

---

# COMP1511 - Programming Fundamentals

— Week 1 - Lecture 2 —

---

# How does a computer remember things?

## Ones and Zeros 二进制

- Computer memory is literally a big pile of on-off switches
- We call these **bits** 二进制数字 two possible value 1 or 0
- We often collect these together into bunches of 8 bits = 1 bytes
- We call these **bytes** 字节

## 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 32bit and 64bit systems

# Variables

## A Variable is . . .

- Our way of asking the computer to remember something for us
- Called a "variable" because it can change its value
- A certain number of bits that we use to represent something
- Made with a specific purpose in mind

We're going to start out with two types of number variables:

- **int** - integer, a whole number (eg: 0,1,2,3)
- **double** - floating point number (eg: 3.14159, 8.534, 7.11)

浮点数

# Naming your Variables

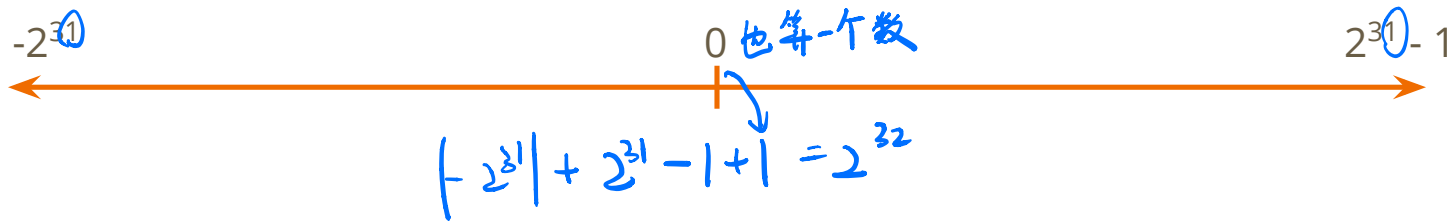
Remember . . . Half our coding is for people!

- Names are a quick description of what the variable is
  - Eg: "answer" and "diameter" 直径
  - Rather than "a" and "b"
- We always use lower case letters to start our variable names 小写
- 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 特定
- Multiple words
  - We can split words with underscores: "long\_answer" or capitals: "shortAnswer"  
分隔 eg. helloWorld

# int

## Integer

- A whole number, with no fractions or decimals
- Most commonly uses 32 bits (which is also 4 bytes)  $4 \times 8 = 32$
- This gives us exactly  $2^{32}$  different possible values
- This also means the integer is limited, it can't just hold any number
- Ranges from  $-2^{31}$  to  $2^{31} - 1$



# double

## A double-sized floating point number

- A decimal value - "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)
- It's called "double" because it's usually 64 bits, hence the double size of our integers

# Code for Variables

```
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;  
}
```

Q not mean equal → 重置 answer 是 7, 不是 42 了

second  
variable

新数与 7 不冲突

# Printing Variables

## printf

- Not just for specific messages we type in advance
- We can also print variables to our display

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

*name*

- `%d` - where in the output you'd like to put an int
- After the comma, you put the name of the variable you want to write



# Printing more variables

## printf with multiple variables

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

顺数来

The variables will match the symbols in the same order as they appear

# 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);
```

*name*

The `%d` and `%lf` are symbols that are part of `printf`

- `%d` stands for "decimal integer"
- `%lf` stands for "long floating point number" (a double)

*lf*

# Reading Input into Variables

`scanf` → opposite of `printf`

Reads input from the user in the same format as `printf`

Note the `&` symbol that tells `scanf` where the variable is

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

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

# Constants

Constants are like variables, only  
they never change

Can't call it a variable if it isn't ...  
variable

Sometimes we will use fixed numbers  
using a special command, **#define**

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) { ...
```

# Maths

算术

## Numbers and Arithmetic

A lot of arithmetic operations will look very familiar in C

- `+, -, *, /`
- These will happen in their normal mathematical order
- We can also use brackets to force precedence

```
// some basic maths
int x = 5;
int y = 10;
int result;
result = (x + y) * x;
printf("My maths comes out to: %d\n", result);
```

整数.

# int issues

## Some things to look out for

- **Overflow** and **Underflow**
- If we add two large ints together, we might go over the maximum value, which will actually roll around to the minimum value and possibly end up negative (trivia: Look up Ariane 5 explosion, a simple error like this caused a rather large problem)
- ints might not always be 32 bits . . . dependent on Operating System

# double issues

Doubles also have a few things to think about . . .

- No such thing as infinite precision 无限精确
- We can't precisely encode a simple number like  $\frac{1}{3}$
- If we divide 1.0 by 3.0, we'll get an approximation of  $\frac{1}{3}$
- The effect of approximation can compound the more you use them

除法

# Division and Variable types

## Division does some interesting things in C

- If either numbers in the division are **doubles**, the result will be a **double**
- If both numbers are **ints**, the result will be an **int**
  - Eg:  $3/2$  will not return 1.5, because ints are only whole numbers
- **ints** will always drop whatever fraction exists, they won't round nicely
  - $5/3$  will result in 1
- **%** is called **Modulus**. It will give us the remainder from a division between integers
  - Eg:  $5 \% 3$  will result in 2 (3 goes into 5 once, with 2 left over)

only  
remainder

不会四舍五入

分数



# Some More Linux

`rm -r` → move all things include directory itself

## File operations

- `cp source destination` - copy `cp name.c new name.c`
- `mv source destination` - move (can also be used to rename) `rm name.c` `..`
- `rm filename` - remove a file (delete) `递归`

The `-r` tag can be added to `cp` or `rm` to recursively go through a directory and perform the command on all the files

Eg: "`cp -r COMP1511 COMP1511Backup`" will copy all files from my `COMP1511` directory to my `COMP1511Backup` directory

# The if statement → make decisions

An **if** is like a question and an answer

- First we ask a question
- If we get the right answer, we run some code

```
// the code inside the curly brackets  
// runs if the expression is true (not zero)  
if (expression) {  
    code statement;  
    code statement;  
}
```

注意，电脑会跳过以上 code

# else - Adding to if statements

We can expand beyond the simple **if** by adding the **else** statement

```
if (expression) {  
    // this runs if the expression results in  
    // anything other than 0 (true)  
} else {  
    // this runs if the earlier expression  
    // results in 0 (false)  
}
```

# Chaining ifs and elses

This code shows ifs and elses joined together

```
if (expression1) {  
    // this runs if expression1 is true  
    // (anything other than 0)  
} else if (expression2) {  
    // this runs if expression1 is false (results in 0)  
    // and expression2 is true (results in anything  
    // other than 0)  
} else {  
    // this runs if both expression1 and  
    // expression2 result in false (0)  
}
```

# Asking the right Question

For our **if** to perform correctly, we need to know how to ask questions

**Relational Operators** work with pairs of numbers:

- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to
- == equals (we've already used "=" to assign values, so we use "==" to ask a question)
- != not equal to

All of these will result in 0 if false and a 1 if true

链接

# Chaining Questions Together

逻辑

We use **Logical Operators** to compare expressions

The first two are used between two expressions

- **&&** **AND** equates to 1 if both sides equate to 1 *both true*
- **||** **OR** is 1 if either side is 1 *one side is true is ok*

This is used in front of an expression

- **!** **NOT** is the opposite of whatever the expression was

# Some examples

| Expression                          | Result            |
|-------------------------------------|-------------------|
| 5 < 10 ✓                            | 1<br><i>true.</i> |
| 8 != 8                              | 0<br><i>false</i> |
| 12 <= 12 ✓                          | 1                 |
| 7 < 15 ✓ && 8 >= 15 ✗<br><i>and</i> | 0                 |
| 7 < 15 ✓    8 >= 15 ✗<br><i>or</i>  | 1                 |
| !(5 < 10 ✓    6 > 13 ✗)             | 0                 |

*both should false*

# What does our program need?

## All recipes need ingredients

- A way to “talk” to our user
  - We know about **printf**
- A way to receive input
  - We learnt about **scanf** today
- A way to compare numbers . . .
  - **Relational Operators**
- . . . Against a secret number
  - A **variable** or a **constant**
- A way to run different code depending on the number comparisons
  - **If** and **else** conditional statements



# Describe our program

We'll write some comments and include `stdio.h` for `printf` and `scanf`

//

```
/* The D&D Dice checker example  
   Marc Chee, June 2019
```

```
   This small example will ask the user to input the  
   result of two dice rolls.
```

```
   It will then check the sum of them against its  
   secret number.
```

```
   It will report back:
```

```
       a success (higher)
```

```
       a failure (lower)
```

```
       a tie (equal)
```

```
*/
```

```
#include <stdio.h>
```

# Define our Secret Number

`#define` is nice for things that we know aren't going to change. Note the use of all caps to signify a constant and underscores to show different words

This will go after our `#include`, but before our main

```
#define SECRET_TARGET 7
```

*constant*

# Main function

We always need a main function for C to know where our program starts

```
int main (void) {  
}
```

# Setting up some variables

We know we're going to be getting dice values from our user, so we can start by declaring them

```
// set up some dice variables so we can store numbers  
int dieOne;  
int dieTwo;
```

# Talk to our user and ask them for their dice rolls

Using printf and scanf, we can print words to the screen and read numbers

We store the two die rolls in the variables we set up earlier

```
// we start by asking the user for their dice rolls
printf("Please enter your first die roll. ");
// then scan their input
scanf("%d", &dieOne);
// repeat for the second die
printf("Please enter your second die roll: ");
scanf("%d", &dieTwo);
```

*Handwritten notes:*

- 整数* (Integer) pointing to `%d` in the first `scanf`.
- 变量* (Variable) pointing to `&dieOne`.
- name* pointing to `&dieTwo`.

# Total the dice

Using some basic arithmetic, we calculate our total

We then save that value in a “total” variable

```
// calculate the total and report it
int total = dieOne + dieTwo;
printf("Your total roll is: %d\n", total);
```

or `printf("-----> ", dieOne + dieTwo);`  
不需要 `int total`, 也可以算出答案

# Test the Total against the Target

We now use an if statement to test for success

```
// Now test against the secret number
if (total > SECRET_TARGET) {
    // success
    printf("Skill roll succeeded!\n");
}
```

# What about the failure?

The other option, which we actually don't have to test for, because it's all that's left.

```
// Now test against the secret number
if (total > SECRET_TARGET) {
    // success
    printf("Skill roll succeeded!\n");
} else {
    // the same as total <= SECRET_TARGET
    // but we don't have to test it because
    // we've already checked all other
    // possibilities
    printf("Skill roll failed!\n");
}
```