

DPST1091 / CPTG1391

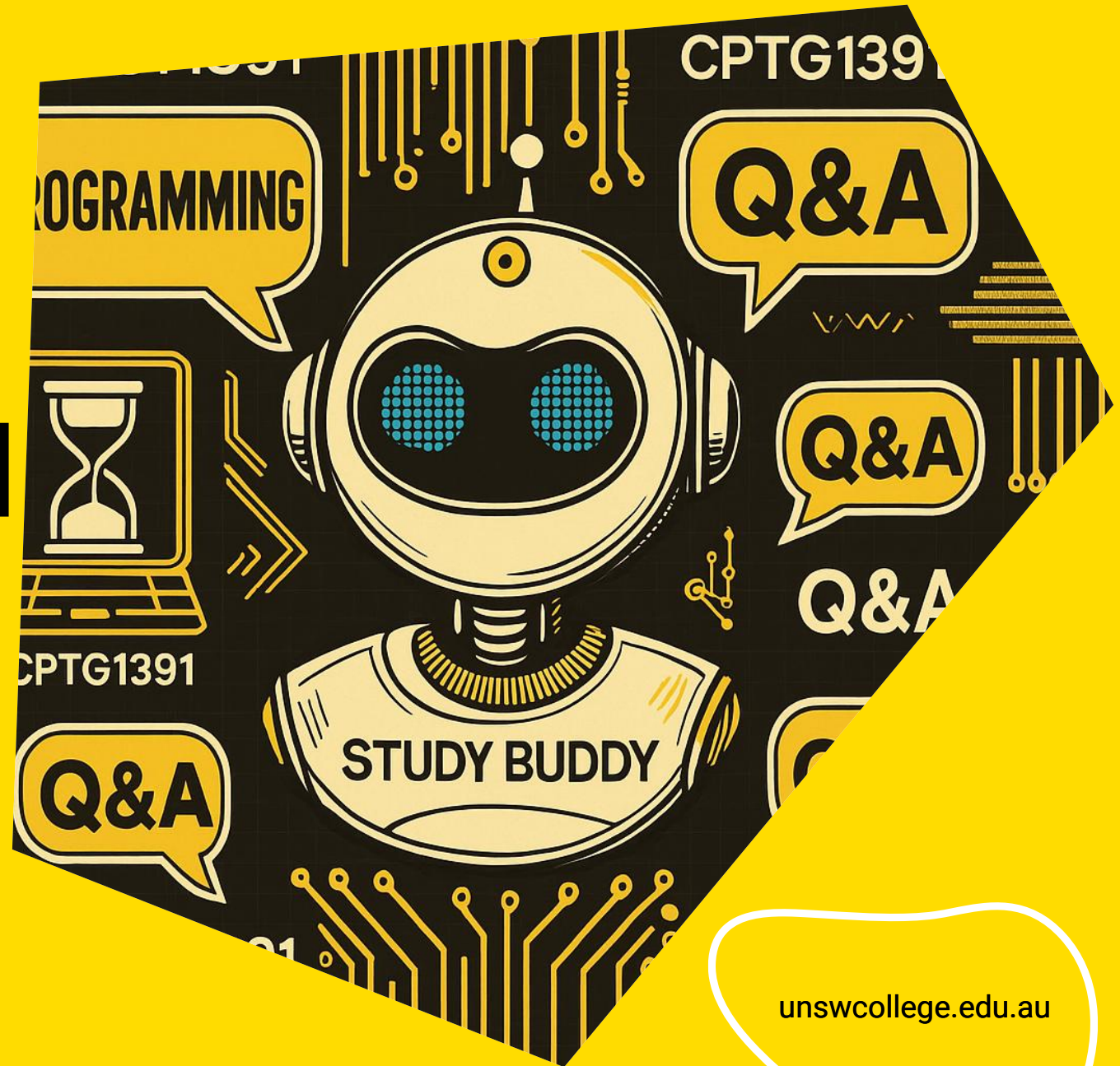
Introduction to Programming

Week 1 – Lecture 2

Lecturer and Course Convener:

Dr Pantea Aria

Variables and Constants



Agenda

- **So far**

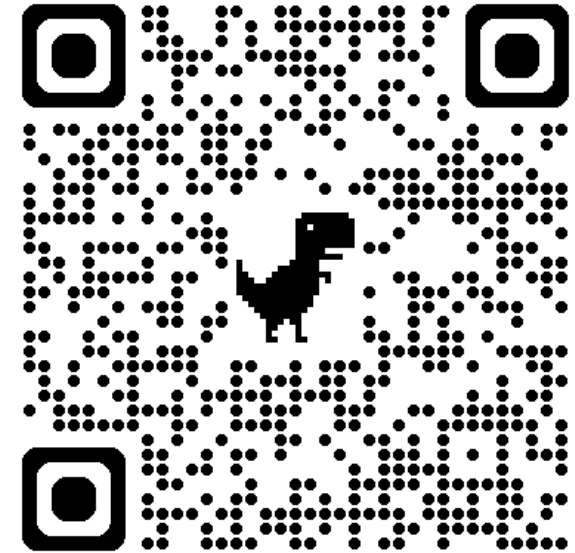
- Welcome
- Vlab and hello world

- **Today**

- Variables
- Constants
- Maths in C

Link to the Live Lecture Code

<https://cgi.cse.unsw.edu.au/~dp1091/26T2/code/01/>



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

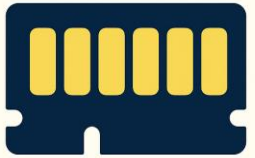
So far, our first program

```
Terminal -
File Edit View Terminal Tabs Help
z3517937@vx22:~$ gcc hello.c -o hello
z3517937@vx22:~$ ./hello
Hello World!
z3517937@vx22:~$
```

```
*hello.c
Open Save
1 // Recap
2 // a program to print out Hello World!
3 // written by: Pantea Aria
4
5 #include <stdio.h>
6
7 int main(void) {
8     printf("Hello World!\n");
9     return 0;
10 }
```

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

Computer Memory



0x0000	12
0x0004	255
0x0008	128
0x000C	64
0x0010	0

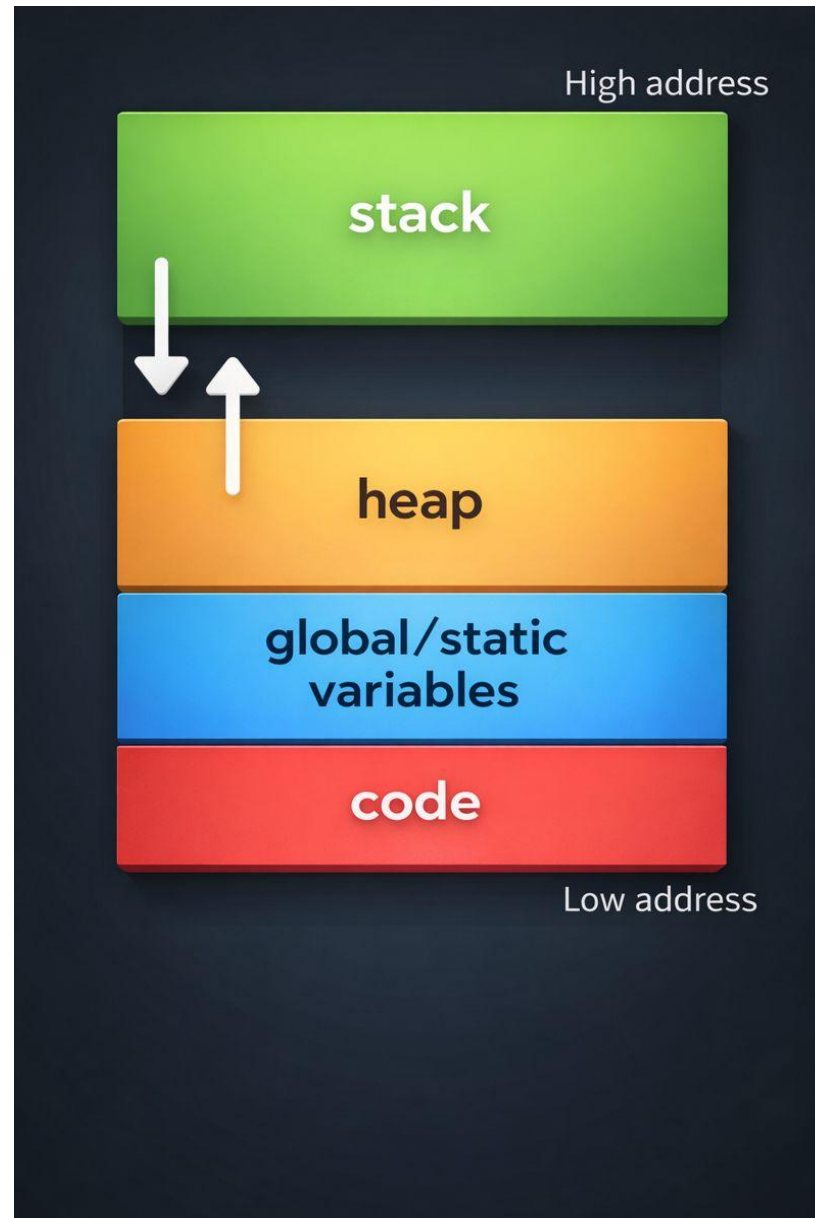
Computer memory is like a large collection of tiny on-off switches.

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

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

- **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/~dp1091/26T2/resources/style_guide.html

int

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 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

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.

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
0x00000001	0101 0001
0x00000000	1100 1101

1 character 8 bits (1 byte)



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>;|
```

```
// 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;
```

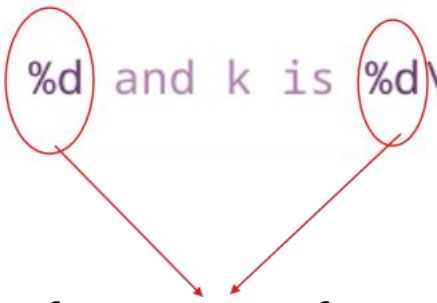
What can we do with those variables?

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

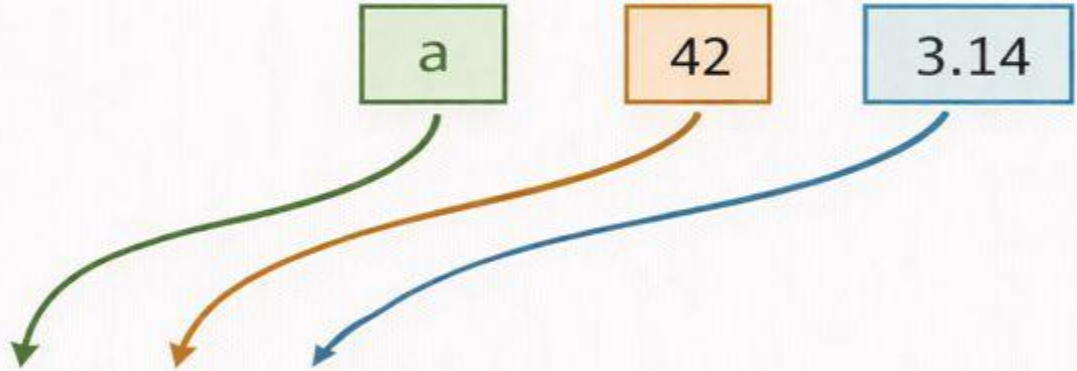


The format specifier must match the data types passed

`%c` for chars

`%d` for ints "decimal integer"

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



The diagram illustrates the mapping of variable values to format specifiers in a `printf` statement. Three colored boxes at the top contain the values: a green box with 'a', an orange box with '42', and a blue box with '3.14'. Colored arrows point from these boxes to the corresponding format specifiers in the code below: a green arrow from 'a' to `%c`, an orange arrow from '42' to `%d`, and a blue arrow from '3.14' to `%lf`.

```
printf("%c %d %lf", letter, number, amount);
```

Interesting with chars

```
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

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

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

```
#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

Arithmetic Operators

C supports the usual **maths** operations: **+** **-** ***** **/**

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

Associativity (grouping) is as you would expect from high school,
e.g.: $a - b - c - d \Rightarrow ((a - b) - c) - d$

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

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

Example:

```
1#include <stdio.h>
2
3int main(void) {
4
5    int a = 10;
6    int b = 3;
7
8
9    // Add a and b
10   int sum = a + b;
11
12   // Subtract b from a
13   int diff = a - b;
14
15   // Multiply a and b
16   int product = a * b;
17
18   // Divide a by b (integer division)
19   int quotient = a / b;
20
21   // Find the remainder when a is divided by b
22   int remainder = a % b;
23
24   // Print the values
25   printf("a = %d, b = %d\n", a, b);
26   printf("Sum: %d\n", sum);
27   printf("Difference: %d\n", diff);
28   printf("Product: %d\n", product);
29   printf("Quotient: %d\n", quotient);
30   printf("Remainder: %d\n", remainder);
31
32   return 0;
33}
```

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**.

$$2.6/2 \Rightarrow 1.3 \text{ (not 2)}$$

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).

$$5/3 \Rightarrow 1 \text{ (not 2)}$$

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

$$14 \% 3 \Rightarrow 2$$

Example

```
1 #include <stdio.h>
2
3 int main(void) {
4     int a = 10;
5     int b = 3;
6
7     // Integer division:
8     // Both operands are int, so the result is an int.
9     // The decimal part is discarded.
10    // 10 / 3 = 3.333... → result becomes 3
11    int int_result = a / b;
12
13    // Double division:
14    // Using double variables, so the division keeps the decimal part.
15    double x = 10.0;
16    double y = 3.0;
17
18    // 10.0 / 3.0 = 3.333...
19    double double_result = x / y;
20
21    // Display results
22    printf("Integer division result: %d\n", int_result);
23    printf("Double division result: %.2f\n", double_result);
24
25    return 0;
26 }
```

Demo

→ Arithmetic.c

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)

gcc includes maths library by default. Most compilers need extra option:

gcc needs `-lm` e.g.:

```
gcc -o heron heron.c -lm
```

Example

```
1#include <stdio.h>
2#include <math.h>
3
4int main(void) {
5    double number = 16.0;
6    double base = 2.0;
7    double exponent = 3.0;
8
9    // Calculate the square root of number
10   // sqrt(16.0) = 4.0
11   double square_root = sqrt(number);
12
13   // Calculate base raised to the power of exponent
14   // pow(2.0, 3.0) = 8.0
15   double power_result = pow(base, exponent);
16
17   // Display results
18   printf("Square root of %.1f is %.1f\n", number, square_root);
19   printf("%.1f raised to the power of %.1f is %.1f\n",
20         base, exponent, power_result);
21
22   return 0;
23 }
24
```

What did we learn today?

Category	Details
Memory & Data	Data stored in RAM using bits (0/1); 8 bits = 1 byte; CPU processes instructions while memory stores data
Variables	Store values in memory; each has a type and holds one value; values can change
Data Types	<code>int</code> (whole numbers) • <code>double</code> (decimals) • <code>char</code> (single characters)
Variable Naming	Meaningful names • lowercase start • <code>snake_case</code> • avoid keywords
Output (<code>printf</code>)	Displays values; uses format specifiers: <code>%d</code> (int), <code>%lf</code> (double), <code>%c</code> (char)
Input (<code>scanf</code>)	Reads user input; uses <code>&</code> to store values (e.g., <code>scanf("%d", &num);</code>)
Constants	Fixed values using <code>#define</code> ; written in UPPERCASE (e.g., <code>#define MAX 100</code>)
Arithmetic	Operators: <code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>%</code> ; follows standard order of operations
Division	Integer division drops decimals ($5/3 = 1$); double division keeps decimals ($5.0/3 \approx 1.67$)
Math Functions	Use <code><math.h></code> for functions like <code>sqrt()</code> , <code>pow()</code> , <code>sin()</code>

Voice of the Student

Anonymous ongoing feedback
Anything you wanted to share with me



Voice of the Student



[Voice of the Student – Fill out form](#)

See you soon ...