Decimal Representation

- Can interpret decimal number 4705 as:

$$4 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$$

- The *base* or *radix* is 10
  Digits 0 – 9
- Place values:

···	1000	100	10	1
···	10^3	10^2	10^1	10^0

- Write number as 4705_{10}
  - Note use of subscript to denote base

Binary Representation

- In a similar way, can interpret binary number 1011 as:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

- The *base* or *radix* is 2
  Digits 0 and 1
- Place values:

···	8	4	2	1
···	2^3	2^2	2^1	2^0

- Write number as 1011_2
  $(= 11_{10})$

Hexadecimal Representation

- Can interpret hexadecimal number 3AF1 as:

$$3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 16^0$$

- The *base* or *radix* is 16
  Digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Place values:

···	4096	256	16	1
···	16^3	16^2	16^1	16^0

- Write number as $3AF1_{16}$
  $(= 15089_{10})$

Binary to Hexadecimal

0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111
8	9	A	B	C	D	E	F
1000	1001	1010	1011	1100	1101	1110	1111

- *Idea:* Collect bits into groups of four starting from right to left
- "pad" out left-hand side with 0's if necessary
- Convert each group of four bits into its equivalent hexadecimal representation (given in table above)

Binary to Hexadecimal

- Example: Convert 1011111000101001_2 to Hex:

1011	1110	0010	1001_2
B	E	2	9_{16}

- Example: Convert 10111101011100_2 to Hex:

0010	1111	0101	1100
2	F	5	C_{16}

Hexadecimal to Binary

- Reverse the previous process
- Convert each hex digit into equivalent 4-bit binary representation
- Example: Convert $\texttt{AD5}_{16}$ to Binary:

A	D	5
1010	1101	0101_2

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 indexes 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.

0xFFFFFFFF	*High Memory*
0xFFFFFFFE	
⋮	
0x00000001	
0x00000000	*Low Memory*

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 refered 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 (in modern C).
- 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

Passing by reference

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);
```

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.

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:

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

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

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.

```
char s[] = "Hi!";

printf("s:\t%p\t*s:\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!

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!

```
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:

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

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

```
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.

```
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