
COMP1511 - Programming Fundamentals

— Term 3, 2019 - Lecture 2 —

What did we learn Yesterday?

Introductions

- Course Administration
- Do you know where to find help? (Course website and forum!)
- Tips on how to do well in Programming

A first look at programming

- A brief look at computers and history
- Our very first C program . . .
- . . . Running in Linux

What are we covering today?

First Hour

- Variables and how we store information
- Some basic maths in code

Second Hour

- Conditionals - Running code based on questions and answers
- The **'if'** statement

A brief recap

```
// Demo Program showing output
// Marc Chee, February 2019

#include <stdio.h>

int main (void) {
    printf("Hello World.\n");
    return 0;
}
```

How does a computer remember things?

Ones and Zeros

- Computer memory is literally a big pile of on-off switches
- We call these **bits**
- 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 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
- 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 (eg: 0,1,2,3)
- **double** - floating point number (eg: 3.14159, 8.534, 7.11)

Naming your Variables

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
 - If we want multiple words, we can use capitals: “answerTwo” or “secondNumber”
- 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

int

Integer

- A whole number, not including fractions or decimals
- Most commonly uses 32 bits (which is also 4 bytes)
- This gives us 2^{32} different possible values
- Ranges from -2^{31} to $(2^{31} - 1)$
 - We have one less than 2^{31} because we had to use one of our values to represent zero



int issues

Some things to look out for

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

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

double issues

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

- No such thing as infinite precision
- We can't safely encode a simple number like $\frac{1}{3}$
- This means that a lot of doubles are approximate numbers
- The effect of approximation can compound the more you use them

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 answerTwo = 88;  
}
```

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);
```

- `%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);
```

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);
```

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)

Reading Input into Variables

scanf

Reads input from the user in the same format as printf

Note the **&** symbol that points the scanf back at a variable to store the input

```
// 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

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);
```

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
- **ints** will always drop whatever fraction exists, they won't round nicely for us
 - $5/3$ 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

Break Time

We'll take 5 minutes here before launching into some more C

What's a program?

- A series of instructions
- Some are: Store this information (using variables)
- Some are: Perform this task
- Usually these will be performed in order from top to bottom
- It's a little bit like a cooking recipe!

Some More Linux

File operations

- ***cp source destination*** - copy
- ***mv source destination*** - move (can also be used to rename)
- ***rm filename*** - remove a file

The **-r** tag can be added to copy or remove 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 folder to my COMP1511Backup folder

Letting our Computer Make Decisions

Sometimes we want to make decisions based on what information we have at the time

We can let our program branch between sets of instructions

In C this is the **if** statement



imgflip.com

JAKE-CLARK.TUMBLR

The if statement

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;
}
```

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 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 ==)
- != not equal to

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

Chaining Questions Together

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
- `||` *OR* is 1 if either side is 1

This is used in front of an expression

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

Some examples

Expression	Result
<code>5 < 10</code>	1
<code>8 != 8</code>	0
<code>12 <= 12</code>	1
<code>7 < 15 && 8 >= 15</code>	0
<code>7 < 15 8 >= 15</code>	1
<code>!(5 < 10 6 > 13)</code>	0

Let's create a program with our new skills

This program will help us in our games of “Catacombs and Large Reptiles”

1. A user will roll two dice and tell us the result of each die
2. Our program will add them together and check them against a target number that only the program knows.
3. It will then report back whether the total of the dice was higher, equal or lower than the secret number.

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
```

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 use two dice and it might also be nice to track the total . . . this goes inside the main

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

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);
```

Total the dice

Using some basic arithmetic, we calculate our total

We then save that value in our “total” variable

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

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");
}
```

Add more tests for other possibilities

We can add a test for a tie if there wasn't a success

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

What about the failure?

There's one 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 if (total == SECRET_TARGET) {
    // tie
    printf("Skill roll tied!\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");
}
```

We have a dice check program!

The finished code will be on the course website for you to peruse

A challenge:

Can you modify this code so that any rolls of “doubles” (both dice are the same number) are registered as criticals and are then output as either a critical success, critical failure or a critical tie?

What did we learn today?

Variables

- They come in different sizes and types
- Printing from variables and reading user input into variables
- Using maths to manipulate variables

Conditions

- **if** and **else** statements
- Relational Operators and Logical Operators