Memory Organisation

When a program is executed, the values of variables are stored in the computer’s memory. Memory is effectively a gigantic array of bytes. Memory address are effectively an index of this array of bytes. These indexes can be very large e.g. up to $2^{32} - 1$ on 32-bit machine. You usually see memory address printed in hexadecimal (base-16).

Pointers

Definition
A pointer is a data type whose value is a reference to another variable. Thus the pointer refers to (points to) another variable.

Pointer Type
Pointers are compound types built from primitive types, they are declared using the primitive type and the pointer star.

```
int *ip; // int pointer
cchar *cp; // char pointer
double *fp; // float pointer
```

In most C implementation pointers store the the memory address of the variable they refer to.

You have seen the address-of operator before.

Address-of (&) operator
This operator returns a pointer of its operand.

```
int i = 5;
int *ip = &i;
```

Dereference (*) Operator
This operator accesses the variable referred to by the pointer.

```
*ip = *ip + 1;
```

Like other variables, pointers need to be initialised before they are used. Like other variables, its best if novice programmers initialise pointers as soon as they are declared. The value NULL can be assigned to a pointer to indicate it does not refer to anything. NULL is just a #define for 0. Most programmers prefer NULL for readability.
Pointer Example

```c
char c1 = 'a', c2 = 'b';
int i1 = 1, i2 = 2;
char *cp;
int *ip = NULL;

printf("%p\n", &c1);
printf("%p\n", &c2);
printf("%p\n", &i1);
printf("%p\n", &i2);

cp = &c1;
ip = &i1;

printf("%c\n", *cp);
printf("%d\n", *ip);

*cp = 'c';
*ip = *ip + 10;

printf("%c\n", c1);
printf("%d\n", i1);
```
**Pointer Arguments**

**Classic Example**

Write a function that swaps the values of its two integer arguments.

Before we knew about pointer arguments this would have been impossible, but now it is straightforward.

```c
void swap(int *n, int *m) {
    int tmp;
    tmp = *n;
    *n = *m;
    *m = tmp;
}
```

Andrew Taylor
COMP1911 Computing IA

**Array Representation**

**C Array Representation**

A C array has a very simple underlying representation, it is stored in a contiguous (unbroken) memory block and a pointer is kept to the beginning of the block.

```c
char s[] = "Hi!";
printf("s:\t%p\ts:\t%c\n\n", s, *s);
printf("&s[0]:\t%p\ts[0]:\t%c\n", &s[0], s[0]);
printf("&s[1]:\t%p\ts[1]:\t%c\n", &s[1], s[1]);
printf("&s[2]:\t%p\ts[2]:\t%c\n", &s[2], s[2]);
printf("&s[3]:\t%p\ts[3]:\t%c\n", &s[3], s[3]);
```

Array variables act as pointers to the beginning of the arrays!

Andrew Taylor
COMP1911 Computing IA

**Pointer Return Value**

You should not find it surprising that functions can return pointers. However, you have to be extremely careful when returning pointers.

**NB**

Returning a pointer to a local variable constitutes an error since that variable is destroyed when the function finishes executing.

But you can return a pointer that was given as an argument:

```c
int *increment(int *n) {
    *n = *n + 1;
    return n;
}
```

Nested calling is now possible: `increment(increment(&i));`

Andrew Taylor
COMP1911 Computing IA

**Array Representation**

Since array variables are pointers, it now should become clear why we pass arrays to `scanf` without the need for address-of (`&`) and why arrays are passed to functions by reference!

We can even use another pointer to act as the array name!

```c
int nums[] = {1, 2, 3, 4, 5};
int *iptr = nums;
printf("%d\n", nums[2]);
printf("%d\n", iptr[2]);
```

**NB**

Since `nums` acts as a pointer we can directly assign its value to the pointer `iptr`!
Array Representation

We can even make a pointer point to the middle of an array:

```c
int nums[] = {1, 2, 3, 4, 5};
i int *iptr = &nums[2];
printf("%d %d\n", *iptr, iptr[0]);
```

So is there a difference between an array variable and a pointer?

```c
int i = 5;
iptr = &i; // this is OK
nums = &i; // this is an error
```

Unlike a regular pointer, an array variable is defined to point to the beginning of the array, it is constant and may not be modified.

Pointer Comparison

Pointers can be tested for equality or relative order.

```c
double ff[] = {1.1, 1.2, 1.3, 1.4, 1.5, 1.6};
double *fp1 = ff;
double *fp2 = &ff[0];
double *fp3 = &ff[4];
printf("%d %d\n", (fp1 > fp3), (fp1 == fp2));
```

NB

Note that we are comparing the values of the pointers, i.e., memory addresses, not the values the pointers are pointing to!

Pointer Summary

Pointers:
- are a compound type
- usually implemented with memory addresses
- are manipulated using address-of(&) and dereference()
- should be initialised when declared
- can be initialised to NULL
- should not be dereferenced if invalid
- are used to pass arguments by reference
- are used to represent arrays
- should not be returned from functions if they point to local variables