More Than You Ever Wanted to Know About Pointers

Kyle Lemons

March 15, 2007

Contents

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

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

pointer "points to" something in memory. A pointer is a number that represents 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

[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.

store the data in an array and pass a *pointer* to the data into the function, only 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 permissible 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 permissible 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 speaking, `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 conceivable 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 consider 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 multidimensional 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-in-specific 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

As the title of this section suggests, when you use an array, you are really using a pointer. This section might get fairly in-depth, so feel free to skip over the more technical parts. Don't forget to come back to it, however, if you get lost in array traversal (section 3.5) or multidimensional arrays (section 4.2).

First, let us consider a simple array declaration.

```
int array[100];
```

What this does under the covers is, in essence, very simple, and the compiler generates the code necessary to do all of this without the programmer's knowledge. A block of memory large enough for 100 `int`egers is allocated, and a pointer to this memory is stored. As you may have guessed, `array` is the pointer in question, and as such is really an `int *`. Now, let us consider the standard method of accessing the contents of an array:

```
array[27] = 63;
```

Conceptually, this sets the 27th element of the array `array` to 63. In pointer speak, however, the above code is actually treated *identically* to the following:

```
*(array+27) = 63;
```

Notice the use of pointer arithmetic (see section 3.5) to set the value of the 27th element. For a more in-depth look at how to iterate through arrays, see section 3.5

Arrays are quite possibly the most common use of pointers, and array access and manipulation would not be possible without pointers. Almost any array

problem beyond the very basic ones can be accomplished with pointers, often more legibly and more efficiently. As we will see later, often problems which might not appear to involve arrays or pointers–notably, strings–can generalize very nicely into pointer-based implementations.

3.4 Fun With Strings (C-style)

Consider the following definition of three standard C-style strings:

```
const char *cstr = "This string is very typical.";
char *str = strdup("This string is very typical.");
char sarr[] = "This string is very typical.";
```

As you can see, each of the above definitions is slightly different, however they would each look identical if printed via `printf`. Each of the above declarations is slightly different. In the following paragraphs, we will explore each one individually. It is important to remember throughout this section and whenever you are working with strings that, by definition, a C-type string is terminated by a `NULL` byte (`'\0'`).

The string `cstr` is stored in read-only memory, and so is declared as a `const char *` so that the compiler will generate compile-time warnings if we violate this. This string may not be changed, cannot be appended to, tokenized, or manipulated in any way. It is, for all intents and purposes, untouchable. It is also interesting to note that two different strings defined in such a way with identical strings will both have the same address in read-only memory! This might seem unexpected, but it makes sense from a memory consumption standpoint. If there is no way that the string will be modified, then there is no danger in having two pointers point to it.

The string `str` is a duplicate of the read-only version which was passed to `strdup` (see section 5.2). The memory location occupied by this string is exactly the length of the string including the `NULL` terminating byte, so we are free to modify, tokenize, rearrange, and otherwise alter this string–as long as we stay within these bounds. Appending to this string would require allocation or reallocation of memory in order to accommodate the added length. This pointer can be incremented, decremented, and reassigned, but none of this changes that this block of memory was allocated dynamically and therefore *must be `free`d*. It would be in your best interest to always use a separate pointer to iterate through a dynamically allocated memory block, as this facilitates `free`ing the memory much more easily.

The string `sarr` is a contiguous block of read-write memory exactly long enough to hold the string and its `NULL` terminating byte. This memory is not dynamically allocated, and therefore must not be explicitly `free`d. This means, however, that it is often difficult to append data to a string like this. It is possible to specify a length in the brackets when defining this string, and so expanding the pre-allocated area into which this string can expand. Strings like this can be modified, tokenized, reorganized, zeroed, and rewritten, and as long as the bounds are respected the memory will be reclaimed automatically when

the variable goes out of scope. Remember, however, that the size of this block of memory is determined at compile time.

We also mustn't forget to `free` our `strcpy`'d string (see 5.1), as it was `malloc`'d implicitly (see section 5.2):

```
free(str);
```

3.5 A Walk in the Park: Traversing Arrays and Strings

As was stated in section 3.3, the following two statements are identical:

```
array[27] = 63;
*(array+27) = 63;
```

To hammer this equivalence point home further, consider the following two lines of code, which are again each identical to the two above examples:

```
*(27+array) = 63;
27[array] = 63;
```

In addition to the immobile pointer `array` used in all examples above, there are a few other ways to access the elements of arrays. Chief among these is access via a separate pointer. First, let us define and initialize such a pointer.

```
int *ap; // _a_rray _p_ointer
ap = array;
```

Recall that `array` is really just a `int *`, and so it contains an address. Assigning `ap` to this value essentially duplicates this pointer, so now `ap` "points" to the beginning of the array just like `array` does. The same array access idioms work here as well:

```
ap[26] = 27[ap] + *(ap+27);
```

The benefit that we gain from using a pointer to access the array comes from the mobility of such a pointer. For instance, iterating through an array can be done in any one of a number of ways, three of which are included here:

```
for (int i = 0; i < 100; ++i)
  array[i] = 0; // <=> // i[array] = 0; // <=> // *(array+i) = 0;
for (int i = 0; i < 100; ++i)
  ap[i] = 0;    // <=> // i[ap] = 0;    // <=> // *(ap+i) = 0;
for (ap = array; ap - array < 100; ++ap)
  *ap = 0;
ap = array; // set ap back to where it was
```

Again, for the pointer arithmetic used above, see section . All three of the methods of iterating through the array accomplish the same thing, but they each have their own unique strengths. The first is the most common when initializing an array for the first time. The second is very useful for initializing

a section of an array, where `ap` points to the beginning of the section and the length is 100. The third is useful in any number of situations, and the `for` condition could easily be changed to fit the situation. One more specialized form of the third would be for clearing a memory section between two pointers, in which case the conditional would be `ap < endptr`.

While all of the above examples can also apply to strings, there are some special (and very useful) ways of dealing with strings. Take a look at the following example, and note how the special characteristics of strings and pointers are used to make the code shorter and more legible.

```
char buf[100] = "This is not a test.  I repeat, this is NOT a test!";
char *c, *s;
c = s = buf;
int word = 0;
do
{
  if (*c == ' ' || *c == NULL)
  {
    printf("Word #%d is:\t\"", ++word); // print tag
    for (const char *x = s; x < c; ++x) // print letters in word
      printf("%c", *x); // one character at a time
    printf("\"\n"); // print the end of line
    while (*c && *c == ' ') ++c; // skip spaces
    s = c; // store new start position
  } // word completed
} while (*c++); // stop when we just finished examining NULL

/* Example Output:
    Word #1 is:     "This"
    ...
    Word #5 is:     "test."
    Word #6 is:     "I"
    ...
    Word #11 is:    "a"
    Word #12 is:    "test!"
*/
```

The key things to note in this example is how `*c` and others were used to easily determine if the end of string had been reached. This is made possible by the fact that the `NULL` byte (zero, in fact) evaluates to false in a Boolean context. This is used in conjunction with the post-increment to simplify the `do ... while` conditional (which is used in favor of a while or for to enable the loop to still operate on the `NULL` byte at the end just like it would on a normal space: as the end of a word) and still stop when the loop would overrun the bounds of the string. The numeric nature of pointers is used in the for loop to easily print out only the characters that appear between the start pointer `s` and the current pointer `c`, and a simple while loop is used to skip adjacent sequences of spaces so

as not to notice zero-length words. The above example is a very typical piece of code for a programmer who is familiar with the nature of pointers and strings, and shows how easily very complicated tasks can be accomplished with enough thought and knowledge of pointers. Read through the example, look at the comments, and try rewriting it from memory to help solidify the meanings and concepts used above. Iteration using pointers is a very important and useful skill in the C/C++ world.

4 Advanced Usage

4.1 Here We Go Again: Arrays of Strings

Arrays of Strings in C are actually not what they sound like they might be. They are actually arrays of pointers to characters. To make life even more interesting, the variable itself that represents the matrix is a pointer to a character pointer. The first step in understanding how to manipulate these is understanding what this means.

A pointer is always the same size. A pointer to a pointer is no exception. Every pointer has a pointer type based on the type of data that it points to, so in this case the pointer will be of type `char**`, because it is a pointer to a character pointer. When one dereferences a pointer, the result is a value of the pointer's type, in this case `char*`. As we know, strings can be referenced by a pointer pointing to the first character in the string, and this is exactly what this value will be. In essence, this means that an array of strings is essentially a pointer to a block of memory containing the addresses of the first characters of the strings in question. Examine the following functions, which are equivalent:

```
void PrintStrFrom(char **strarray, unsigned int idx)
{
  printf("strarray[%d] = %s\n", strarray[idx]);
}

void PrintStrFrom2(char **strarray, unsigned int idx)
{
  printf("strarray[%d] = ", idx);
  char *p = *(strarray+idx);
  while (*p)
  {
    printf("%c", *p++);
  }
  printf("\n");
}
```

```
void PrintStrFrom3(char **strarray, unsigned int idx)
{
  printf("strarray[%d] = ", idx);
  int cidx = 0;
  while (true)
  {
    char c = *(*(strarray+idx)+cidx);
    if (c)
      printf("%c", c);
    else
      break;
  }
  printf("\n");
}

char **strs; // = {"Testing", "One", "Two", "Three"};
strs = (char **)calloc(4, sizeof(char*));
strs[0] = strdup("Testing");
strs[1] = strdup("One");
strs[2] = strdup("Two");
strs[3] = strdup("Three");
PrintStrFrom(strs,1);
PrintStrFrom2(strs,2);
PrintStrFrom3(strs,3);
for (int i = 0; i < 4; ++i) free(strs[i]);

/* Example Output:
  strarray[1] = One
  strarray[2] = Two
  strarray[3] = Three
*/
```

You will notice that in `PrintStrFrom`, the strings are accessed "as a whole" through the array. This is by far the cleanest way to use the strings in the array, but the other two methods show equivalent but messier ways which, while bad examples of good code, are good examples of what's really going on behind the scenes. In `PrintStrFrom2`, a pointer is set to the beginning of the string in question (notice the method of indexing strarray, and note its equivalence to `strarray[idx]`), and then incremented as the characters are printed out. Finally, in `PrintStrFrom3`, the string itself is indexed by character, and the character to be printed is determined each time by calculation (notice the method of indexing both strarray and the string, and note its equivalence to `strarray[idx][cidx]`). The final section of code gives a little preview of what's to come with memory management and its application to arrays and pointers. If you are unfamiliar with the memory allocation facilities of C, take note of the comment at the instantiation of `strs` as a vague equivalent.

While arrays of strings are common and useful, the most interesting part of using them is their implementation. Continue reading to see how it compares to multidimensional arrays; if you understood this section well, you should find multidimensional arrays to be only a very small step up on the difficulty scale.

4.2 Kill Me Now: Multidimensional Arrays

As you should know if you have read the preceding material, an array is a block of memory. A multidimensional array is a special case of this concept. A multidimensional array is a block of memory containing pointers to other arrays. Before we delve into how they work, how to make them, and how to use them, let's look at some multidimensional array expressions. You might want to read some of these interpretations more than once.

- `int **mdarr1;` Define a variable called `mdarr` which holds an address of a block of memory which contains the address of an integer.
- `int *mdarr2[10];` Define a variable called `mdarr` which represents a block of memory containing ten pointers to integers.
- `int mdarr3[10][];` Define a variable called `mdarr` which represents a block of memory containing ten arrays of integers.
- `int mdarr4[10][10];` Define a variable called `mdarr` which represents a block of memory containing ten arrays of ten integers.

One thing that is very important to notice is the difference in wording of the above descriptions. When a variable or array contains the address of a value (integers in the above examples), it does not necessarily point to an array! It is up to the programmer to know if it points to an array (that is, a block of values), a single value, or no value at all (NULL pointer). Another very important thing to note is the behavior of the last three examples above. If you noticed the use of the word "represents" in the description, you were paying the requisite amount of attention to detail to be able to understand the nuances of pointers... If not, now is a good time to start. With the latter three examples above, the compiler knows something important about the array (the size), and can thus make optimizations (`mdarr[10][10]` is actually a single block of memory itself, **not** a pointer to one, and the compiler optimizes your array indexing to point within this block). For most coding purposes, these optimizations are completely transparent to the programmer... however, in some situations they are not.

Basically[4], the point is is that a double pointer is fundamentally different[5] from a 2D array allocated statically in that a 2D array is actually the same thing as a 1D array. It is simply a linear block of data elements in memory. This affects the pointer math a great deal. For example, say you have this:

[4]This explanation provided by David Worsham

[5]http://www.ibiblio.org/pub/languages/fortran/append-c.html - See section entitled "Why a double pointer can't be used as a 2D Array"

```
int arr[3][3] = {{3, 1, 1}, {1, 3, 1}, {1, 1, 3}};
int** ptr = (int**)arr; //This cast should be your first clue that
// a difference exists...arr is actually an int*!

printf("arr: %p\n", arr);
printf("ptr: %p\n", ptr);
// Check out the pointer math for this element...arr[x][y] is the same
// as *(arr + array_width * y + x)!  The compiler remembers the array
// width at compile time and does this translation
printf("arr[0][0]: %i also %i\n", arr[0][0], *(arr + 3 * 0 + 0));
// This is the same as (arr + 3 * 0 + 0) cause' of the &
printf("&arr[0][0]: %p\n", &arr[0][0]);
// This is equivalent to &(*(*(ptr + 0) + 0)) which becomes
// (*(ptr + 0) + 0) or just *ptr
printf("&ptr[0][0]: %i\n", &ptr[0][0]);
```

On my system this program output:

```
arr: 0x8F01C378
ptr: 0x8F01C378
arr[0][0]: 3 also 3
&arr[0][0]: 0x8F01C378
&ptr[0][0]: 3
```

So, 2D arrays still define the same linear data structure as 1D arrays but double pointers define a hierarchical data structure. Thus, they are not the same thing. Just remember: all arrays, regardless of dimension, are single pointers.

4.2.1 Example of multidimensional pointers and string handling

```
void split(char* tosplit, char* delim, char*** parsed, int* len)
{
  char **list = NULL;
  int tf = 0; // Token First index
  int tl = 0; // Token Length
  int tc = 0; // Token Count
  char *c = tosplit; // current character
  char *d = NULL; // delim pointer
  char *temp = NULL; // temp string
  while (c) // infinite loop and sanity check
  {
    d = my_strchr(delim, *c); // search delim for *c
    if (d || *c == '\0') // if we're at the end of a token
    {
      tl = int(c - tosplit) - tf; // calc length of token
      temp = (char *)calloc(tl+1, sizeof(char));
      strncpy(temp,tosplit+tf,tl+1);
      temp[tl] = '\0'; // null terminate
      list = (char**)realloc(list, (tc+1) * sizeof(char*));
      list[tc] = temp; // append to list
      tf = int(c - tosplit) + 1;
      ++tc; // increment token count
    } // Found Delimiter
    if (*(c++) == '\0') break; // exit AFTER '\0', not on.
  }
  if (parsed != NULL) *parsed = list; // the list we made
  if (len != NULL)    *len    = tc; // the final token count
}
void freesplit(char ***parsed, int *strings)
{
  // Sanity checks
  if (parsed == NULL || strings == NULL) return;
  // Loop over the strings
  for (int i = 0; i < *strings; ++i)
  {
    // free the small arrays (the strings)
    free((*parsed)[i]);
  }
  // Free the big array
  free(*parsed);
  *parsed = NULL; // prevent the usage of the freed memory
  *strings = 0; // set the string count to be zero.
}
```

Notice in the above code how the hierarchical data structure is created. The variable `list` is `reallocated` each time a new token is found and a pointer to the newly allocated string is put into the newly extended array. Also note that the caller is responsible for freeing both the strings in the list and the list itself, as they are allocated dynamically. The function `freesplit` can be the memory freeing complement to the above `split` function. The two functions above can be called as in the following example:

```
int main(int argc, char **argv)
{
  char *src = "Cut this string into pieces by spaces and the letter 'i'.";
  char *delim = " i";
  char **words;
  int wordcount;
  split(src, delim, &words, &wordcount);
  for (int w = 0; w < wordcount; ++w)
    printf("word[%d] = \"%s\"\n", w, words[w]);
  freesplit(&words, &wordcount);
  return 0;
}
```

4.3 Pointers to Nowhere: Handles

4.4 Back to Kindergarten: Pointer Arithmetic

4.5 Data, Data Everywhere: Pointers to Structs

5 Memory Management

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

Dynamic memory allocation is a very important concept in C. This section is a work in progress.

5.2 Even More Fun With Strings: String *Fun*ctions

`strlen` `size_t strlen(const char *str)`
Returns a `size_t` (typically an `unsigned int`) representing how many characters are in the string, not including the `NULL` byte. This will be zero if `*str == '\0'`.

`strdup` `char *strdup(const char *str)`
Returns a **newly allocated** copy of `str`.

`strcpy` `char *strcpy(char *dest, const char *src)`
Copies the string pointed to by `src` into the memory location beginning at `dest`. Be careful, this does no size checking! Returns `dest`.

`strncpy` `char *strncpy(char *dest, const char *src, size_t len)`
Copies at most `len` characters from the string pointed to by `src` into the memory location beginning at `dest`. Returns `dest`. This is preferred to `strcpy`, as it is much safer and less overflow-prone.

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