• 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:

  \[
  \cdots 1000 \quad 100 \quad 10 \quad 1 \\
  \cdots 10^3 \quad 10^2 \quad 10^1 \quad 10^0
  \]

• Write number as \(4705_{10}\)

  - Note use of subscript to denote base
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:

\[ \cdots 2^3 \quad 2^2 \quad 2^1 \quad 2^0 \]

Write number as 10112

(= 1110)
Hexadecimal Representation

- Can interpret hexadecimal number $3AF1_{16}$ 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:
  
  $$\ldots \quad 4096 \quad 256 \quad 16 \quad 1$$
  
  $$\ldots \quad 16^3 \quad 16^2 \quad 16^1 \quad 16^0$$

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
</tr>
</tbody>
</table>

• *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:

  $$
  \begin{array}{cccc}
  1011 & 1110 & 0010 & 1001_2 \\
  B & E & 2 & 9_{16}
  \end{array}
  $$

- Example: Convert $10111101011100_2$ to Hex:

  $$
  \begin{array}{cccc}
  0010 & 1111 & 0101 & 1100 \\
  2 & F & 5 & C_{16}
  \end{array}
  $$
Hexadecimal to Binary

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

<table>
<thead>
<tr>
<th>A</th>
<th>D</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>1101</td>
<td>0101(_2)</td>
</tr>
</tbody>
</table>
• 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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFFFFFFFFFFF</td>
<td>High Memory</td>
</tr>
<tr>
<td>0xFFFFFFFFFFFFFFFFF</td>
<td></td>
</tr>
<tr>
<td>0xFFFFFFFFFFFE</td>
<td></td>
</tr>
<tr>
<td>0x0000000000000001</td>
<td>Low Memory</td>
</tr>
<tr>
<td>0x0000000000000000</td>
<td></td>
</tr>
</tbody>
</table>
• 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 *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.
Variables in Memory

```c
int k;
int m;

printf( "address of k is %p\n", &k );
// prints address of k is 0xbfffffb80

printf( "address of m is %p\n", &m );
// prints address of k is 0xbfffffb84
```

- k occupies the four bytes from 0xbfffffb80 to 0xbfffffb83
- m occupies the four bytes from 0xbfffffb84 to 0xbfffffb87
Arrays in Memory

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

```c
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
```
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.

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

For example:

```c
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, it’s 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 interchangeable (where a pointer is expected).
- Most programmers prefer NULL for readability.
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.

```c
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!

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

```c
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.
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;
}
```
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:

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

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

```c
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:

```c
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.

```c
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.
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.
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));
// 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!
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