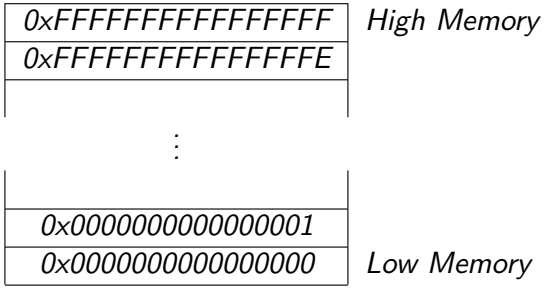
Memory Organisation

- During execution programs variables are stored in memory.
- Memory is effectively a gigantic array of bytes.
  COMP1521 will explain more
- Memory addresses are effectively an index to this array of bytes.
- These indices can be very large
  up to $2^{32} - 1$ on a 32-bit platform
  up to $2^{64} - 1$ on a 64-bit platform
- Memory addresses usually printed in hexadecimal (base-16).

Memory Organisation

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



Memory

- computer memory is a large array of *bytes*
- a variable will stored in 1 or more bytes
- on CSE machines a *char* occupies 1 byte, a an *int* 4 bytes, a *double* 8 bytes
- The & (address-of) operator returns a reference to a variable.
- Almost all C implementations implement pointer values using a variable's address in memory
- Hence for almost all C implementations & (address-of) operator returns a memory address.
- It is convenient to print memory addresses in Hexadecimal notation. should

Variables in Memory

```
int k;
int m;

printf( "address of k is %p\n", &k );
// prints address of k is 0xbffffb80

printf( "address of m is %p\n", &m );
// prints address of k is 0xbffffb84
```

- `k` occupies the four bytes from `0xbffffb80` to `0xbffffb83`
- `m` occupies the four bytes from `0xbffffb84` to `0xbffffb87`

Arrays in Memory

Elements of the array will be stored in *consecutive* memory locations:

```
int a[5];

int i = 0;
while (i < 5) {
    printf("address of a[%d] is %p\n", i, &a[i]);
}
// prints:
// address of a[1] is 0x7ffe693d61c4
// address of a[2] is 0x7ffe693d61c8
// address of a[3] is 0x7ffe693d61cc
// address of a[4] is 0x7ffe693d61d0
```

Size of a Pointer

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

- 32-bit platform: pointers likely to be 4 bytes
  e.g. older operating systems/machines
- 64-bit platform: pointers likely to be 8 bytes
  e.g. CSE machines, many student machines
- tiny embedded CPU: pointers could be 2 bytes
  e.g. your microwave

Pointers

A pointer is a data type whose value is a reference to another variable.

```
int *ip;   // pointer to int
char *cp;  // pointer to char
double *fp; // pointer to double
```

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

Pointers

- The & (address-of) operator returns a reference to a variable.
- The * (dereference) operator accesses the variable referred to by the pointer.

For example:

```
int i = 7;
int *ip = &i;
printf("%d\n", *ip); // prints 7
*ip = *ip * 6;
printf("%d\n", i);   //prints 42
i = 24;
printf("%d\n", *ip); // prints 24
```

Pointers

- 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 a #define in stdio.h
- NULL and 0 interchangable (where a pointer is expected).
- Most programmers prefer NULL for readability.

Pointer Arguments

We've seen that when primitive types are passed as arguments to functions, they are passed by value and any changes made to them are not reflected in the caller.

```
void increment(int n) {
    n = n + 1;
}
```

This attempt fails. But how does a function like `scanf` manage to update variables found in the caller? `scanf` takes pointers to those variables as arguments!

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

Pointer Arguments

We use pointers to pass variables *by reference*! By passing the address rather than the value of a variable we can then change the value and have the change reflected in the caller.

```
int i = 1;
increment(&i);
printf("%d\n", i);
//prints 2
```

In a sense, pointer arguments allow a function to 'return' more than one value. This greatly increases the versatility of functions. Take `scanf` for example, it is able to read multiple values and it uses its return value as error status.

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.

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

    tmp = *n;
    *n = *m;
    *m = tmp;
}
```

Pointer Return Value

You should not find it surprising that functions can return pointers. However, you have to be extremely careful when returning pointers. Returning a pointer to a local variable is illegal - that variable is destroyed when the function returns.
But you can return a pointer that was given as an argument:

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

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

Pointers to structs

If a function needs to modify a struct's field or if we want to avoid the inefficiency of copying the entire struct, we can instead pass a *pointer* to the struct as a parameter:

```
int scanZid(Student *s) {
    return scanf("%d", &((*s).zid));
}
```

The "arrow" operator is more readable :

```
int scan_zid(Student *s) {
    return scanf("%d", &(s->zid));
}
```

If **s** is a pointer to a struct **s->field** is equivalent to **(*s).field**

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.

```
char s[] = "Hi!";
printf("s: %p *s: %c\n\n", s, *s);
printf("&s[0]: %p s[0]: %c\n", &s[0], s[0]);
printf("&s[1]: %p s[1]: %c\n", &s[1], s[1]);
printf("&s[2]: %p s[2]: %c\n", &s[2], s[2]);
printf("&s[3]: %p s[3]: %c\n", &s[3], s[3]);
// prints
// s: 0x7fff4b741060 *s: H
// &s[0]: 0x7fff4b741060 s[0]: H
// &s[1]: 0x7fff4b741061 s[1]: i
// &s[2]: 0x7fff4b741062 s[2]: !
// &s[3]: 0x7fff4b741063 s[3]:
```

Array variables act like pointers to the beginning of the array!

Array Representation

Because array variables act like pointers, when we passed them to functions we can change the array.
We can also use pointers like array names **if they point at an array**:

```
int nums[] = {1, 2, 3, 4, 5};
int *p = nums;

printf("%d\n", nums[2]);
printf("%d\n", p[2]);
// both print: 3
```

Array Representation

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

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

There are differences between an array variable and a pointer.

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

Unlike a pointer, an array variable is constant and may not be modified.
It always points to the start of the array. of the array, it

Arrays As Function Parameters

Arrays are converted to pointers when pass as function parameters

```
// all 3 prototypes are equivalent
void print_array(int length, int array[length]);
void print_array(int length, int array[]);
void print_array(int length, int *array);
```

The first prototype is more readable but the length is ignored in the 2nd parameter.

Pointer Comparison

Pointers can be tested for equality or relative order.

```
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));
// prints: 0 1
```

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