Understanding Numbers in Other Bases - Quick Intro

- helpful in COMP1511 if we understand writing numbers in other bases
- 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:

<table>
<thead>
<tr>
<th>( \ldots )</th>
<th>1000</th>
<th>100</th>
<th>10</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ldots )</td>
<td>10^3</td>
<td>10^2</td>
<td>10^1</td>
<td>10^0</td>
</tr>
</tbody>
</table>

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

writing Numbers in Binary (base 2)

- binary numbers are important in understanding computers
- The base or radix is 2
  Digits 0 and 1
- Place values:

<table>
<thead>
<tr>
<th>( \ldots )</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ldots )</td>
<td>2^3</td>
<td>2^2</td>
<td>2^1</td>
<td>2^0</td>
</tr>
</tbody>
</table>

- Write number as 1011\(_2\) and interpret as:
  \[ 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11_{10} \]
- You don’t need to convert numbers to/from binary in COMP1511
  - but nice to know how they work

writing Numbers in Hexadecimal (base 16)

- Binary numbers hard for humans to read - too many digits
- Conversion to decimal awkward and hides bit values
- Solution write the number in hexadecimal!
- 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:

<table>
<thead>
<tr>
<th>( \ldots )</th>
<th>4096</th>
<th>256</th>
<th>16</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ldots )</td>
<td>16^3</td>
<td>16^2</td>
<td>16^1</td>
<td>16^0</td>
</tr>
</tbody>
</table>

- Write number as 3AF\(_{16}\) and interpret as:
  \[ 3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 16^0 = 15089_{10} \]
- You don’t need to convert numbers to/from hexadecimal in COMP1511
  - but nice to know how they work
More detail about how memory works in our computer

- Let’s start with an idea of a neighbourhood
- Each house is a piece of memory (a byte or more, depending)
- Every house has a unique address that we can use to find it

Arrays work a bit like this . . .

- We’ve already seen indexing into arrays to find elements
- We could think of our entire computer’s memory as a big array of bytes

Every block of memory has an address

- The address is actually an integer
- If I have that address, it means I can find a variable wherever it is in memory
- Just like if I have an address to a house, I’ll be able to find it

Somewhere in memory:

<table>
<thead>
<tr>
<th>105</th>
<th>106</th>
<th>107</th>
<th>108</th>
<th>109</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Houses and addresses

Continuing the idea . . .

- A variable is a house
- That house is in a certain location in memory, its address
- The house contains the bits and bytes that decide what the value of the variable is

The address is an integer

- A 64-bit system, uses a 64 bit integer for address
- We can address \(2^{64}\) bytes of memory

Memory Organisation - Summarised

- During execution programs variables are stored in memory.

- Memory is effectively a gigantic array of bytes.
  - a variable will stored in 1 or more bytes
  - on CSE machines an \(\text{int}\) 4 bytes, a \(\text{double}\) 8 bytes

- Memory addresses are effectively an index to this array of bytes.

- These indices can be very large up to \(2^{64} - 1\) on a 64-bit platform

- Memory addresses usually printed in hexadecimal (base-16).

- COMP1521 will explain more
Introducing Pointers

A New Variable Type - Pointers

- Pointers are variables that hold memory addresses
- They are created to point at the location of other variables
- If a variable was a house, the pointer would be the address of that house
- In C, the pointer is like an integer variable that stores a memory address

Pointers in C

Pointers can be declared, but slightly differently to other variables

- A pointer is always aimed at a particular variable type
- We use a * to declare a variable as a pointer
- A pointer is most often "aimed" at a particular variable
- That means the pointer stores the address of that variable

We use & to find the address of a variable

```c
int i = 100;
// create a pointer called ip
int *ip;
// make it point at i
int *ip = &i;
```

Different pointers to point at different variables

```c
// some variables
int i;
double d;
// some pointers to particular variables
// * declares a pointer variable
// & finds the address of a variable
int *ip = &i;
double *dp = &d;
```

Initialising Pointers

Pointers should be initialised like other variables

- Generally pointers will be initialised by pointing at a variable
- NULL is a #define from most standard C libraries (including stdlib.h)
- If we need to initialise a pointer that is not aimed at anything, we will use NULL
Using Pointers

If we want to look at the variable that a pointer “aims at”

- We use * on a pointer to access (dereference) the variable it points at
- This is like following an address to get to a house, then looking inside

```c
int main(void) {
    int i = 100;
    // create a pointer called ip
    int *ip;
    // make it point at i
    ip = &i;
    // prints something like this:
    // address of x in hexadecimal is 0x7fff51155000
    printf("address of x in hexadecimal is %p\n", (void *)ip);
    return 0;
}
```

* %p in printf will print the address stored in a pointer

Pointers and Functions

Pointers allow us to pass around an address instead of a variable

- We can create functions that take pointers as input
- All function inputs are always passed in “by value” which means they’re copies, not the same variable
- But if I have a copy of the address of a variable, I can still find exactly the variable I’m looking for

Function variables pass in “by value”

In this case, the copy of the variable can’t ever change the value of the variable, because it’s just a copy

Pointers pass in “by value” also

The function has a copy of the pointer. However, even a copy of a pointer contains the address of the original variable, allowing the function to access it.
The following code illustrates the two examples

- A variable passed to a function is a copy and has no effect on the original
- A pointer passed to a function gives us the address of the original

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

// This function will have no effect!
// It only modifies its own variable 'n',
void increment_ptr(int *n) {
    *n = *n + 1;
}
```

pointers and Functions
We can now do more with functions

- Pointers mean we can give a function access to multiple variables
- This means one function can now change multiple variables at once

```c
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Pointers to Pointers
Pointers are variables
Pointers can point at variables
uh oh ...

For now, we will not use pointers aimed at other pointers, but in the future you may find uses for them

Let’s make a program using functions and pointers
This program is called The Shuffler

- It will take some numbers as inputs
- It will shuffle them a little, changing their order
- Then it will print them back out
- We’ll make some use of functions to separate our code
- We’ll show how pointers let us access memory in our program
What functions do we want?

Deciding how to split up our functionality

- A function that reads the inputs as integers
- A function that swaps two numbers
- A function that swaps several numbers
- A function that prints out our numbers

Reading Input

A function to read inputs into an array

- We’re also going to want to know how many numbers are being entered!

```c
int read_input(int deck[MAX_CARDS]) {
    int num_cards;
    printf("How many cards are in your deck?\n");
    scanf("%d", &num_cards);
    // loop through the deck, writing in the numbers that user types in
    int i = 0;
    while (i < num_cards && i < MAX_CARDS) {
        scanf("%d", &deck[i]);
        i++;
    }
    return i;
}
```

Printing our numbers

This is a straightforward function

The only issue is that we might have to work with an array that isn’t full

So we use `num_count` to stop us early if necessary

```c
void print_deck(int deck[MAX_CARDS], int length) {
    int i = 0;
    while (i < length && i < MAX_CARDS) {
        printf("%d, ", deck[i]);
        i++;
    }
    printf("\n");
}
```

Using Pointers to swap variable values

A simple swap function

This function doesn’t even know whether the ints are in arrays or not

It sees two memory locations containing ints

and uses a temporary int variable to swap them

```c
void swap_nums(int *num1, int *num2) {
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```
Shuffle performs some swaps

This function just loops through and swaps a few numbers

This is a good candidate for a function that could be changed or written differently and just used by our main without thinking about it

// call swap multiple times on different pairs to shuffle the deck
void shuffle(int deck[MAX_CARDS], int length) {
    int i = 0;
    while (i < length) {
        // find another index to swap with
        int j = (i * 3) % length;
        swap_nums(&deck[i], &deck[j]);
        i++;
    }
}

source code for shuffler.c

Using all the functions in the main

A nice main makes use of its functions

- It's very easy to read this main!
- It shows its steps using its function names
- There isn't much code to dig through

int main(void) {
    int deck[MAX_CARDS] = {0};
    int num_cards = read_input(deck);
    shuffle(deck, num_cards);
    print_deck(deck, num_cards);
}

source code for shuffler.c

It's a simple program, but what's different?

Using functions, we have much more readable code

- Large sections of code are outside of the main
- The main itself is now very readable
- Each separate piece of functionality is on its own

Pointers give us access to other parts of memory

- We can give access to our variables via pointers

Another Pointers Example

// Simple example illustrating use of pointers
// to pass 3 values back from a function
// Andrew Taylor - andrewt@unsw.edu.au
#include <stdio.h>
void powers(double x, double *square, double *cube, double *fourth_power);

int main(void) {
    double s, c, f;
    powers(42, &s, &c, &f);
    printf("42^2 = %lf\n", s);
    printf("42^3 = %lf\n", c);
    printf("42^4 = %lf\n", f);
    return 0;
}

void powers(double x, double *square, double *cube, double *fourth_power) {
    double s, c, f;
    *square = x * x;
    *cube = x * x * x;
    *fourth_power = x * x * x * x;
}

source code for powers.c
Functions, Arrays & Pointers

- if you pass an array to a function, actually a pointer to the first element is passed
- Still looks like as an array in the function
- This is why arrays as input to functions let you change the array
- Also why there is no way to get array size in the function

Functions and Arrays - Strategy 1 - Passing A Fixed Size Array

- Sometimes we want to pass the same size array every time to a function
- makes coding the function simple

```c
// print an array containing LENGTH elements
void print_fixed_size_array(int array[LENGTH]) {
    int i = 0;
    while (i < LENGTH) {
        printf("%d\n", array[i]);
        i++;
    }
}
```

source code for print_array.c

Functions and Arrays - Strategy 2 - Size As Separate Argument

- If we want to pass different-size arrays, can pass size as a separate argument
  - less convenient
  - can also be used to print first n elements of any array'

```c
// print size elements of array
void print_any_size_array(int size, int array[size]) {
    int i = 0;
    while (i < size) {
        printf("%d\n", array[i]);
        i++;
    }
}
```

source code for print_array.c

Functions and Arrays - Strategy 3 - Special Marker Value

- We could also put a special value in the array to mark finish
  - convenient
  - needs to be special value not used for anything else
  - error if array doesn’t contain special value

```c
// print array elements until element containing -1 reached
void print_array_until_marker(int array[]) {
    int i = 0;
    while (array[i] != -1) {
        printf("%d\n", array[i]);
        i++;
    }
}
```

source code for print_array.c
// Count the big positive and big negative values in the array nums.
// A value is big is < -100 or > 1000
// Counts are stored in the variables that big_positive & big_negative point to

void count_big(int length, int nums[length], int *big_positive, int *big_negative) {
    int big_pos = 0;
    int big_neg = 0;
    int i = 0;
    while (i < length) {
        if (nums[i] < -100) {
            big_neg++;
        } else if (nums[i] > 100) {
            big_pos++;
        }
        i++;
    }
    // store counts in the variables that big_positive & big_negative point to
    *big_positive = big_pos;
    *big_negative = big_neg;
}

#include <stdio.h>

void count_big(int length, int nums[length], int *big_positive, int *big_negative) {
    int array[SIZE] = {3, -4004, 110, 500, -6, 111, -3};
    int big_p;
    int big_n;
    count_big(SIZE, array, &big_p, &big_n);
    printf("big negative=%d big positive=%d\n", big_n, big_p);
    return 0;
}

source code for count_big.c