

COMP1917: Computing 1

9. Pointers

Reading: Moffat, Chapter 6.

Memory Structure

In order to fully understand how pointers are used to reference data in memory, here's a few basics on memory organisation.

0xFFFF	High Memory
0xFFFE	
⋮	
0x0001	Low Memory
0x0000	

Pointers

Pointers:

- a **pointer** is a special type of variable for storing the memory location or “address” of another variable
- pointers shouldn't be scary or confusing – provided they are used correctly
- pointers can make your code more compact and efficient.

Memory

- computer memory is a large array of consecutive data cells or **bytes**
- a **char** normally occupies one byte, a **short** 2 bytes, an **int** or **float** 4 bytes, a **double** 8 bytes, etc.
- when a variable is declared, the operating system finds a place in memory to store the appropriate number of bytes.
- if we declare a variable called **k**, the place where **k** is stored (also called the “address” of **k**) is denoted by **&k**
- it is convenient to print memory addresses in Hexadecimal notation

Variables in Memory

```
int k;
int m;

printf( "address of k is %X\n", &k );
printf( "address of m is %X\n", &m );

address of k is BFFFFB80
address of m is BFFFFB84
```

This means that `k` occupies the four bytes from `BFFFFB80` to `BFFFFB83`, and `m` occupies the four bytes from `BFFFFB84` to `BFFFFB87`.

Arrays in Memory

When an array is declared, the elements of the array are guaranteed to be stored in **consecutive** memory locations:

```
int array[5];

for( i=0; i < 5; i++ ) {
    printf("address of array[%d] is %X\n", i, &array[i]);
}

address of array[0] is BFFFFB60
address of array[1] is BFFFFB64
address of array[2] is BFFFFB68
address of array[3] is BFFFFB6C
address of array[4] is BFFFFB70
```

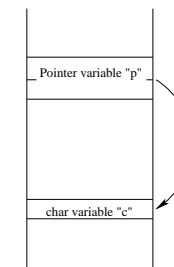
Size of a Memory Address

Just like any other variable of a certain type, a variable that is a pointer also occupies space in memory. The number of memory cells needed depends on the computer's architecture. For example:

- for an old computer, or a hand-held device with only 64KB of addressable memory, a pointer only requires 2 memory cells (i.e. 2 bytes) to hold any address from 0000 to $FFFF_{16} = 65535_{10}$
- for desktop machine with 4GB of addressable memory, a pointer requires 4 memory cells (i.e. 4 bytes) to hold any address from 00000000 to $FFFFFFFF_{16} = 4294967295_{10}$

Pointers

Suppose we have a pointer `p` that “points to” a `char` variable `c`. Assuming that the pointer `p` requires 2 bytes to store the address of `c`, here is what the memory map might look like:



The * Notation

Now that we have assigned to `p` the address of variable `c`, we need to be able to reference the data in that memory location.

operator `*` is used to access the object the pointer points to. Hence to change the value of `c` using the pointer `p`:

```
*p = 'T';    // sets the value of c to 'T'
```

The `*` operator is sometimes described as “dereferencing” the pointer, to access the underlying variable.

The * Notation *cont.*

Things to note:

- All pointers are constrained to point to a particular type of object.

```
// a potential pointer to any object of type char
char *s;
```

```
// a potential pointer to any object of type int
int *p;
```

- If pointer `p` is pointing to an integer variable `x`, then `*p` can occur in any context that `x` could.

Examples of Pointers

```
int *p; int *q; // this is how pointers are declared
int a[5];
int x = 10, y;
```

```
p = &x;    // p now points to x
*p = 20;   // whatever p points to is now equal to 20
y = *p;    // y is now equal to whatever p points to
p = &a[2];  // p points to an element of array a[]
q = p;     // q and p now point to the same thing
q++;       // q now points to the next element of a[]
```

Example - Swapping Variables

Question:

Can we write a function to “swap” two variables?

We will first show how to do it the **wrong** way, and then the **right** way (using pointers).

swap() - the wrong way

```
void swap( int a, int b )
{
    int temp;
    // only the local "copies" of a and b will swap
    temp = a;  a = b;  b = temp;
}
int main( void )
{
    int a = 5, b = 7;
    swap( a, b );
    // a and b will still have their original values
    printf( "a = %d, b = %d\n", a, b );
}
```

Function Parameters and Pointers

- in C, parameters are “call-by-value”
 - ▶ changes made to the value of a parameter do not affect the original variable
 - ▶ function swap() tries to swap the values of *a* and *b*, but fails because it only swaps the **copies**, not the “real” variables in main()
- we can achieve “simulated call-by-reference” by passing pointers as parameters
 - ▶ this allows the function to change the “actual” value of the variables.

swap() - the right way

```
void swap( int * p, int * q )
{
    int temp;
    // this will change the actual values of a and b
    temp = *p;  *p = *q;  *q = temp;
}
int main( void )
{
    int a = 5, b = 7;
    swap( &a, &b );
    // a and b will now be successfully swapped
    printf( "a = %d, b = %d\n", a, b );
}
```

Pointers and Arrays

Pointers and arrays are very similar. In general, any operation that can be achieved using an array can also be achieved using a pointer.

Hence if we have:

```
int a[10], x;
int *p;
```

```
p = &a[0]; // or equivalently  p = a;
```

then *p* now points to the start of the array *a*. Consequently:

```
x = a[4]  is the same as  x = *(p+4)
```

Pointers and Arrays *cont.*

- In general, if p points to $a[0]$, then
 $a[i]$ is the same as $*(p+i)$
- This is true regardless of the *type* or *size* of the variables in the array $a[]$. Hence:
 - ▶ $(p+1)$ always points to the next object;
 - ▶ $(p+i)$ points to the i -th object beyond p

Pointer Arithmetic

A subtle point to note:

- when you add (or subtract) an integer n from a pointer p , *e.g.*:

$p += n;$

the result is that p now points to the n -th object beyond the one p previously pointed to.

- this is true regardless of the kind of object p points to.
- in other words, n is *scaled* according to the size of the type of object p points to, which is determined by the declaration of p .

Pointers and Arrays *cont.*

The similarity between arrays and pointers does not end there. It is possible to write:

$*(p+i)$ as $p[i]$

and

$a[i]$ as $*(a+i)$

Given that it is possible to write $a[i]$ as $*(a+i)$, then a must be a synonym for the location of the array's initial element. Hence the following are equivalent:

$p = \&a[0]$ is the same as $p = a$

Pointers and Arrays *cont.*

Beware: although there are lots of similarities between array names and pointers, the following caveats apply:

- a pointer is a variable, so $p++$ is ok.
- an array name is **not** a variable, so $a++$ is **not** ok
 (for this reason, an array is sometimes regarded as a “constant pointer”)
- an array definition causes an allocation of data storage space equal to the size specified.
- when a pointer is declared, no space is set aside to store anything other than the pointer itself.

Expression and Precedence

Both `*` and `&`, being unary operators, have a precedence higher than arithmetic operators. So assuming `p` is a pointer then:

```
*p = *p + 3; // add 3 to whatever p points to
*p -= 2;     // subtract 2 from whatever p points to
*p != *q     // compare the values p and q point to
```

But note how `*` interacts with `++` and `--`

```
x = *(p++); // increment the pointer p itself
x = *p++;   // same as above, but considered bad style
x = (*p)++; // increment what p points to
```

Pointer Hazards

If an uninitialized or otherwise invalid pointer is used, or an array is accessed with a negative or out-of-bounds index, one of a number of things might happen:

- program aborts immediately with a “segmentation fault”
- a mysterious failure much later in the execution of the program
- incorrect results, but no obvious failure
- correct results, but maybe not always, and maybe not when executed on another day, or another machine

The first is the most desirable, but cannot be relied on.

Passing Arrays as Parameters

When an array is passed as a parameter to a function, what the function receives is actually a pointer to the first element of the array.

Hence, the function can change the actual values stored in the array (as in the “swap” example).

```
int main( void )
{
    int count[26] = { 0 };

    get_freq( count ); // count frequencies of input chars

    print_freq(count); // print the frequencies in a table
}
```

malloc() and free()

What happens if you do not know the size of an array at compile time?

You can dynamically create an array at run-time and assign it to a pointer, using the `malloc()` function.

The `malloc()` function is used to acquire a block of memory of a specified size from a central pool of memory called the “heap”.

If successful, `malloc()` returns a pointer to the space requested, else it returns a `NULL` pointer if the request cannot be satisfied.

The `free()` function is used to free up a previously `malloc`’ed block of memory, i.e. the parameter to `free()` must be a pointer to a space previously allocated by `malloc()`.

malloc() and free()

For example, let's assume we need a block of memory to hold an array of doubles, whose size is typed in by the user.

```
double *p; int n;
printf("Enter size of array: ");
scanf("%d",&n);
p = (double *)malloc( n * sizeof(double));
if( p == NULL ) {
    printf( "Error: array could not be allocated.\n" );
    exit( 1 );
}

//...we can now use p[] just like an array...

free(p); // free up the memory that was used
```

COMP1917

©Alan Blair, UNSW, 2006-2014

malloc() and free()

- Because malloc() returns a void * pointer (i.e. a pointer that is not typed) we need to **cast** the pointer into an appropriately typed pointer, in this instance we cast the pointer into a char pointer.
- Once you've finished using the block of memory, the memory needs to be returned back to the "heap" (central pool) using the free() function.
- You are only allowed to free() a block of memory that was previously malloc'ed; if you attempt to free a NULL pointer, or an unallocated pointer, it might cause an error.
- A malloc() without a corresponding free() is called a **memory leak**. Memory leaks eventually lead the program to crash, and are very difficult to track down.

COMP1917

©Alan Blair, UNSW, 2006-2014

Two-Dimensional Arrays

When a two-dimensional array is declared, the elements of the first row are stored into consecutive memory locations, followed by those of the second row, and so on.

```
float table[2][3];

table[0][0] is at BFFFFB80
table[0][1] is at BFFFFB84
table[0][2] is at BFFFFB88
table[1][0] is at BFFFFB8C
table[1][1] is at BFFFFB90
table[1][2] is at BFFFFB94
```

COMP1917

©Alan Blair, UNSW, 2006-2014