits) has a unique address. This memory address can be thought of like your Georgia Tech P.O. Box number. How do you find your post office box? First, assuming the first two numbers are 33, you ignore them. Then you take the next number and you look for the hall of P.O. boxes with that number. The next number represents which block of P.O. boxes your box is in, and the last two numbers represent which row and column, respectively. In the same way, memory addresses represent an exact place in memory. It is not necessary to understand exactly how memory is organized, but if you look at the above example, you will notice that both of the addresses (in hex) have almost identical addresses. The only difference is the second to last character. In a 32-bit operating system, pointers are 32 bits wide[2]. This is also the width of the stack (the thing that passes data back and forth between function calls). This is not a coincidence, and one reason that it is done that way is to facilitate easy transfer of pointers between functions. See section 4.3 on handles to see how you can prove to yourself that pointers are, indeed, 32 bits wide.

The take-home message here is simple. A pointer is a number. Almost every error that is made with pointers stems from the programmer forgetting that their pointer is actually a number. If you want your pointer to be anything other than a number (say, a character, a string, an array) you have to treat it differently.

2.2 Why Use a Pointer?

As was mentioned in section 2.1, a pointer is a number. What good does this do us? Well, let's take a fairly common example in computer programming. Let's say that you are writing a program to analyze the data that your biology professor collected in her latest research experiment. You have upwards of ten thousand data points, each one with double floating-point precision. That could be more than 80 MB of data, which you read into memory so that you can manipulate it more easily. If you have different routines for performing operations on the data, for instance to sort it, find a good polynomial fit, do error analysis, and to rewrite the data in a new format, you will want to get the data into that routine somehow. If you recall, when data gets passed to functions in C and C++[3], it is passed by value. This means that a *copy* of the data is pushed onto the stack every time you need to pass that data into a function. I'm sure you see that you could quickly use up your memory if you are pushing 80MB of data all over the place on the stack. This is where pointers come in. In C and C++, an array is really a pointer (see section 3.3), so if you store the data in an array and pass a *pointer* to the data into the function, only

[2]This document will assume that you are using a 32-bit operating system. a 64-bit operating system will, obviously, have 64-bit pointers. If you are using such an operating system with a compiler which can compile natively for it, substitute 64-bits where appropriate.

[3]Actually, C++ can pass arguments by reference, but this is beyond the scope of this article.

32 bits must be pushed onto the stack.

2.3 Compiler Speak: Syntax

Declaring a pointer is easy. Nothing, however, can substitute for examples.

1. `char *a; //` creates a pointer. This pointer is designed to point to data of the `char`acter type.
2. `int *b; //` creates a pointer. This pointer is designed to point to data of the `int`eger type.
3. `double *data; //` creates a pointer. This pointer is designed to point to data of the `double`-precision floating-point type.
4. `const char *c; //` creates a pointer. This pointer is designed to point to data of the `char`acter type, but the data may not be modified. `const` is short for "constant"
5. `const char *d = "test"; //` creates a pointer. This pointer points to data of the `char`acter type, the data may not be modified, and the memory is already allocated.
    - Whenever you use a pointer to string constant (a double quoted string), be aware that the string is in read only memory. If you want to avoid program crashes (called segmentation faults) caused by attempting to modify these strings, always use a `const char *` with them.
    - See section 3.4 for a more detailed explanation on strings.
6. `a = d; //` Error
    - Notice that in this example, the programmer is assigning the value of a `const char *` to a `char *`. This is not allowed.
    - Modifying the contents of the memory at `d` is not allowed, because `d` is declared `const`. Storing this value into a non-`const` pointer would allow the data to be modified.
    - Notice, however, that it is permissable to assign a non-`const` pointer value to a `const` pointer. This would still not allow the contents of the memory location pointed to by the `const` pointer (via the `const` pointer).
7. `b = d; //` Because pointers are, at the very lowest level, numbers, `b` and `d` now hold the same address. This means that they both point to the same data.
8. `char e[] = "Everybody hates exceptions"; //` Creates a pointer to an array of `char`acters. The memory for this array is allocated automatically.

- Note that when you assign a string constant (double quoted string) to an array, the value of the string is copied into the array, so it can be modified (unlike constant strings pointed to by a `char *`).
- Note also that this is still a `char *`, and can be used as one in pointer arithmetic, assignment statements, and function calls.
- See section 3.4 for a more detailed explanation on string arrays.

9. `a = e; //` As is the case with "`b = d;`" above, a now points to the same string as e. However, it is permissable to modify the contents of `e` through `a`.

10. `char f = *d; //` `f` is a `char`acter, and in this example, will contain the value `'t'`.

    - Notice the different usage of the asterisk (`*`) in this example.
    - `*d` does not create a pointer in this case, it *dereferences* one. This takes the value of the data *pointed to* by d, which is a character.
    - See section 3.1 for a more detailed explanation of dereferencing pointers.

11. `char *g = &f; //` `g` is a pointer which holds the *address of* `f`.

    - See section 3.2 for a more detailed explanation of the address-of operator.

12. `double vector[10]; //` `vector` is a pointer to a memory block big enough to hold 10 `double`-precision floating-point numbers

13. `double matrix[10][10]; //` `matrix` is a pointer to a memory block big enough to hold 100 `double`-precision floating-point numbers

    - The above comment states that it points to a block of memory large enough to hold 100 `double`-precision floating-point numbers. This may not actually be the case. Strictly soeaking, `matrix` points to a contiguous block of pointers to contiguous blocks of 10 `double`s, and this is exactly how it is treated. Because the compiler knows all of the dimensions of this matrix, it can optimize by doing what the comment suggests, by turning this into a single-dimension array and computing the indices based on the known dimensions. Most new compilers will do this for you, but it is not wise to write your code to make this assumption.
    - For a more in-depth discussion of multidimensional arrays, see the advanced pointer usage in section 4

3 Using Pointers

3.1 What's Behind Door #3: Dereferencing a Pointer

Creating pointers is important, and understanding the different kinds of pointers is essential, but quite possibly the most important thing to understand about pointers is how to access the data that they point to. This is called *dereferencing* the pointer, and is denoted in C and C++ by prefixing the name of the pointer with an asterisk. The pointer dereference operator is often read "the value at," or "the value pointed to by." When you dereference a pointer, the compiler looks at the number that is stored in the pointer, finds that location in memory, and uses what it finds there. Recall that all pointers are the same size. Even though a `char` is one byte and a `double` is 8, both kinds of pointers are the same size. The difference between the two is only ever known to the compiler (not to the computer when the application is running). The two pointers themselves are identical in memory. When you dereference a `char *`, the result is a `char`. When you dereference an `int *`, the result is an `int`. When you dereference a `double *`, the result is a `double`. It is concievable that there could be one pointer of each of these types which all point to the exact same memory address. When dereferencing each of these pointers, the memory itself will be interpreted as the data type in question, no casting will occur.

As an example, let us conside an `int *` which points to the same memory address as a `char *`. You dereference the character pointer and set the value to ASCII `'A'`. You then print out the dereferenced `int *`. The printed value will not be the same as it would be if you printed out the value of ASCII `'A'` casted to an `int`, but will be the value of the 4-byte quantity pointed to by the `int *`. Only one of these bytes will have been changed with the assignment to the `char *`. The code for this example is included below. See section 3.2 for the meaning of `&victim`.

```
int victim = 65535;
printf("victim = %d\n",      victim);
int  *p =          &victim; // Pointer to victim
char *q = (char *) &victim; // Pointer to victim
*q = 'A'; // assign the byte at q to 'A' (97)
printf("*p     = %d\t(%p)\n", *p, p);
printf("*q     = %c\t(%p)\n", *q, q);
/* Example output:
  victim = 65535
  *p     = 65345  (0x7fffc6ba26d4)
  *q     = A      (0x7fffc6ba26d4)
*/
```

It is possible to create pointers to pointers, which when dereferenced result in a pointer, which can then be dereferenced again to get the data. This technique is used to create multidemsnional arrays and to create handles, which are discussed later in sections 4.2 and 4.3 respectively.

3.2 Where are you? The Address-Of Operator

The final key syntactical and strategical point in pointers is how to make a given pointer point to a certain memory location. In the above examples, you have seen how to define pointers, how to assign to pointers, and have even seen a few examples of pointers to strings. The address-of operator is the ampersand (&), is used to prefix the variable whose address is needed, and is read "address of." The address-of operator prefixes a variable name and the result is the memory address of the data that that variable is storing. The address-of operator can be used with any variable, and the data type of the result is a pointer to the data type of the variable in question. For instance, if there is a `double width = 48.25;`, then `&width` would be a `double *`, and can be assigned to a variable of that type. As you saw in the example in 3.1, it is possible to cast from one pointer type to another.

Sometimes, the programmer needs to use a variable as a pointer to a different type. As stated above, the address-of operator always results in a pointer to the same type of data as the variable in question. For circumstances like this, just like any other variable, a pointer can be casted. Pointer casting makes sense in general because all pointers are the same size, and all they really represent is a location in memory. Casting a pointer from one type to another does not modify its value, it simply lets the compiler know that the programmer knows what he is doing, and can therefore be allowed to use the pointer as the alternate type. Any pointer type can be converted into any other; it is up to the programmer to make sure that the pointer is still meaningful after the cast. When a general, data-type-inspecific pointer is needed (for instance, for memory allocation and deallocation) a `void *` is used. As you can see, pointers are infinitely flexible and with the right casts and forethought, a pointer can be made to do just about anything. Herein lies the greatest source of the power and flexibility of C and C++, but also the greatest danger.

In the following example, pointers are created and used with various typical data types. The concept of *output parameters* is also introduced. An output parameter is a parameter which does not pass a value into a function, but is intended to store a value computed by a function. Output parameters are accomplished with pointers as shown below.

```
/*  height - INPUT   Height of the rectangle (height >= 0)    *
 *  width  - INPUT   Width of the rectangle (width >= 0)      *
 * *diag   - OUTPUT  Diagonal of the rectangle (diag != NULL) *
 * *area   - OUTPUT  Area of the rectangle (area != NULL)     */
 void rectstats(int height, int width, double *diag, double *area)
 {
   if (diag && height >= 0 && width >= 0)
     *diag = sqrt( height*height + width*width );
   if (area && height >= 0 && width >= 0)
     *area = height*width;
 }
```

```
int height = 10;
int width = 20;
double diagonal;
double area;

rectstats(height, width, &diagonal, &area);
printf("Rectangle:\n");
printf("  Height:   %d\n", height);
printf("  Width:    %d\n", width);
printf("  Diagonal: %f\n", diagonal);
printf("  Area:     %f\n", area);

/* Example output:
   Rectangle:
     Height:   10
     Width:    20
     Diagonal: 22.360680
     Area:     200.000000
 */
```

3.3 I Swear It's Really A Pointer: Arrays

3.4 Fun With Strings (C-style)

3.5 A Walk in the Park: Traversing Arrays and Strings

4 Advanced Usage

4.1 Here We Go Again: Arrays of Strings

4.2 Kill Me Now: Multidimensional Arrays

4.3 Pointers to Nowhere: Handles

4.4 Back to Kindergarten: Pointer Arithmetic

5 Memory Management

5.1 The usual suspects: `malloc` and `free`

5.2 Array Management: `calloc`, `realloc`, and the `mem*` Family