STEP TWO STUDY NOTES

This document contains some brief notes on topics covered in *Step Two* of the course '*Advanced C Programming: Pointers*'. You will find a similar set of study notes for each step of the course. These notes aim to *summarise* key concepts described in the lectures. You may find it useful to read the study notes *after* you have watched the lectures in each step of the course.

Some notes refer to specific sample programs supplied in the code archive of the course. If you need to explore the topics in more detail, please refer to the sample program. In these notes, relevant program names are shown like this:

ASampleProgram

NOTE:

These study notes aim to focus on *key points* explained in this course. They are not a substitute for the video lessons. The lessons cover topics in far more detail than the study notes. Moreover, also refer to the sample programs in the code archive. These contain working code – and lots of comments – to illustrate and explain how various pointer operations actually work.

Addresses and Indirection

Key Points

Arrays and Strings

- The value of an array identifier is the address of the array.
- The address of the first element of an array is also the address of the array.
- A string is an array of characters.
- A null character '\0' terminates a string.
- An array *is* an address but a pointer *contains* an address.
- A pointer may point to another pointer – this is called multiple indirection
- To obtain a value from a pointer that points to a pointer, use `**`
- A generic pointer points to `void` which means it can point to any data type
- It is good practice to free memory that is no longer referenced by a pointer

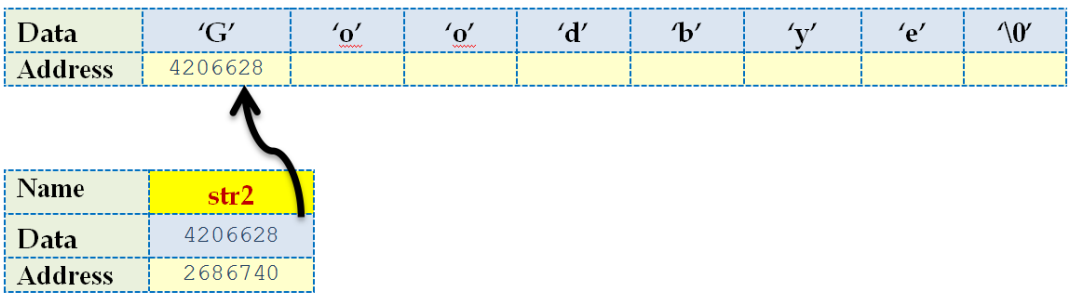
String Variables

```
char str1[] = "Hello";
char *str2 = "Goodbye";
```

Here `str1` is an array. It is the address at which the array of characters in the string "Hello" are stored. The address of `str1` and the address of "Hello" are the same. So, if (for instance) "Hello" has the address: 2686746, then the identifier `str1` also has the address 2686746.

But `str2` is a pointer. It points to (that is, it stores the address of) the string "Goodbye". The address of `str2` (the pointer variable) is *different from* the address of "Goodbye" (the string). But the *value* of `str2` (the data it 'contains') is the address of "Goodbye".

In this diagram the *address* of `str2` is 2686740 and the address of "Goodbye" is 4206628. So `str2` is stored at one address and "Goodbye" is stored at a different address. But the *value* of `str2` is the address of "Goodbye":



> The actual identifiers – names such as `str1` and `str2` – are replaced with numerical values when your program is compiled. It is the compiler which makes array identifiers identical with array addresses but causes pointer variables to be stored at different addresses from the array address to which they 'point'.

ARRAYS ARE SPECIAL!

It is commonplace to refer to array identifiers such as `str1` in the example given earlier as 'variables'. But they don't behave like normal variables because they can't be varied: once a value is assigned to an array identifier, that value cannot be reassigned. Let's assume we have assigned an array like this:

```
char str1[] = "Hello";
```

We cannot assign a new string (`char` array) to `str1`. This would not compile:

```
str1 = "New String";
```

Remember that `str1` is an address. If you try to assign a new array to it, you are trying to assign a new address. That is not permitted. You can, however, assign new values to specific elements by indexing into the array, so this *is* allowed:

```
str1[1] = 'x';
```

MULTIPLE INDIRECTION

Sometimes a pointer may point to a pointer. So if, for example, to have a pointer variable called `pi` that points to an integer, you can have another pointer variable, called `ppi` that points to the pointer variable, `pi`.

```
int *pi;            // a simple pointer to an integer
int **ppi;          // a pointer to a pointer to an integer
```

Refer to the *MultipleIndirection* sample program. This shows how to set `pi` to point to an array of integers (so `pi` stores the address of the array) and `ppi` points to `pi` (the address of the pointer variable, `pi`):

MultipleIndirection

```
ppi = &pi;
```

I can use `ppi` to obtain the item (the actual data, not the address) pointed to by `pi` using two stars `**` like this:

```
printf("item pointed to by double indirection of ppi is %d\n\n",
       **ppi);
```

**ARGV

Probably the commonest place where you will see multiple indirection is in the `main()` function of many C programs. This often declares the argument `argv` preceded by two stars.

```
int main(int argc, char **argv)
```

As you know that means it is a pointer to a pointer to, in this case, a `char`. The `argc` argument gives us the number or 'count' of arguments, while the `argv` (the 'argument vector') argument is initialized with any arguments passed at the commandline. The first argument is the program name itself and subsequent arguments are any strings entered after the program name.

If you want to access the arguments you can simply treat `argv` as an array and print out the strings in a simple for loop as I have done here:

MultipleIndirectionWithStrings

```
for (int i = 0; i < argc; i++) {
     printf("arg %d is %s\n", i, argv[i]);
}
```

GENERIC POINTERS

A generic pointer is written as a pointer to `void`:

GenericPointers

```
void *gp;
```

When you use a generic pointer to access some data you must first cast it to a specific data type. Here I cast `gp` to a pointer to `int`:

```
printf("item pointed to by gp is now %d\n", *(int*)gp);
```

MEMORY ALLOCATION

You can allocate memory using the `malloc` function. This allocates memory on the heap. The heap is where global data is stored in memory. The local variables declared inside a function are allocated memory in an area called the stack. When you exit from a function, the variables on the stack are cleaned up. But data assigned to variables on the heap continues to exist even after you've exited the function in which they were allocated. You may use `sizeof` to calculate the amount of memory you need to allocate.

malloc

```
char* s;
int stringsize;
stringsize = sizeof("hello");
s = (char*)malloc(stringsize);
```

CALLOC AND REALLOC

The `calloc` function is similar to `malloc` but it clears memory before allocating. `calloc` takes two arguments, the number of elements in the array and the size of each element. Here I allocate 6 integers:

calloc

```
int* p;
p = (int*)calloc(6, sizeof(int));
```

The `realloc` function lets you change the size of a block of allocated memory. You can call `realloc` with the pointer to the previously allocated block of memory, which here is `s`, a pointer to `char`, as the 1st argument. And the new size required, here 12 bytes, as the second argument:

realloc

```
s = (char*)realloc(s, 12);
```

FREE

It is good practice to free (dispose of) memory that was allocated when it is no longer required. It is no longer required when no pointer or variable in your program refers to that memory any longer. Keeping memory allocated even when it is not needed creates what are usually referred to as 'memory leaks'. Small memory leaks are rarely a problem on modern desktop computers with huge amounts of memory. This could be a problem if you happen to be writing C programs for devices that don't have much memory. For example, hardware controllers for machines or robotic devices might have very small amounts of memory. Here I free memory that was previously allocated for data referenced by the pointer `p`:

```
free(p);
```

POINTER ARITMETIC

You can increment a pointer variable to move one-element's worth of memory along an array. Here I have an array, `a`, of integers. I assign this to the `int` pointer `p`:

```
p = a;
```

On my PC, an `int` takes 4 bytes of memory. When I add 1 to the pointer `p`, it moves the pointer 4 bytes along in memory so it now points to the next integer in the array:

AddressArithmetic1

```
p = p + 1;
```

Pointer arithmetic works in this way with other data types too. If you are working with an array of doubles, long ints or other data types (even structs), adding 1 to the pointer will actually increment it *not by 1* but by *the number of bytes* taken up by whichever data-type the array stores. This has the effect of setting the pointer to 'point to' the next element in the array.

THE NUMBER OF BYTES NEEDED BY A STRUCT MAY VARY!

It is hazardous to assume you know how much memory a struct requires. That is because the data types (the fields) of a struct are 'aligned' in memory. The actual amount of memory required by a struct may vary according to the order in which its fields are declared.

To understand data alignment, imagine that I am storing my struct in a matrix of cells, like in a spreadsheet.

A	A	A	A				
B	B	B	B	B	B	B	B
C	C	C	C				
D	D	D	D	D	D	D	D

Here's my int A. It takes up 4 bytes. My Double B comes next. That takes 8 bytes. Another int, C, takes 4 bytes. And the long long int D takes 8. These data fields are arranged in memory much as I've arranged them in my spreadsheet. They form a nice, neat, regular matrix. It's not exactly the way memory is arranged in your computer but it might be useful to imagine it that way. This struct uses a matrix of 4 by 8 including some padding (unused bytes) after the smaller int fields. That's **32 bytes**.

But when I rearrange the field order, by putting the two int fields one after the other, the C compiler can optimize its use of memory. Instead of padding out the first int, A, with extra bytes, it just puts the next data field, which is now another int, C, alongside it. This also forms a nice neat matrix.

A	A	A	A	C	C	C	C
B	B	B	B	B	B	B	B
D	D	D	D	D	D	D	D

But this time, no extra bytes are needed to align the 8-byte data fields at 8-byte boundaries. So the matrix now takes 8 fewer cells in my spreadsheet or 8 fewer bytes in memory. My spreadsheet now has 3 rows and 8 columns. 3 times 8 equals 24. And my struct now takes up **24 bytes**.

Make a habit of using `sizeof` to calculate the amount of memory needed like this `(sizeof(MYSTRUCT))` rather than trying to work it out yourself. That is much, much safer.