How does a computer store values

Ones and Zeros

- Computer memory can be thought of as a big pile of on-off switches with 0s stored as off and 1s stored as on
- We call these bits - short for binary digit
- We often collect these together into bunches of 8 bits
- We call these bytes

Bits and Bytes

- We are going to be working with chunks of memory made up of specific sizes
- You may have heard of things like 32-bit and 64-bit systems
Variables are a way of asking the computer to remember something for us.

Called a “variable” because it can change its value.

At any point in time a variable stores one value. Except quantum computers!

C variables have a type.

We’ll only use 2 types of variable for the next few weeks:

- **int** for integers (whole numbers), e.g.: 42, -1
- **double** for floating point numbers, e.g.: 3.14159, 2.71828
Naming your Variables

Remember . . . Half our coding is for people!

- Names are a quick description of what the variable is
  - Eg: “answer” and “diameter”
  - Rather than “a” and “b”
- We always use lower case letters to start our variable names
- C is case sensitive:
  - “ansWer” and “answer” are two different variables
- C also reserves some words
  - “return”, “int” and “double” can’t be used as variable names
- Multiple words
  - We can split words with underscores: “long_answer”
- **int** variables can store only whole numbers, with no fractions or decimals
- Most commonly **int** variables use 32 bits (which is also 4 bytes)
- This gives us exactly $2^{32}$ different possible values
- This is used for $-2^{31}$ to $2^{31} - 1$ i.e. -2,147,483,648 to +2,147,483,647
  - integers outside this range cannot be stored in an **int** variable
  - luckily we mostly don’t need integers this large

- Why are limits asymmetric?
  - zero needs a representation (pattern)!
int - what bits are used

- we can use C features taught in COMP1521 to peek at the bits of an int
- we will explain why these bit patterns are used in COMP1521

```bash
$ dcc print_bits_of_int.c -o print_bits_of_int
$ ./print_bits_of_int
Enter an int: 42
00000000000000000000000000101010
$ ./print_bits_of_int
Enter an int: 1
00000000000000000000000000000001
$ ./print_bits_of_int
Enter an int: 0
00000000000000000000000000000000
$ ./print_bits_of_int
Enter an int: -1
11111111111111111111111111111111
$ ./print_bits_of_int -42
Enter an int: -42
11111111111111111111111111111111010110
```
- **double** variables used to store **floating-point** numbers
- “floating point” means the point can be anywhere in the number
- Eg: 10.567 or 105.67 (the points are in different places in the same digits)
- commonly 8 bytes used to store a **double** variable
  - hence **double** because it is double the usual 4 byte size of ints
- 8 bytes \(\rightarrow\) 64 bits \(\rightarrow\) \(2^{64}\) possible values (bit patterns)
  - \(2^{64}\) is huge number but infinite number of reals in any interval
- use of bit patterns more complex, if you can’t wait until COMP1521: [https://en.wikipedia.org/wiki/Double-precision_floating-point_format](https://en.wikipedia.org/wiki/Double-precision_floating-point_format)
- reals in (absolute) range \(10^{-308}\) to \(10^{308}\) can be approximated
- DANGER: approximation errors can accumulate
  - explained in COMP1521
int main(void) {
    // Declaring a variable
    int answer;
    // Initialising the variable
    answer = 42;
    // Give the variable a different value
    answer = 7;
    // we can also Declare and Initialise together
    int answer_two = 88;
}
Printing Variables

Not just for specific messages we type in advance

We can also print variables to our terminal

// no variables
printf("Hello Tom\n");

// printing a variable
int number = 7;
printf("My number is %d\n ", number);

// print two variables
int first = 5;
int second = 10;
printf("First is %d and second is %d\n ", first, second);
Printing different types of variables

%d is for ints, %lf is for doubles

// print an int and a double
int diameter = 5;
double pi = 3.14159;
printf("Diameter is %d, pi is, %lf\n", diameter, pi);

- %d stands for “decimal integer”
- %lf stands for “long floating point number” (a double)
Reading Input into Variables

**scanf**

Reads input from the user in the same format as printf

The `&` symbol that tells scanf where the variable is (more details later in term)

```c
// reading an integer
int input;
printf("Please type in a number: ");
scanf("%d", &input);

// reading a double
double input_double;
printf("Please type in a decimal point number: ");
scanf("%lf", &input_double);
```

- scanf can be used in more complicated ways - don’t do it
  - keep your use of scanf simple
  - we’ll show you better ways to do other input
```c
int x, y, sum;
printf("Enter x: ");
scanf("%d", &x);
printf("Enter y: ");
scanf("%d", &y);
sum = x + y;
// These 6 printfs can be better replaced by a single printf
printf("%d", x);
printf(" + ");
printf("%d", y);
printf(" = ");
printf("%d", sum);
printf("\n");
```

source code for sum2a.c
int x, y, sum;
printf("Enter x: ");
scanf("%d", &x);
printf("Enter y: ");
scanf("%d", &y);
sum = x + y;
printf("%d + %d = %d\n", x, y, sum);

source code for sum2b.c
```c
int x, y, sum;
printf("Enter x: ");
scanf("%d", &x);
printf("Enter y: ");
scanf("%d", &y);
sum = x + y;
printf("%d + %d = %d\n", x, y, sum);
```

[source code for sum2b.c]
Numbers and Types

- Numbers in programs have types.
- Numbers with a decimal point are type `double`, e.g. 3.14159, -34.56, 42.0
- C also lets us write numbers in scientific notation: $2.4e5 \Rightarrow 2.4 \times 10^5 \Rightarrow 240000.0$
  - numbers in scientific notation are also type `double`
- Numbers without decimal point or exponent are type `int`, e.g. 42, 0, -24
- Numbers in programs are often called constants
  - unlike variables they don’t change
Giving Constants Names

- Constants are like variables, only they never change**

- It is useful to give constants (numbers) a name.
  - makes your program more readable.
  - make your program easier to update, if the constant appears in many places

- We can do this with the \#define command

- We name them in all caps so that we remember that they’re not variables

```
// Variables demo
// Marc Chee, February 2019
#include <stdio.h>

#define PI 3.14159265359
#define SPEED_OF_LIGHT 299792458.0

int main(void) {
    ...
```
• C supports the usual maths operations:

• Precedence is as you would expect from high school, e.g.:
  \[ a + b \times c + d/e \implies a + (b \times c) + (d/e) \]

• Associativity (grouping) is as you would expect from high school, e.g.:
  \[ a - b - c - d \implies ((a - b) - c) - d \]

• Use brackets if in doubt about order arithmetic will be evaluated.

• Beware division may not do what you expect.
Division in C

- C division does what you expect if either operand is a `double`
  - If either operand is a `double` the result is a `double`.
  - \(2.6/2 \implies 1.3\) (not 2!)

- C division may not do what you expect if both arguments are integers.
- The result of dividing 2 integers in C is an integer.
- The fractional part is discarded (not rounded!).
  - \(5/3 \implies 1\) (not 2!)

- C also has the `%` operator (integers only).
  - computes the modulo (remainder after division)
  - \(14 \% 3 \implies 2\)
Mathematical functions

- Mathematical functions not part of standard C library essentially because tiny CPUs may not support them
- A library of mathematical functions is available including: `sqrt()`, `sin()`, `cos()`, `log()`, `exp()`
  
  Above functions take a **double** as argument and return a **double**
- Functions covered fully later in course
- Extra include line needed at top of program: `#include <math.h>` (explained later in course)
- `dcc` includes maths library by default most compilers need extra option: `gcc needs -lm` e.g.:
  
  `gcc -o heron heron.c -lm`
int x, y;
int answer;
printf("Enter x: ");
scanf("%d", &x);
printf("Enter y: ");
scanf("%d", &y);
answer = x * x + y * y;
printf("x squared + y squared = %d\n", answer);

source code for squares.c
double fahrenheit, celsius;
printf("Enter Fahrenheit temperature: ");
scanf("%lf", &fahrenheit);
celsius = 5.0 / 9.0 * (fahrenheit - 32);
printf("%lf Fahrenheit = %lf Celsius\n", fahrenheit, celsius);

Source code for celsius2fahrenheit.c
printf & scanf as functions

- printf & scanf are functions
- scanf returns a value returns number of items read
- Use this value to determine if scanf successfully reads number.
- scanf could fail e.g. if the user enters letters
- OK for now to assume scanf succeeds
- Good programmers always check
Integer Overflow/Underflow - Advanced Topic

- storing a value in an int outside the range that can be represented is illegal
- unexpected behaviour from most C implementations
e.g. the sum of 2 large positive integers is negative
- may cause programs to halt, or not to terminate
- can creates security holes or explosions
  - google Ariane 5 explosion
- bits used for int can be different on other platforms
- C on tiny a embedded CPU in washing machine may use 16 bits
  \(-2^{15} \text{ to } 2^{15} - 1\) i.e. -32,768 to +32767
- we’ll show you in COMP1521 how to handle this, for now assume 32 bit ints
- also arbitrary precision libraries available for C
  manipulate integers of any size (memory permitting)
- only a finite subset of the reals can be represented exactly using `double`
  - e.g 1/3 and 1/10 can't be represented exactly as a double
- the rest are approximated

```c
// 0.1 can not be precisely represented as a double
double a = 0.1;
double b = 1 - (a + a + a + a + a + a + a + a + a + a);
// prints 1.11022e-16
printf("%g\n", b);
```

```
// 9007199254740993 can not be precisely represented as a double
double d = 9007199254740993;
// prints 9007199254740992.000000
printf("%lf\n", d);
```

(source code for precision0.c)

(source code for precision1.c)
- double approximation can produce subtle & dangerous errors
- COMP1521 will explain more but difficult area

```c
double x = 0.000000011;
double y = (1 - cos(x)) / (x * x);
// prints 0.917540 which is wrong by a factor of almost two
// correct answer is ~0.5
printf("%lf\n", y);
```

source code for double_catastrophe.c
Linux Command: `cp`

- Linux Command **`cp`**: copies files and directories.
- **`cp sourceFile destination`**
- If the destination is an existing file, the file is overwritten.
- If the destination is an existing directory, the file is copied into the directory.
- To copy a directory use **`cp -r sourceDir destination`**.
Linux Command: mv

- Linux Command mv moves or renames a file.
- mv source destination
- If the destination is an existing file, the file is overwritten
- if the destination is an existing directory
  the file is moved into the directory.
Linux Command: `rm`

- Linux Command `rm` removes a file.
- Usually no undo or recycle bin - be careful & have backups
- `rm filename`
- `rm -r directoryName`
  - This will delete a whole directory.
  - Be extra careful with this command