Pointer Arithmetic

Introduction

Prep by:Mir Waqar

There's a joke that C has the speed and efficieny of assembly language combined with readability of....assembly language. In other words, it's just a glorified assembly language.

It's perhaps too harsh a judgement of C, but certainly one of the reasons the language was invented was to write operating systems. Writing such code requires the ability to access addresses in memory in an efficient manner. This is why pointers are such an important part of the C language. They're also a big reason programmers have bugs.

If you're going to master C, you need to understand pointer arithmetic, and in particular, the relationship between arrays and pointers.

Arbitrary Pointer Casting

Because most ISAs use the same number of bits as integers, it's not so uncommon to cast integers as pointers.

Here's an example.

```
    // Casting 32 bit int, 0x0000ffff, to a pointer
    char * ptr  = reinterpret_cast<char *>( 0x0000ffff ) ;
    char * ptr2 = reinterpret_cast<char *>( 0x0000ffff ) ;
```

In general, this is one of the pitfalls of C. Arbitrary pointer casting allows you to point anywhere in memory.

Unfortunately, this is not good for safe programs. In a safe programming language (say, Java), the goal is to access objects from pointers only when there's an object there. Furthermore, you want to call the correct operations based on the object's type.

Arbitrary pointer casting allows you to access any memory location and do anything you want at that location, regardless of whether you can access that memory location or whether the data is valid at that memory location.

Arrays and Pointer Arithmetic

In C, arrays have a strong relationship to pointers.

Consider the following declaration.

```
     int arr[ 10 ] ;
```

What type does **arr** have? You might say "it's an int array". And, so it is. However, **arr**, by itself, without any index subscripting, can be assigned to an integer pointer.

OK, now what do you think of **arr[i]**? What is that? (Assume **i** is some int). You might say, it's the **i**th element of the array. That's correct too. What type does **arr[i]** have?

It's an **int**! (Beginning programming students often think **arr[i]** is not the same as an int, because it uses brackets. It takes a while to convince them they have the same type as a plain int variable).

Do you know, however, that **arr[i]** is defined using pointer arithmetic?

In particular:

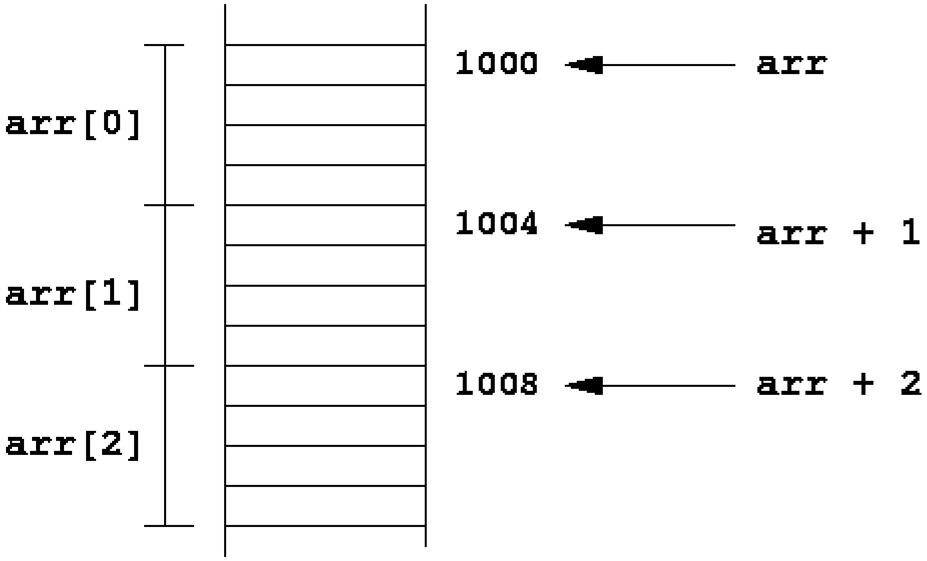
```
     arr[ i ] == * ( arr + i )
```

Let's take a closer look at **arr + i**. What does that mean? **arr** is a pointer to **arr[0]**. In fact, it is defined to be **& arr[0]**.

A pointer is an address in memory. **arr** contains an address in memory---the address where **arr[0]** is located.

What is **arr + i**? If **arr** is a pointer to **arr[0]** then **arr + i** is a pointer to **arr[i]**.

Perhaps this is easier shown in a diagram.



We assume that each **int** takes up 4 bytes of memory. If **arr** is at address 1000, then **arr + 1** is address 1004, **arr + 2** is address 1008, and in general, **arr + i** is address **1000 + (i * 4)**.

Now that you see the diagram, something may strange. Why is **arr + 1** at address 1004 and not 1001?

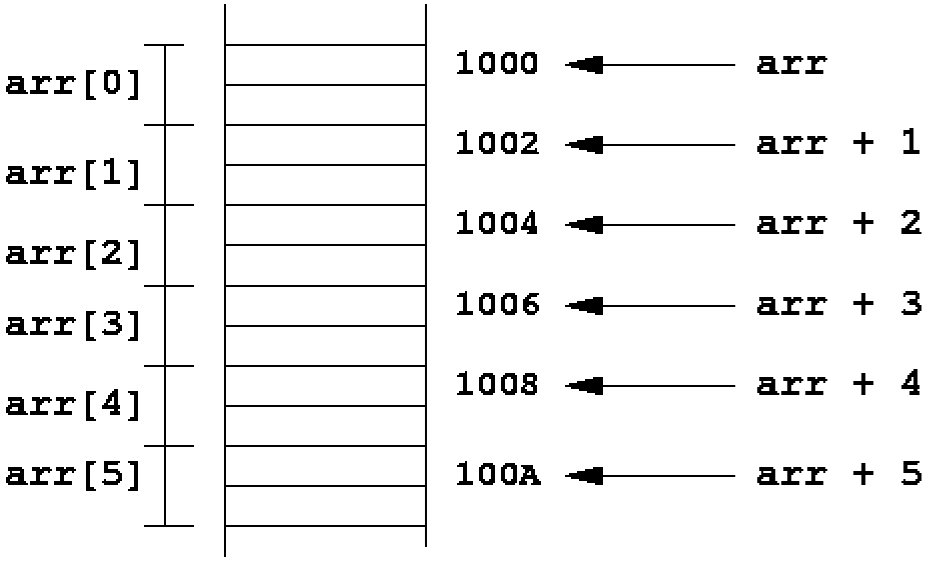
That's pointer arithmetic in action. **arr + 1** points to one element past **arr**. **arr + 3** is points to 3 elements past **arr**. Thus, **arr + i** points to **i** elements past **arr**.

The idea is to have **arr + i** point to **i** elements after **arr** regardless of what type of element the array holds.

The Type's the Thing

Suppose the array had contained **short** where a **short** is only 2 bytes. If **arr** is at address 1000, then, **arr + 1** is address 1002 (instead of 1004). **arr + 2** is at address 1004 (instead of 1008), and in general **arr + i** is at address **1000 + (2 * i)**.

Here's a diagram of that.



Notice that **arr + 1** is now 2 bytes after **arr** and **arr + 2** is 4 bytes. When we had an int array, **arr + 1** was 4 bytes after **arr** and **arr + 2** was 8 bytes afterwards.

Why is there a difference? What's the differnce between **arr** before and now?

The difference is in the type. Before, **arr** had (roughly) type **int *** and now **arr** has type **short ***.

In C, a pointer is not only an address, it tells you what (in principle) the data type at that address is. Thus, **int *** is a pointer to an int. Not only does it tell you the data type, but you can also determine the data type's size.

You can find out the data type's size with the **sizeof()** operator.

```
sizeof( int ) ==> 4
```

```
     sizeof( short ) ==> 4
```

That's important, because that's how pointer arithmetic computes address. It uses the size of the data type, which it knows from the pointer's type.

Here's the formula for computing the address of **ptr + i** where **ptr** has type **T ***. then the formula for the address is:

```
   addr( ptr + i ) = addr( ptr ) + [
sizeof( T ) * i ]
```

T can be a pointer

How does pointer arithmetic work if you have an array of pointers, for example:

```
   int * arr[ 10 ] ;
   int ** ptr = arr ;
```

In this case, **arr** has type **int ****. Thus, **T** has type **int ***. All pointers have the same size, thus the address of **ptr + i** is:

```
   addr( ptr + i ) = addr( ptr ) + [
sizeof( int * ) * i ]
                   = addr( ptr ) + [ 4 * i
]
```

Static arrays are constant

When you declare

```
   int arr[ 10 ] ;
```

arr is a constant. It is defined to be the address, **& arr[0]**.

You can't do the following:

```
   int arr[ 10 ] ;

   arr = arr + 1 ; // NO! Can't reassign to
arr--it's constant
```

However, you can declare a pointer *variable*.

```
   int arr[ 10 ] ;
   int * ptr ;

   ptr = arr + 1 ; // This is OK. ptr is a variable.
```

Parameter Passing an Array

If you pass an array to a function, it's type changes.

```
void foo( int arr[ 10 ], int size ) {
  // code here
}
```

The compiler translates arrays in a parameter list to:

```
void foo( int * arr, int size ) {
  // code here
}
```

Thus, it becomes **arr** becomes a pointer variable.

Why doesn't this cause problems? After all, won't we pass **arr** as an argument? Isn't **arr** a constant?

Yes, it is. However, we pass a **copy** of the address to **arr** the parameter. Thus, the copy can be manipulated while the original pointer address that was passed during the function call is unchanged.

Subtraction

We can also compute **ptr - i**. For example, suppose we have an int array called **arr**.

```
int arr[ 10 ] ;
int * p1, * p2 ;

p1 = arr + 3 ; // p1 == & arr[ 3 ]
p2 = p1 - 2 ; // p1 == & arr[ 1 ]
```

We can have a pointer point to the middle of the array (like **p1**). We can then create a new pointer that is two elements back of **p1**.

As it turns out, we can even point way past the end of the array.

```
int arr[ 10 ] ;
int * p1, * p2 ;

p1 = arr + 100 ; // p1 == & arr[ 100 ]
p2 = arr - 100 ; // p1 == & arr[ -100 ]
```

The compiler still computes an address, and does not core dump. For example, if **arr** is address 1000_{ten}, then **p1** is address **1400**$_{\mathbf{ten}}$ and **p2** is address **600**$_{\mathbf{ten}}$. The compiler still uses the formula for pointer arithmetic to compute the address.

Passing the Array Size

This is why it's usually important to keep track of the array size and pass it in as a parameter. When you're in a function, you can't tell the size of the array. All you have is a pointer, and that's it. No indication of how big that array is.

If you try to compute the array's size using **sizeof**, you just get 4.

```
   // Compiler translates arr's type to int *
   void foo ( int arr[], int size ) {
      // Prints 4
      cout << sizeof( arr ) ;

      int arr2[ 10 ] ;

      // Prints 40
      cout << sizeof( arr2 ) ;
   }
```

If you declare a local array (not using dynamic memory allocation), you can get the size of the array. However, once you pass that array, all that's passed is the address. There's no information about the array size anymore.

Two dimensional Arrays

Suppose you declare:

```
   int arr[ 10 ][ 12 ] ;
```

What type is **arr**? You may have been told that it's **int ****, but that's incorrect. Two dimensional arrays (as declared above) are contiguous in memory. If you create an array of pointers to dynamically allocated arrays, such as:

```
   int * arr[ 10 ] ;
```

then, **arr** has type **int **** (or at least has a type compatible with **int ****).

The type is rather complicated, and is due to the fact that **arr[0] = & arr[0][0]**, **arr[1] = & arr[1][0]**, and in general, **arr[i] = & arr[i][0]**.

Pointer arithmetic says that **arr + i** gives you **& arr[i]**, yet this skips an entire row of 12 elements, i.e., skips 48 bytes times **i**. Thus, if **arr** is address **1000_{ten}**, then **arr + 2** is address **1096_{ten}**.

If the array's type were truly **int ****, pointer arithmetic would say the address is **1008_{ten}**, so that doesn't work.

So, what's the lesson?

A two-dimensional array is not the same as an array of pointers to 1D arrays

The actual type for a two-dimensional array, is declared as:

```
   int (*ptr)[ 10 ] ;
```

Which is a pointer to an array of 10 elements. Thus, when you do pointer arithmetic, it can compute the size of the array and handle it correctly. The parentheses are NOT optional above. Without the parentheses, **ptr** becomes an array of 10 pointers, not a pointer to an array of 10 ints.

If you have a conventional two dimensional array, and you want to compute the address for **arr[row][col]** and you have **ROWS** rows (where ROWS is some constant) and **COLS** columns, then the formula for the address in memory is:

```
   addr( & arr[ row ][ col ] ) = addr( arr ) + [ sizeof(
int ) * COLS * row ]
                                              + [ sizeof(
int ) * col ]
```

Two dimensional arrays are stored in row major order, that is, row by row. Each row contains **COLS** elements, which is why you see **COLS** in the formula. In fact, you don't see **ROWS**.

When you have a 2D array as a parameter to a function, there's no need to specify the number of rows. You just need to specify the number of columns. The reason is the formula above. The compiler can compute the address of an element in a 2D array just knowing the number of columns.

Thus, the following is valid in C, i.e. it compiles:

```
   void sumArr( int arr[][ COLS ], int
numRows, int numCols ) {
   }
```

The following is also valid in C.

```
   void sumArr( int arr[ ROWS ][ COLS ], int numRows, int numCols ) {
   }
```

The compiler ignores ROWS. Thus, any 2D array with the **COLS** columns and any number of rows can be passed to this function.

The following, however, is NOT valid in C:

```
   void sumArr( int arr[][], int numRows,
int numCols ) {
   }
```

It's not syntactically valid to declare **int arr[][]** in C.

However, it's OK to write:

```
   void sumArr( int **arr, int numRows, int
numCols ) {
      }
```

Note that **int **arr** is an array of pointers (possibly to 1D arrays), while **int arr[][COLS]** is a 2D array. They are not the same type, and are not interchangeable.

Pointer Subtraction

It turns out you can subtract two pointers of the same type. The result is the distance (in array elements) between the two elements.

For example:

```
int arr[ 10 ] ;
int * p1 = arr + 2 ;
int * p2 = arr + 5 ;

cout << ( p2 - p1 ) ; // Prints 3
cout << ( p1 - p3 ) ; // Prints -3
```

The formula used is rather simple. Assume that **p1** and **p2** are both pointers of type **T ***. Then, the value computed is:

```
  ( p2 - p1 ) == ( addr( p2 ) - addr( p1 ) ) / sizeof( T )
```

This can result in negative values if **p2** has a smaller address than **p1**.

p2 and **p1** need not point to valid elements in an array. The formula above still works even when **p2** and **p1** contain invalid addresses (because they contain *some* address).

Pointer subtraction isn't used very much, but can be handy to determine the distances between two array elements (i.e., the difference in the array indexes). You may not know exactly which element you're pointing to using pointer subtraction, but you can tell relative distances.

Dereferencing causes problems

In general, you can make a pointer point anywhere. For example, **arr + 1000** is valid, even if the array has only ten elements. **arr - 1000** is also valid. That is, you can compute it, and it won't core dump.

However, *dereferencing* pointers to invalid memory causes problems. Thus, ***(arr - 1000)** core dumps because you are trying to access the address.

Here's an analogy. You can write down anyone's address on a piece of paper, however, you can't just go inside the person's house at that address (which is like dereferencing). Thus, computing addresses is fine, dereferencing it may cause problems if the address is not a valid address in memory.

Nasty Types in C

In "C" you can create really difficult types. For example, function pointers have horrid syntax, as do pointers in 2D arrays. This makes certain kinds of declarations in C a pain to read.

Fortunately, our goal is simply to understand pointer arithmetic which is adding integers to pointers and subtracting pointers. This allows us to get a view of how C computes addresses.

Summary

A knowledge of pointer arithmetic separates those who passably know C, from those who know C really well. It's said that a good programmer understands pointers very well, while an average programmer finds anything with more than one pointer difficult to manage (e.g., a pointer to a pointer to an int is seen as difficult).

Part of the reason we study this is the importance of pointers in C and the importance of C to systems programming, which falls under "low-level" pr