COMP1511 PROGRAMMING FUNDAMENTALS

Lecture 2

Throwing ourselves into the thick of it:
Variables and IF Statements

RECAP...

- Welcome and Introductions
- Course Administration
- How COMP1511 works
- How to get help and the best ways to approach learning Programming
- What is programming?
- What is Linux and working in Linux

TODAY...

- Variables and how we store information
- Maths in C!

- Conditionals - running out code based on some sort of condition being met
- IF statements

WHERE IS THE CODE?

LIVE LECTURE CODE CAN BE FOUND HERE:

https://cgi.cse.unsw.edu.au/~cs1511/21T3/live/

A BRIEF RECAP

OUR FIRST PROGRAM

```c
// A demo program showing output in C
// Sasha Vassar, July 2021 Hello!

#include <stdio.h>

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

BREAKING IT DOWN INTO PARTS

HEADER

```
// A demo program showing output in C
// Sasha Vassar, July 2021 Hello!
```


Words for humans

- Half our code is for the machine, the other half is for humans! (roughly)
- We put **“comments”** in to describe to our future selves or our colleagues what we intended for this code
- **//** in front of a line makes it a comment
- If we use **/*** and ***/** everything between them will be comments
- The compiler will ignore comments, so they don’t have to be proper code

BREAKING IT DOWN INTO PARTS

#INCLUDE IS A SPECIAL TAG FOR OUR COMPILER

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

- It asks the compiler to grab another file of code and add it to ours
- In this case, it's the Standard Input Output Library, allowing us to make text appear on the screen (as well as other things)
- Almost every C program you will write in this course will have this line

BREAKING IT DOWN INTO PARTS

THE "MAIN" FUNCTION

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

A **function** is a block of code that is a set of instructions

Our computer will run this code **line by line**, executing our instructions

- The first line has details that we'll cover in later lectures
  - int is the output - this stands for integer, which is a whole number
  - main is the name of the function
  - (void) means that this function doesn't take any input

BREAKING IT DOWN INTO PARTS

THE "MAIN" FUNCTION

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

Between the **{** and **}** are a set of program instructions

printf() makes text appear on the screen. It is actually another function from stdio.h which we included.

return is a C keyword that says we are now delivering the output of the function. A main that returns 0 is signifying a correct outcome of the program

EDITING AND COMPILATION

LET'S TRY THIS IN OUR EDITOR AND COMPILE IT

```c
// A demo program showing output in C
// Sasha Vassar, July 2021 Hello!

#include <stdio.h>

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

In the linux terminal we will open the file to edit by typing:

gedit helloWorld.c &

Once we're happy with the code we've written, we'll compile it by typing:

dcc helloWorld.c -o helloWorld

The -o part tells our compiler to write out a file called "helloWorld" that we can then run by typing:

./helloWorld

The **./** lets us run the program "helloWorld" that is in our current directory

AND WE ARE OFF!

WE NOW HAVE OUR FIRST WORKING PROGRAM...

- Try this yourself!
- Try it using VLAB via your own computer
- Try setting up a programming environment on your own computer (differing levels of difficulty depending on your operating system)

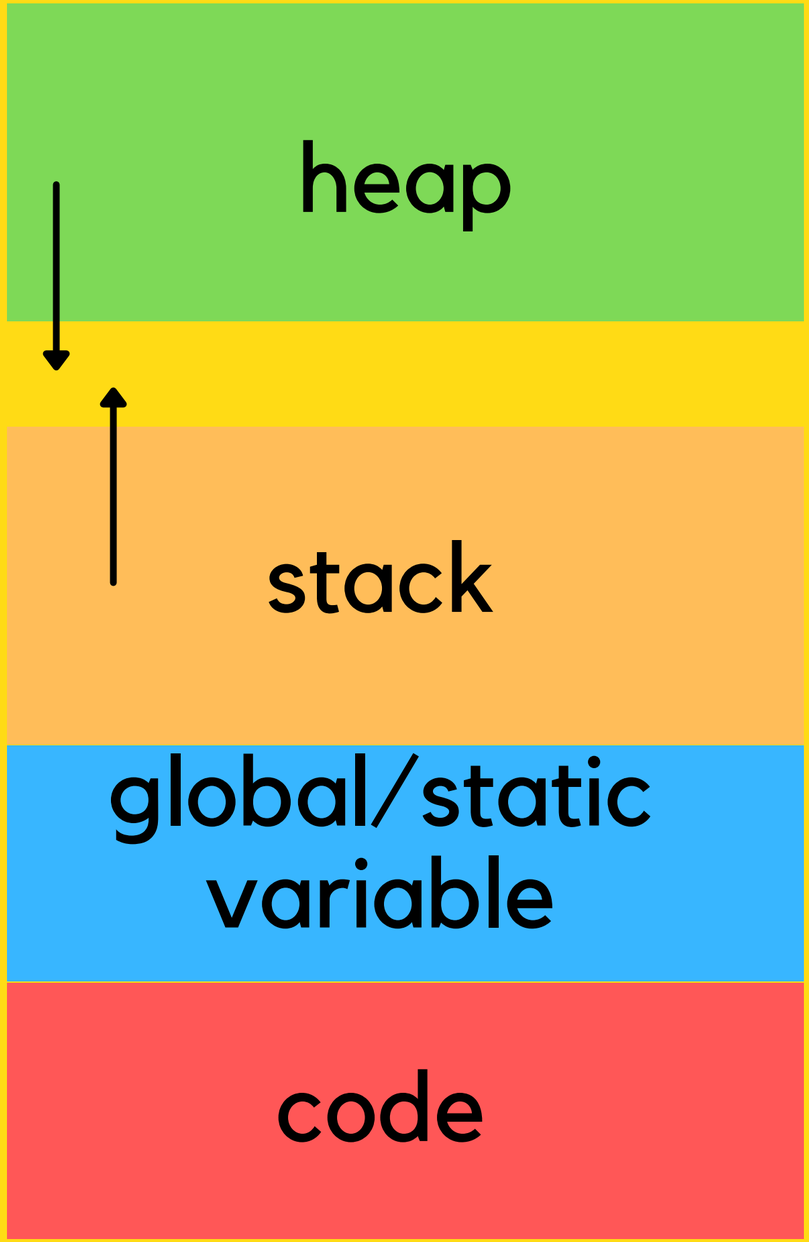
HOW DOES A COMPUTER REMEMBER THINGS?

ONES AND ZEROS!

- Computer memory is literally a big pile of on-off switches
  - We call these bits (smallest possible unit in computing, a bit is a choice between two things a 0 or a 1)
- We often collect these together into bunches of 8 bits
  - We call these bytes

WHAT DOES THIS LOOK LIKE?

When we execute code, the CPU will actually process the instructions and perform basic arithmetic, but the RAM will keep track of all the data needed in those instructions and operations.



WHAT IS VARIABLE?

- 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

WHAT KINDS OF VARIABLE WILL WE LEARN TODAY?

We're going to start out with three *data types* of variables:

int - integer, a whole number (eg: 0,1,2,3)

char - a single character (eg. 'a', 'A', etc)

double - floating point number (eg: 3.14159, 8.534, 7.11)

Each of these has a different number of bytes that are allocated in memory once the program is run...

NAMING OUR VARIABLES

IT IS AN ART - CALL IT LIKE YOU SEE IT, LIKE YOU USE IT AND SOMEONE ELSE HAS TO SEE IT!

Style Guide: We name our variables in ways that make it obvious what they are representing. Remember someone else has to be able to skim your code and know what you are saying/doing!

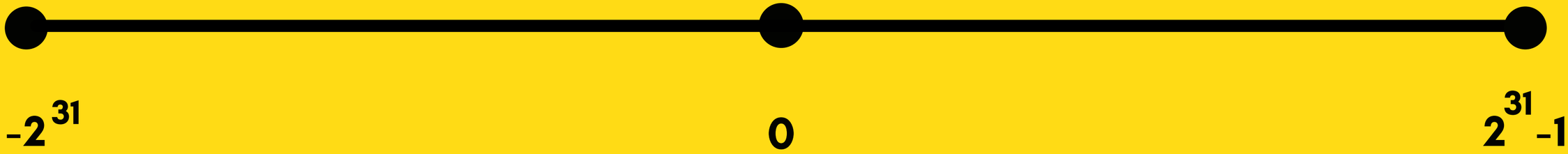
- Names are a quick description of what the variable is
  - Eg: “answer” and “diameter”
  - Rather than “a” and “b”
- We always use lower case letters to start our variable names
- C is case sensitive:
  - “ansWer” and “answer” are two different variables
- C also reserves some words
  - “return”, “int” and “double” can’t be used as variable names
- Multiple words
  - We can split words with underscores: "long_answer"

INTEGER

DATA TYPE: INT

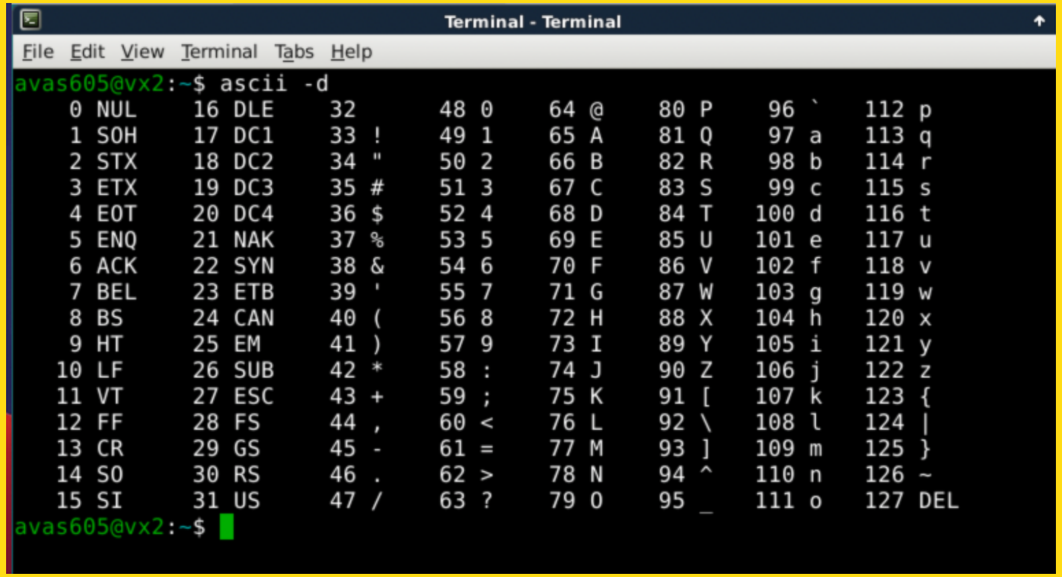
- A whole number, with no fractions or decimals
- Most commonly uses 32 bits (which is also 4 bytes)
- This gives us exactly 232 different possible values
- The maximum is very large, but it's not infinite!

Exact ranges from -2147483648 (-231) to 2147483647 (231 - 1)



CHARACTER

DATA TYPE: CHAR



- A single character in C can also be represented as an int!
- This is because a single character variable holds an ASCII value (integers 0-127), as opposed to the character itself
- The syntax to assign a single character is to put the character in single quotes: ‘a’
- So for a capital letter A:, the character is ‘A’ and the int stored is 65
- You use a char to declare a character: char letter = ‘a’ - this will assign 97 to the variable letter

DOUBLE

DATA TYPE: 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 (or 8 bytes)

LET'S TRY SOME CODE

DECLARE AND INITIALISE A VARIABLE

```c
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 OUT TO TERMINAL

PRINTF()

```
// printing a variable
int number = 13;
printf("My number is %d\n", number);
```

- Not just for specific messages we type in advance
- We can also print variables to our display!
- To print out a variable value, we use *format specifiers*, this is a % symbol followed by some characters to let the compiler know what data type you want to print..
- **%d** - where the output you'd like to put an int (decimal value, hence %d)
- After the comma, you put the name of the variable you want to write

PRINT OUT MANY VARIABLES

WHY NOT?

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

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

You can have as many as you want and of different types also!

LET'S TRY DIFFERENT TYPES OF NUMBERS

INTS AND DOUBLES - OH MY!

```
// 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 format specifiers that are used in printf statement to let the compiler know what data type we need to output.

Remember that we have to be very prescriptive when we tell the computer what to do and that extends to even telling it what types we are printing in C

- **%d** stands for "*d*ecimal integer"
- **%lf** stands for "*l*ong *f*loating point number" (a double)

WHAT ABOUT CHAR?

CAN'T FORGET THE LONELY CHAR

- The **%c** format specifier can also be used in printf statement to let the compiler know what data type we need to output (character).
    - **%c** stands for “character”

Don't forget that when you declare a char, you enclose it in single apostrophes to let the computer know that you are using a letter character

```
// print an int as a number and as a character
char letter = 'A';
printf("The letter %c has the ASCII value of %d\n",
letter, letter);
```

GREAT, WE CAN PRINT TO TERMINAL, CAN WE TAKE SOMETHING FROM TERMINAL?

- Reads *input* from the user in the same format as printf
- *Format specifiers (%d, %lf or %c)* are used in the same way as for the printf statement
- The *&* symbol tells *scanf* the address of the variable in memory (where the variable is located) that we want to place the value into (more details later in term)

SCANF()

```
// 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);
```

WHAT ABOUT OUR LONELY CHAR?

CHAR

- If you want scanf to read in a character, you will need to declare a character by using the keyword: **char**
- Even though you have declared a char to store your character into, it is still stored as an ASCII value… so you can move between **%d** and **%c** when you printf this variable

```
// reading a single character as a character
char input_character;
printf("Please type in a single character: ");
scanf("%c", &input_character);
```

WHAT IF A VARIABLE NEVER CHANGES?

THEN IT IS MOST LIKELY A CONSTANT...

Style Guide: We name them in all caps so that we remember that they're not variable

Constants are like variables, only they **never** change!
To define a constant, we use *#define* and follow it with the name of the constant and the value

```
// Using constant variables
// Sasha Vassar, July 2021

#include <stdio.h>

#define PI 3.14159265359
#define MEANING_OF_LIFE 42
#define SPEED_OF_LIGHT 299792458.0

int main (void) { ...
```

LET'S TALK ABOUT MATHS

WE LOVE MATHS, RIGHT? C ALSO LOVES MATHS (SOMETIMES WITH QUIRKS).

A lot of arithmetic operations will look very familiar in C

- adding +
- subtracting -
- multiplying *
- dividing /

These will happen in their normal mathematical order

We can also use brackets to force precedence

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

SO SUPER FUN FACT, YOU CAN DO MATHS WITH CHAR BECAUSE THEY ARE JUST INTS!

- Because characters are represented as ints inside the variable, you are able to move around the ASCII values by adding or subtracting to them.
- For example, if you are at *'a'* and you want to get to *'b'*, you can add 1

```
// some basic maths
char letter = 'a';
char next_letter = letter + 1;
printf("My original letter %c has the ASCII value
%d\nThe next letter %c has the ASCII value %d\n",
letter, letter, next_letter, next_letter);
```

THE QUIRKS OF INTEGERS… THERE ARE ALWAYS PROBLEMS

INTEGER OVERFLOW/ INTEGER UNDERFLOW

- Check out Boeing 787 that had to be rebooted every 248 days (2^31 hundredths of a seconds) https://www.engadget.com/2015-05-01-boeing-787-dreamliner-software-bug.html
- 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 (Check out Ariane 5 explosion, a simple error like this caused a rather large problem: https://www.bbc.com/future/article/20150505-the-numbers-that-lead-to-disaster)
- In a less destructive example, the video Gangham Style on YouTube maxed out the views counter : https://www.bbc.com/news/world-asia-30288542
- ints might not always be 32 bits . . . dependent on Operating System

THE QUIRKS OF DOUBLES...

OFFENDING REPEATERS

- No such thing as infinite precision
- We can’t precisely encode a simple number like ⅓
- If we divide 1.0 by 3.0, we'll get an approximation of ⅓
- The effect of approximation can compound the more you use them

NOW A LITTLE BIT ABOUT DIVISION

DIVISION IS INTERESTING IN C

- Remember that C thinks in data types
- 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 = 2 (because 5/3 = 1 rem 2)

BREAK TIME (5 MINUTES)

Fun riddle: You are standing in a room with three light switches. Each switch controls exactly one light bulb in the next room (we are on a tight budget here, so nothing fancy). The door to the next room is closed, and there are no windows, so you cannot see the light bulbs. You may manipulate the switches as many times as you like, then when you finish, you can go through to the room with the light bulbs. You must then say which switch controls which bulb. How do you do it?

ASKING THE COMPUTER TO MAKE SOME DECISIONS

IF STATEMENTS

- 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

WHAT KINDS OF PROBLEMS DO WE SOLVE WITH IF STATEMENTS?

DECISION PROBLEMS (YES/NO)

- A decision problem is a question with a YES/NO answer
- This is the perfect time to use an IF statement to help make the decision
- Eg. Is a number even? Is a number larger than 10? Is a number prime? etc.

IF STATEMENT

IT IS LIKE A QUESTION AND AN ANSWER

- First we ask the question - this is our *condition*
- If the answer to our question (*condition*) is YES, then we run the code in the curly brackets

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

WHAT IF THE ANSWER IS NO?

THERE ARE OPTIONS, THERE ARE ALWAYS OPTIONS

- If the answer to our question (*condition*) is NO, then we can add an **else** statement to let the computer know which other code may run

```
if (condition) {
    // code to run if the condition is true
    // or anything other than 0
} else {
    // run some other code instead
    // else is entered if the previous code
    // results in 0 (false)
}
```

WHAT IF THE ANSWER IS NO AGAIN?

MORE OPTIONS...

- If the answer to our question (*condition*) is NO, and the answer to our question (*condition*) in the else is also NO, then we can chain some **if** and **else** together to make an **else if** and create even more options in choosing which code to run...

```
if (condition1) {
    // code to run if condition1 is true
    // (anything other than 0)
} else if (condition2) {
    // code to run if condition1 is false (results in
0)
    // and condition2 is true (results in anything
    // other than 0)
} else {
    // code to run if both condition1 and
    // condition2 result in false (0)
}
```

HOW DO WE ASK GOOD QUESTIONS?

RELATIONAL OPERATORS

Notice that in C, we have both == and = These are not the same and do not mean what you are used to in Maths!
Using = when you assign values
Using == when you are checking for equivalence

Relational Operators work with pairs of numbers:

< less than

> greater than

<= less than or equal to

>= greater than or equal to

== equals

!= not equal to

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

I LIKE QUESTIONS, HOW DO I ASK TWO QUESTIONS AT THE SAME TIME?

LOGICAL OPERATORS

The first two are used between two questions (expressions):

- **&&** AND: if both expressions are true then the condition is TRUE (equates to 1 if both sides equate to 1)
- **||** OR: if any of the two expressions are true then the condition is TRUE (is 1 if either side is 1)

This is used in front of an expression:

- **!** NOT: reverse the expression (is the opposite of whatever the expression was)

SOME EXAMPLES

LET'S TRY THESE OUT...

```
if (12 <= 12) {
    //do something
}
```

```
if (8 != 8) {
    //do something
}
```

```
if (5 < 10) {
    //do something
}
```

```
if (7 < 15 && 8 >= 15) {
    //do something
}
```

```
if (7 < 15 || 8 >= 15) {
    //do something
}
```

```
if !(5 < 10 || 6 > 13) {
    //do something
}
```

LET'S PUT OUR SKILLS TO THE TEST

PRACTICAL EXAMPLE

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

BREAKING DOWN THE PROBLEM INTO A SUM OF SIMPLE PARTS

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

1. A user will roll two dice - *done outside of our program*
2. Take in the result of each die - *how do we read input?*
3. Add the die numbers together
4. Check them against a target number - *based on steps 4 and 5, it looks like we need to make a decision - therefore IF statement*
5. Output if total of the dice was higher, equal or lower than the target number - *output based on the decision that we made*

BREAKING DOWN FURTHER

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

1. ~~A user will roll two dice - done outside of our program~~
2. Take in the result of each die - how do we read input?
   - Read input of die 1
   - Read input of die 2
3. Add the die numbers together
   - sum = die1+die2
4. Check them against a target number - based on steps 4 and 5, it looks like we need to make a decision - therefore IF statement
   - Define the target number
5. Output if total of the dice was higher, equal or lower than the target number. - output based on the decision that we made
   - Is sum greater than target number?
   - Is sum less than target number?
   - Is sum equal to the target number?

NOW LET'S CODE!

Switch over toVLab

Open Terminal

Open a new file: *gedit dice_checker.c &*

Feel free to follow along with lecture coding, or you can also find the code here: https://cgi.cse.unsw.edu.au/~cs1511/21T3/live/

WHAT DID WE LEARN TODAY?

RECAP

Hello!
our first program

VARIABLES

They come in different shapes and sizes - int, double and char
Printing from variables (printf)
Reading user input into variables (scanf)
Using maths with variables

CONDITIONS

if /else /else if
Decision problems
Relational Operators
Logical Operators

DICE_CHECKER

Putting it all together in code

ANY QUESTIONS?

DON'T FORGET YOU CAN ALWAYS EMAIL US ON CS1511@CSE.UNSW.EDU.AU FOR ANY ADMIN QUESTIONS

PLEASE ASK IN THE FORUM FOR CONTENT RELATED QUESTIONS