DPST1091 / CPTG1391

Introduction to Programming

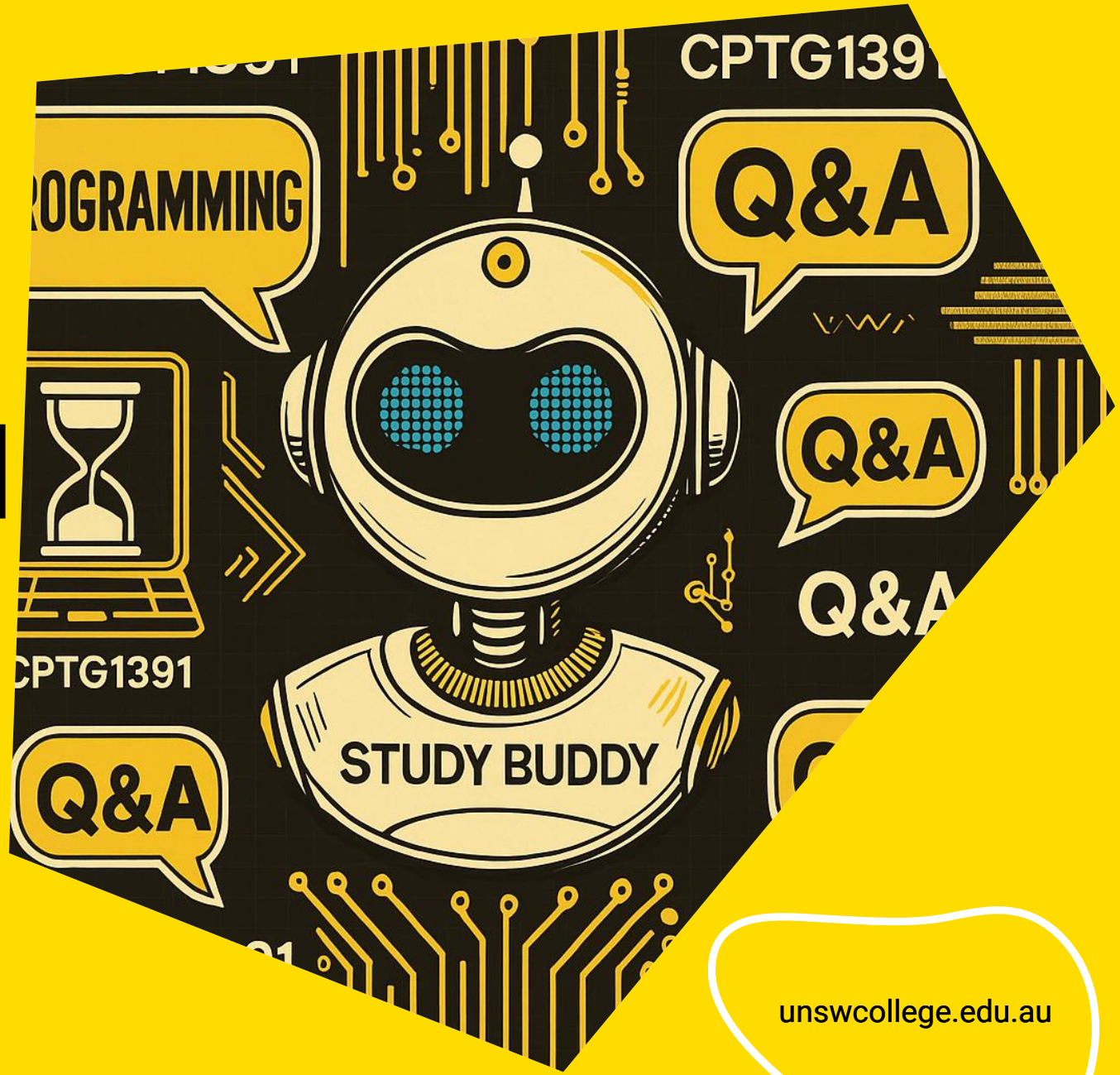
Week 1 – Lecture 2

Lecturer and Course Convener:

Dr Pantea Aria

UNSW College

Variables and Constants



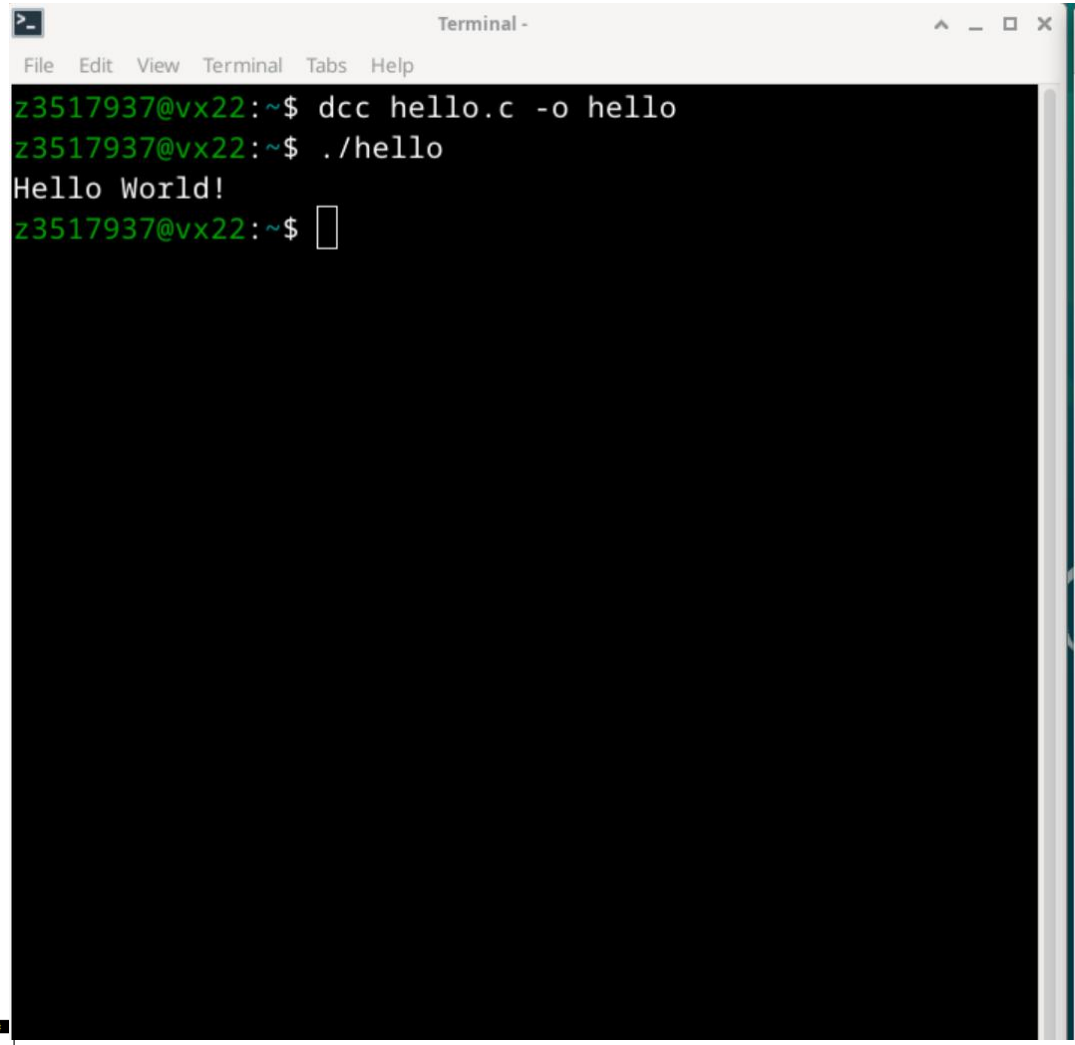
unswcollege.edu.au

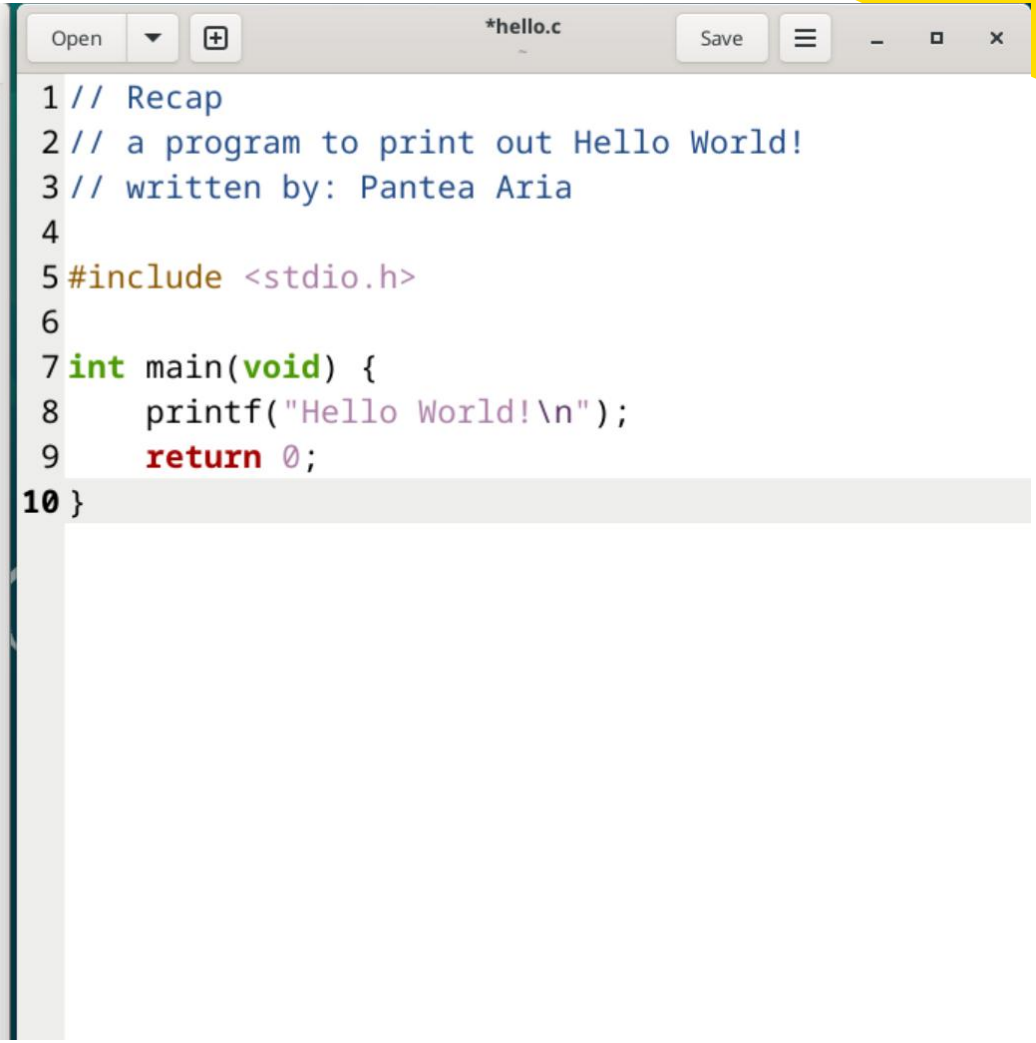
Agenda

- **So far**
  - Welcome
  - Vlab and hello world

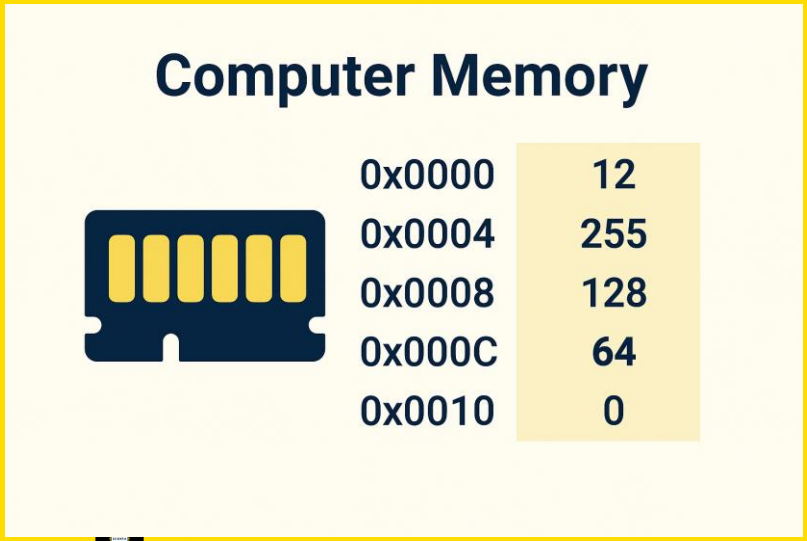
- **Today**
  - Variables
  - Constants
  - Maths in C

So far, our first program





What part of a computer keeps information so it can be used when needed?



Computer memory is like a large collection of tiny on-off switches.

Each switch is called a **bit**, which can be either 0 or 1.

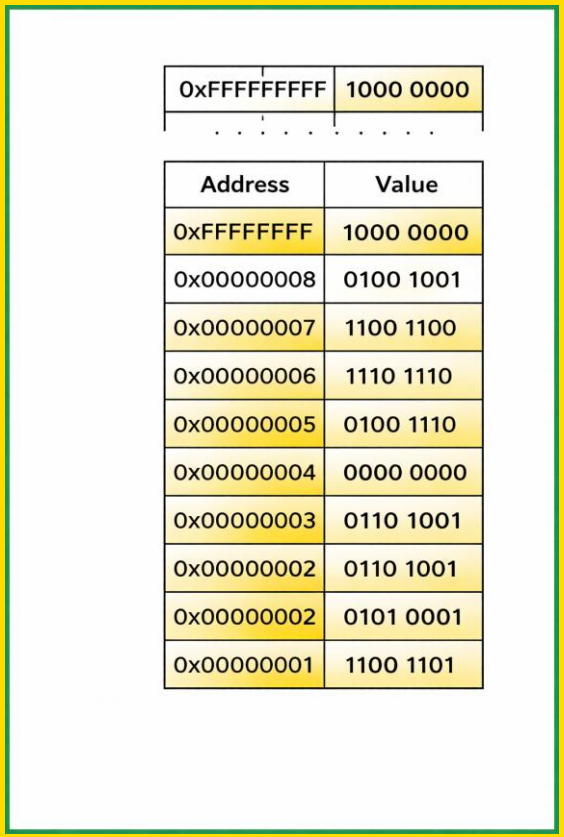
We usually group **8 bits together** to make a **byte**.

Memory

When code runs, the CPU handles the instructions and calculations, and the RAM holds all the data needed for those tasks.

How can we store data in the memory?

Variable



→ **Variables** are used to **store a value somewhere in the memory.** Values are **stored** as **binary**.

→ The value a variable holds **may change** over its lifetime

→ At any point in time a variable stores **one value** (except quantum computers!)

→ C variables have a **type**

→ We'll only use **3 standard types** of variable for the next few weeks:

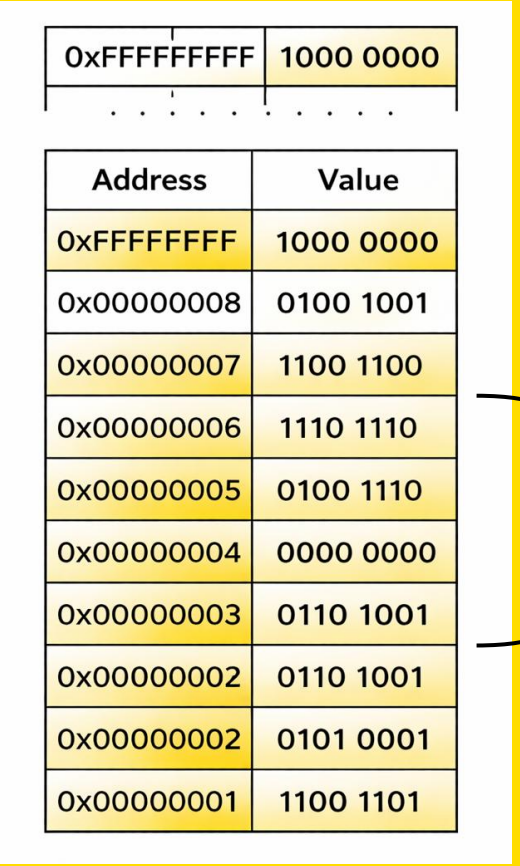
Data Type	What it stores	Examples
int	Integer (whole numbers)	42, -1
double	Decimal (real numbers)	3.14159, 2.71828
char	Single character	'A', '+'

Naming a Variable

Rule	Explanation	Example
Use meaningful names	Variable names should describe what the variable stores	temperature, max_speed (not t, ms)
Start with lowercase	Variable names start with lowercase letters	total_value
C is case-sensitive	Names with different capitalization are different variables	totalValue ≠ totalvalue
Avoid C keywords	Reserved words cannot be used as variable names	for, while, char
Use snake_case	For multiple words, use underscores	student_score

https://cgi.cse.unsw.edu.au/~dp1091/26T1/resources/style_guide.html

int



1 integer number
32 bits (4 bytes)

An **int** variable typically uses **4 bytes** of memory.

4 bytes = 32 bits → **2^32 bit** patterns.

Only **2^32 integers** can be represented:

- Range: **-2^31 to 2^31 - 1**

- Decimal: **-2,147,483,648 to +2,147,483,647**

char

```
z3517937@vx22:~$ ascii -d
   0 NUL    16 DLE    32      48 0    64 @    80 P    96 `   112 p
   1 SOH    17 DC1    33 !    49 1    65 A    81 Q    97 a   113 q
   2 STX    18 DC2    34 "    50 2    66 B    82 R    98 b   114 r
   3 ETX    19 DC3    35 #    51 3    67 C    83 S    99 c   115 s
   4 EOT    20 DC4    36 $    52 4    68 D    84 T   100 d   116 t
   5 ENQ    21 NAK    37 %    53 5    69 E    85 U   101 e   117 u
   6 ACK    22 SYN    38 &    54 6    70 F    86 V   102 f   118 v
   7 BEL    23 ETB    39 '    55 7    71 G    87 W   103 g   119 w
   8 BS     24 CAN    40 (    56 8    72 H    88 X   104 h   120 x
   9 HT     25 EM     41 )    57 9    73 I    89 Y   105 i   121 y
  10 LF     26 SUB    42 *    58 :    74 J    90 Z   106 j   122 z
  11 VT     27 ESC    43 +    59 ;    75 K    91 [   107 k   123 {
  12 FF     28 FS     44 ,    60 <    76 L    92 \   108 l   124 |
  13 CR     29 GS     45 -    61 =    77 M    93 ]   109 m   125 }
  14 SO     30 RS     46 .    62 >    78 N    94 ^   110 n   126 ~
  15 SI     31 US     47 /    63 ?    79 O    95 _   111 o   127 DEL
z3517937@vx22:~$
```

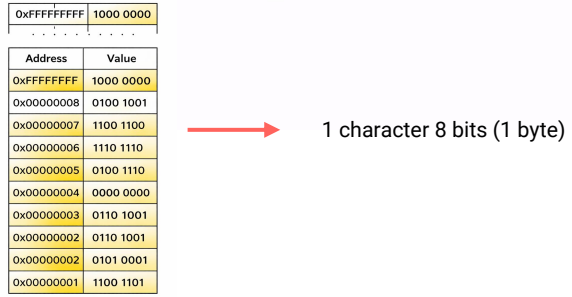
A single character in C can also be stored as an integer.

- This is because characters are represented by ASCII values (numbers from 0 to 127).
- The character itself is written using single quotes, for example 'a'.
- Each character has a matching integer value (e.g. 'A' is stored as 65).

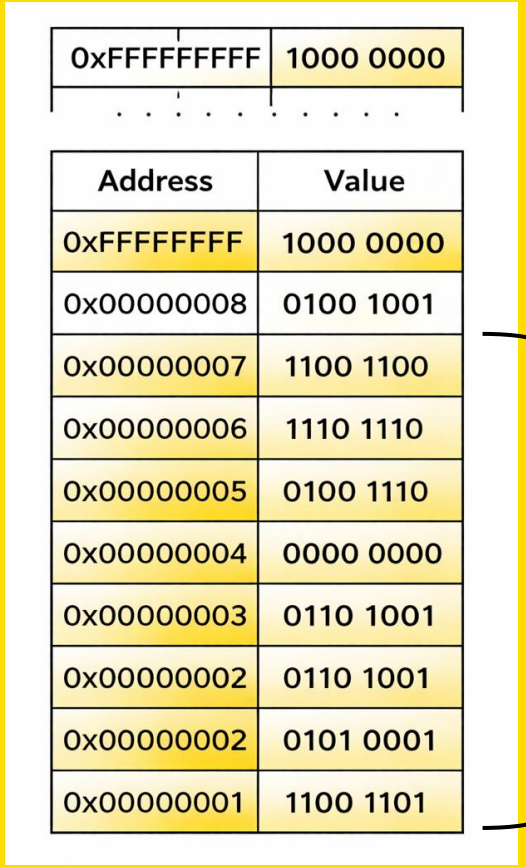
Char	ASCII
a	97
A	65
+	43

Characters are declared using the **char** type.

Example: `char letter = 'a';` stores the value 97 in the variable **letter**.



double



0xFFFFFFFF	1000 0000
. . . . .	. . . . .

Address	Value
0xFFFFFFFF	1000 0000
0x00000008	0100 1001
0x00000007	1100 1100
0x00000006	1110 1110
0x00000005	0100 1110
0x00000004	0000 0000
0x00000003	0110 1001
0x00000002	0110 1001
0x00000002	0101 0001
0x00000001	1100 1101

1 double number
64 bits (8 bytes)

A **double** is a number that can have decimals (like 3.14 or 10.5).

It's called a **floating-point number** because the decimal point can "float"

it can appear in different places depending on the size of the number. Examples: **10.567** and **105.67** use the same digits, but the decimal point is in a different spot.

It's called a **double** because it uses **64 bits** of memory, which is **twice as much** as a normal integer (32 bits).

Variables syntax:

```
<type> <name>;
```

```c
// Week 1 Lecture 2
// variables
// written by: Pantea Aria

#include <stdio.h>

int main(void) {

    // Declare an integer variable
    int first;
    // initialise variable first
    first = 10;

    // declare a character variable
    char ch;
    // initialise variable ch
    ch = 'f';

    // Declare an integer and store a whole number
    int number = 25;

    // Declare a character and store a single character
    char letter = 'A';

    // Declare a double and store a decimal number
    double value = 3.14;

    return 0;
}
```

What can we do with those variables?

Output using printf()

⟶ We can **print** variables to our terminal!

⟶ We describe the **format** of how we want text printed, then the **actual values**.

⟶ To print out a variable value, we use **format specifiers** with printf

```
int j = 5;
int k = 17;
printf("j is %d and k is %d\n", j, k);
```

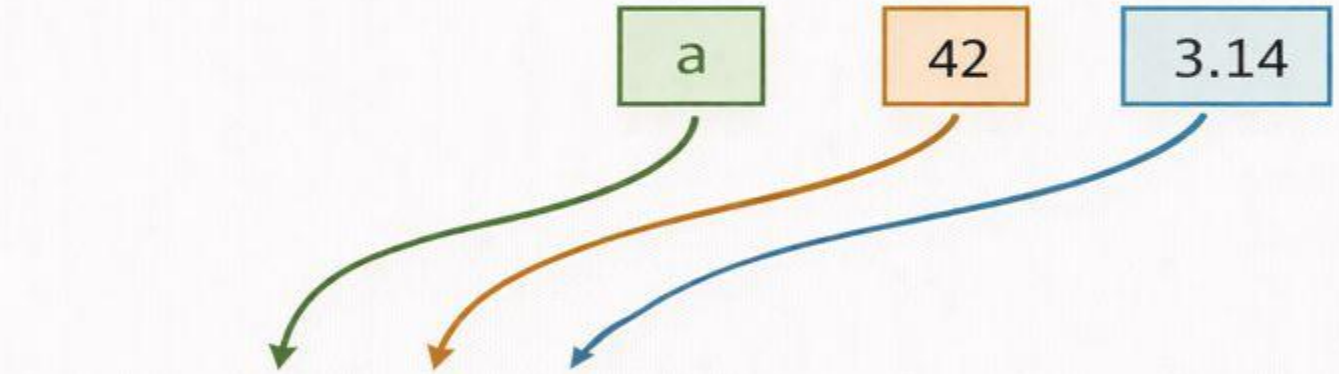
format specifiers

The format specifier must match the data types passed

%c for chars

%d for ints "decimal integer"

%lf for "long floating point number" (a double)



Interesting with chars

```c
char letter = 'A';
printf("%c", letter); // prints A

printf("%d", 'a'); // prints 97
printf("%d", 'A'); // prints 65
printf("%d", '0'); // prints 48
printf("%d", ' ' + '\n'); // prints 42 (32 + 10)
```

Demo

→Variables.c

Live lecture code is written for teaching, not perfection.
It may include extra comments and may not always follow ideal coding style

UNSW College


IT'S BREAK TIME!
#include <stdio.h>
#define ON_BREAK 1
int main(){
// Time for a 10 minute break! Switch to PARTY_MODE
#define PARTY_MODE ON_BREAK
if (PARTY_MODE == ON_BREAK) ;
print("Program will resume in 10 minutes...");
sleep(600);
// Take a break
exit(0);
}
10 MINUTES BREAK!
BRB
CHIPS
Relax... We'll be back soon!

Input using scanf()

- The `&` symbol tells scanf where to store the data (more details later in term)

```
// Use %d to read an int (integer) value
int answer;
printf("Enter the answer: ");
scanf("%d", &answer);


// Use %lf to read a double (floating point) value
double e;
printf("Enter e: ");
scanf("%lf", &e);


char ch;
printf("Enter a character: ");
// scanning a char does not ignore whitespace
// We can ignore leading whitespace with chars:
scanf(" %c", &ch);
```

Demo

→scanf.c

Live lecture code is written for teaching, not perfection.
It may include extra comments and may not always follow ideal coding style

UNSW College

Constants

```
#define <NAME> <value>
```

→ A value that will **never change**

→ More efficient to store a constant (less memory)

→ We use **UPPERCASE** to signify it's a **constant**

→ **#define** statements go at the top of your program after **#include** statements

```c
#include <stdio.h>


#define MAX_STUDENTS 30
#define PI 3.14159


int main() {
    printf("Welcome to the Student Management System!\n");
    printf("Maximum number of students allowed: %d\n", MAX_STUDENTS);
    printf("Value of PI: %.5f\n", PI);
    return 0


}
```

Demo

→Constants.c

Live lecture code is written for teaching, not perfection.
It may include extra comments and may not always follow ideal coding style

UNSW College

Arithmetic Operators

C supports the usual **maths** operations: **+ − ∗ /**

Precedence is as you would expect from high school,
e.g.: **a + b ∗ c + d/e ⇒ a + (b ∗ c) + (d/e)**

Associativity (grouping) is as you would expect from high school,
e.g.: **a − b − c − d ⇒ ((a − b) − c) − d**

*Beware **division** may not do what you expect*

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

Example:

```c
#include <stdio.h>

int main(void) {

    int a = 10;
    int b = 3;


    // Add a and b
    int sum = a + b;

    // Subtract b from a
    int diff = a - b;

    // Multiply a and b
    int product = a * b;

    // Divide a by b (integer division)
    int quotient = a / b;

    // Find the remainder when a is divided by b
    int remainder = a % b;

    // Print the values
    printf("a = %d, b = %d\n", a, b);
    printf("Sum: %d\n", sum);
    printf("Difference: %d\n", diff);
    printf("Product: %d\n", product);
    printf("Quotient: %d\n", quotient);
    printf("Remainder: %d\n", remainder);

    return 0;
}
```

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** .

$2.6/2 \Rightarrow 1.3$ (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).

$5/3 \Rightarrow 1$ (not 2)

C also has the **%** operator (integers only), computes the **modulo** (**remainder after division**)

$14 \% 3 \Rightarrow 2$

Example

```c
#include <stdio.h>

int main(void) {
    int a = 10;
    int b = 3;

    // Integer division:
    // Both operands are int, so the result is an int.
    // The decimal part is discarded.
    // 10 / 3 = 3.333...  → result becomes 3
    int int_result = a / b;

    // Double division:
    // Using double variables, so the division keeps the decimal part.
    double x = 10.0;
    double y = 3.0;

    // 10.0 / 3.0 = 3.333...
    double double_result = x / y;

    // Display results
    printf("Integer division result: %d\n", int_result);
    printf("Double division result: %.2f\n", double_result);

    return 0;
}
```

Demo

→Arithmetic.c

Live lecture code is written for teaching, not perfection.
It may include extra comments and may not always follow ideal coding style

UNSW College

Mathematical functions

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
```

Example

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

int main(void) {
    double number = 16.0;
    double base = 2.0;
    double exponent = 3.0;

    // Calculate the square root of number
    // sqrt(16.0) = 4.0
    double square_root = sqrt(number);

    // Calculate base raised to the power of exponent
    // pow(2.0, 3.0) = 8.0
    double power_result = pow(base, exponent);

    // Display results
    printf("Square root of %.1f is %.1f\n", number, square_root);
    printf("%.1f raised to the power of %.1f is %.1f\n",
           base, exponent, power_result);

    return 0;
}

```

Voice of the Student

Anonymous ongoing feedback
Anything you wanted to share with me

Voice of the Student

26T1 Voice of the Student – Fill out form

See you soon …

UNSW College