C

Introduction to C

What is C?

C is a general purpose programming language developed in 1972. Almost all modern day languages are inspired by C or built WITH C (the compiler used for python was built in C).

Advantages of C

- **High performance and efficiency**
- **Small footprint**
- **Access to Low-Level System Resources**

Disadvantages of C

- **High complexity**
- **Manual memory management**
- **Slower development time**

How human written code is read by a computer

Computers only understand binary instructions, so humans write code in a **readable** language like **C** which is converted to machine code (binary code) by a **compiler**

General syntax

Example code snippet

Take a look at this piece of code, it outputs the sentence "Hello World".

```c
#include <stdio.h>

int main(){
        printf("Hello World");
        return 0;
}
// OUTPUT : Hello World
```

Line by line explanation :

1. `#include <stdio.h>` : This tells the compiler to **include** the *stdio.h* file. This file contains most general functions needed to perform actions in C. Almost all C programs start with this line.
2. `BLANK LINE` : The second line is blank, this line is ignored by the compiler . People include blank lines to make code more readable so this is optional.
3. `int main(){` : Another thing that always appear in a C program is `main()`. This is called a **function**. Any code inside its curly brackets `{}` will be executed.
4. `printf("Hello World");` : The **printf** function is used to output values to the screen. Anything inside the brackets is shown on the output terminal. The semicolon shows that the command ends there. All lines in C must end with a semicolon.
5. `return 0` : This line is at the end of the function. This is used to check if the code had any issues or not. If the code did not have any problems then it will output 0.
6. `}` : The closing bracket shows that the function has ended.
7. `// Terminal : Hello World` : This is called a **comment** , comments are ignored by the computer as they are only there to make the code easier to understand for someone else other than the programmer. All comments start with a `//` or a `/**/` if you want a multi line comment. Here the comment is just used to show what the output would have been if it were run in a compiler.
   Each line inside a function is called a **statement**, all statements need to end with a semicolon. The computer executes the statements line by line, so the statement at the top is executed first.

Escape sequences

Escape Sequence	Purpose/ Description	HEX Value in ASCII	Syntax
\a	*Alert or bell:* Produces an audible alert or bell sound.	0x07	printf("Hello \a World");
\b	*Backspace:* Moves the cursor back by one position.	0x08	printf("Hello\bWorld");
\f	*Form feed:* Advances the printer to the next logical page.	0x0C	printf("Hello\fWorld");
\n	*Newline:* Moves the cursor to the beginning of the next line.	0x0A	printf("Hello\nWorld");
\r	*Carriage return:* Moves the cursor to the beginning of the current line.	0x0D	printf("Hello\rWorld");
\t	*Horizontal tab:* Moves the cursor to the next horizontal tab stop.	0x09	printf("Hello\tWorld");
\v	*Vertical tab:* Moves the cursor to the next vertical tab stop.	0x0B	printf("Hello\vWorld");

Escape Sequence	Purpose/ Description	HEX Value in ASCII	Syntax
\	*Backslash:* Inserts a backslash character.	0x5C	printf("Hello\World");
\’	*Single quote:* Inserts a single quote character.	0x27	printf("Hello'World");
\”	*Double quote:* Inserts a double quote character.	0x22	printf("Hello"World");
?	*Question mark:* Inserts a question mark character.	0x3F	printf("Hello?World");
\0	*Null character:* Inserts a null character.	0x00	char str[] = "Hello\0World";
\nnn	*Octal value:* Inserts the character represented by the octal value nnn.	0x00 to 0xFF	printf("Hello \072 World");
\xhh	*Hexadecimal value:* Inserts the character represented by the hexadecimal value hh.	0x00 to 0xFF	printf("Hello\x41World");
\uhhhh	*Universal character name:* Inserts the character represented by the Unicode value hhhh.	None	printf("Hello\u03A9World");

Preprocessor keywords

Use of `#include`

`#include` is used to import libraries with pre-programmed functions that are essential for c , a common library is `<stdio.h>` which contains functions like `printf()` and `scanf()`

Use of `#define`

`#define` is used to define constant values outside the main function. This is done to decrease compiler time by defining values outside of the code instead of declaring it inside. For example

Data types and variables

Types of Data

The main 3 types of variables used in C are:

- `int` - stores integers (whole numbers) // 4 bytes // -2^{32} ~ 2^{32}

- `float` - stores floating point numbers// 4 bytes // 1.2E-38 ~ 3.4E+38
- `double` - stores floating point numbers with higher precision // 8 bytes // 2.3E-308 ~ 1.7E+308
- `char` - stores a single character // 1 byte // -128 ~ 127 OR 0 ~ 255
- `bool` - `true` or `false` values only (must use `#include <stdbool.h>` in the top of the code to use booleans) // 1 byte
- `char string[] = "Hello World"` : Arrays store a collection of variables/constants. Here an array is used to store a collection of `char` type variables
  The 7th data type is a constant which is denoted with the `const` variable. Once you state a constant variable you cannot change it

Declaration and Initialization

Before we can use a variable we must do two things, **declaration** and **initialization**.
You HAVE to declare ALL variables before any statements in the code.

1. Declaration : First we need to ***declare*** to the compiler the **name** of the variable and the **type** of the variable. The variable currently has a value of **undefined** as we haven't stored a value for it yet.
2. Initialization : After declaration we need to initialize the value of the variable. This stores a user-defined value for the variable.

```
// Declaration
int myNumber; // Declares a integer variable with the name myNumber
float myFloat; // Declares a floating variable with the name myFloat
char myChar; // Declares a character variable with the name myChar
char myWord[]; // Declares a string variable with the name myWord
// Initialization
myNumber = 7; // Assigns a integer value to the variable
myFloat = 9.11; // Assigns a floating value to the variable
myChar = 'c'; // Assigns a single character to the variable
myWord = "Hello World"; // Assigns a word to the array
```

The Declaration and initialization step can be merged into one step

```
int myNumber = 7; // Declares and assigns a value to the variable called myNumber
```

You can also declare multiple variables of the same type in the same line

```
int x = 1, y = 2, z = 3;
```

You can also change the value of a variable as many times as you want, or assign the value of one variable to another variable

```c
int x = 1; // Declares and assigns a value
int y = 2; // Declares and assigns a value

x = 3; // Changes value of x from 1 to 3
x = y; // Changes value of x from 3 to value of y (2)
printf("%d", x);
/*
OUTPUT
------
2
*/
```

The `%d` is called a format specifier and it is used to print the values of variables.

Format specifiers

This is how to print variables

```c
int myNum = 7
float myFloat = 7.23
char myChar = 'n'
printf("%d %f %c", myNum, myFloat, myChar);
/*
OUTPUT
------
7 7.23 n
*/
```

Here the `%d` is a **format specifier** , this specifies that the variable to output is a integer. Each variable type has a unique format specifier.

- `%d` - int, bool
- `%u` - unsigned int
- `%o` - unsigned octal
- `%x / %X` - unsigned hexadecimal
- `%f` - float
- `%e` - floating point with E notation
- `%c` - char
- `%s` - string
  the order of appearance of the format specifier should match the order of appearance of the variable name inside the `printf()` function.

Real life example

The following snippet of code is an example of the cases where declaring, initializing and then printing the variable with format specifiers is useful.

```c
//Student data
int studentID = 2411371042;
int studentAge = 18;
char studentDept[] = "CSE";
char studentName[] = "Fahim Muntasir";
//Print the data
printf("Name: %s\nAge: %d\nID: %d\nDepartment: %s\n", studentName, studentAge,
studentID, studentDept);
/*
OUTPUT
------
Name: Fahim Muntasir
Age: 18
ID: 2411371042
Department: CSE
*/
// Each line is on the next line because of the usage of the \n escape sequence
```

Operators

Arithmetic Operators

Operator	Type	Function	Example
`+`	Addition	Add two values	`x + y`
`-`	Subtraction	Subtract two values	`x - y`
`*`	Multiplication	Multiply two values	`x * y`
`/`	Division	Divide two values	`x / y`
`%`	Modulus	Returns the division remainder	`x % y`
`++`	Increment	Increase value by 1	`++x`
`--`	Decrement	Decrease value by 1	`--x`

Assignment Operators

Operator	Syntax	Explained
`=`	`x = 5`	`x = 5`
`+=`	`x += 5`	`x = x + 5`
`-=`	`x -= 5`	`x = x - 5`

Operator	Syntax	Explained
`*=`	`x *= 5`	`x = x * 5`
`/=`	`x /= 5`	`x = x / 5`
`%=`	`x %= 5`	`x = x % 5`

Comparison Operators

Operator	Type
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`<`	Lesser than
`>=`	Greater than or equal to
`<=`	Lesser than or equal to

Comparison operators output `1` if true and `0` if false

Bitwise Operators

Operator	Type
`&`	bitwise AND
|`	bitwise OR
`~`	bitwise NOT
`^`	bitwise XOR
`<<`	shift left
`>>`	shift right

Logical Operators

Operator	Type
`&&`	AND
||	OR
`!`	NOT

Operator precedence

Example

```c
#include <stdio.h>
int main() {
        int result = 5 + 3 * 2 - 4 / 2;
        printf("Result: %d\n", result);
        return 0;
}
/*
OUTPUT
------
Result: 9
*/
```

1. `3 * 2` is evaluated first, resulting in `6`.
2. `4 / 2` is evaluated next, resulting in `2`.
3. Then, the result of the multiplication (`6`) is added to `5`, resulting in `11`.
4. Finally, `11` is subtracted by `2`, resulting in a final `9`.

Order of operators

1. Function calls (printf(), scanf(), etc.)
2. !, ++, --
3. , /, %
4. +, -
5. == , >= , <= , >, <, !=
6. &&
7. | |
8. =

if...else statements

If statements

If statements are useful for decision making.
Format for an if statement:

```c
if (condition){
        code that will execute only if the condition is true
}
```

Example

```
int age = 26;
if (age > 18){
        printf("You are a %d year old adult", age);
}
/*
OUTPUT
------
You are a 26 year old adult
*/
```

Else statements

Else statements are used when the condition in the if statement is not true
Formant for an else statement:

```
if(condition){
        code that will execute if condition is true
} else {
        code that will execute if condition is false
}
```

Example

```
int age = 16;
if (age > 18){
        printf("You are an adult");
} else {
        printf("You are not an adult);
}
/*
OUTPUT
------
You are not an adult
*/
```

Else if statements

Else if statements are used to check for another condition if the previous condition was false
Format for else if statements:

```
if(condtion 1){
        code that will execute if condtion 1 is true
} else if(condtion 2){
        code that will execute if condtion 2 is true
} else {
```

```
        code that will execute when all conditions are false
}
```

Example

```
int age = -2;
if(age > 18){
        printf("You are an adult");
} else if(age < 0){
        printf("Invalid age, age cannot be negative");
} else {
        printf("You are not an adult);
}
/*
OUTPUT
------
Invalid age, age cannot be negative
*/
```

Shorthand if statement

This method is used for small and simple statements that can be written in a single line
Format for shorthand if statements

```
(condtion) ? code that will execute if true : code that will execute if false;
```

Example

```
int age = 21;
(age > 18) ? printf("Adult") : printf("Not an adult");
```

Switch case statements

Switch case statements are used instead of using many lines of if statements. Using it is optional, If statements and switch case statements have the same function
Format

```
switch(variable/expression){
        case a:
                //code
                break;
        case b:
                //code
                break;
        default:
```

```
                //code
}
```

The default block can be though of as an else statement

Loops

While loop

The while loop repeats a piece of code as long as a condition is true
Format:

```
while(condition){
        //code to be executed
}
```

Example

```
int i = 0; // this variable is declared so that it can be used to count the loops
while(i < 5){
        printf("%d\n", i); // outputs the current value of i
        i++; // every loop i is increase by one so that the code is not looped
forver
}
/*
OUTPUT
------
0
1
2
3
4
*/
```

Do/While loop

Do while loop is almost the exact same thing as the while loop, the only difference is that the loop first executes code then checks the condition
Format:

```
do {
        // code to be executed
} while (condition);
```

For loop

For loops repeat code a fixed amount of times unlike the while loop which can loop forever;
Format:

```
for (expression 1; expression 2; expression 3){
        //code to be executed
}
```

Expression 1 : code that is executed one time before the loop
Expression 2 : condition for running the loop
Expression 3 : code that is executed after every loop

Example:

```
for(int i = 0; i < 5; i++){
        printf("%d\n", i);
}
/*
OUTPUT
------
0
1
2
3
4
*/
```

Break/Continue

The `break` statement is used to break out of a loop;
Example:

```
for(int i = 0; i < 10; i++){
        if(i == 4){
                break;
        }
        printf("%d\n", i);
}
/*
OUTPUT
------
0
1
2
3
*/
// loop stops when i = 4
```

The `continue` statement is used to skip one iteration of the loop;
Example:

```
for(int i = 0; i < 10; i++){
        if(i == 4){
                continue;
        }
        printf("%d\n", i);
}
/*
OUTPUT
------
0
1
2
3
5
6
7
8
9
*/
// loop skips the step of printing 4 and goes onto the next iteration
```

Arrays

Arrays are used to store a collection of variables of the same type
The declaration and initialization of arrays is identical to variables ;

```
int myArray[] = {25, 62, 91, 11};
```

Each element of an array has an index, the first element of an array has an index of 0.

```
int myArray[] = {25, 62, 91, 11};
printf("%d", myArray[2])
/*
OUTPUT
------
91
*/
```

You can use loops to list all elements of an array ;

```
int myArray[] = {25, 62, 91, 11};
for(int i = 0; i < 4; i++){
        printf("%d ", myArray[i]);
```

```
}
/*
OUTPUT
------
25 62 91 11
*/
```

To know the size of an array we can use the `sizeof()` method. This outputs the number of bytes of memory that is being used by the array.

```
int myNumbers[] = {10,25,50,75,100};
int size = sizeof(myNumbers);

printf("%lu", size);
/*
OUTPUT
------
20 (20 bytes)
*/
```

When we want to know the number of elements in an array we can divide the (total memory taken by the array) by (the memory taken by one element).

```
int myNumbers[] = {10,25,50,75,100};
int length = sizeof(myNumbers)/sizeof(myNumbers[0]);

printf("%d", length);
/*
OUTPUT
------
5
*/
```

Strings

To use string functions you need to include the strings header file by using `#include <string.h>`
Some common functions:

- `strlen()` = returns length of string
- `strcat(str1, str2)` = adds str2 to the end of str1 and the result is stored in str1
- `strcpy(str2, str1)` = copies value of str1 to str2
- `strcmp(str1, str2)` = compares two st1 and str2 ; returns 0 if same

User input

`printf()` is used to show output. Similarly, `scanf()` is used to take user input.
Format:

```
int myNum;
scanf("%d", &myNum);
// use a format specifier to define type of input and then declare the variable
that will be assigned to that value
```

Memory addresses

Preceding any variable with the `&` sign gives the memory address of the variable. Format specifier for addresses is `%p`

Functions

What is a function?

A function is a piece of code that is used to complete a specific task. Functions are useful because they only need to be written once and can be reused without writing the same code over and over again.

Types of functions

Pre-defined

- Built-in functions provided by C that are used by programmers without having to write any code for them. ex. printf( ), scanf( ), etc.
  User-defined
- Functions written by the programmer

Function prototypes

Normally functions have to be written above the main function like this:

```
#include  <stdio.h>
void function(int argument){
        printf("%d", argument)
}
int main(){
        int x = 5
        function(x);// Call the function with argument
        return 0;
}
/*
OUTPUT
------
```

```
5
*/
```

The problem with writing code like this is that the main function is at the bottom. This makes the code less readable.
A different approach of writing functions is through function prototypes.:

```
void function(int);
// state type of function and type of arguments only
int main(){
        int x = 5;
        function(x);
        return 0;
}
void function(int argument){
        printf("%d", argument);
}
/*
OUTPUT
------
5
*/
```

The output of both approaches are the same which shows that both methods are correct. The only advantage of using function prototypes is that the main function is near the top.

Header files

What are header files?

Header files are a collection of pre-written functions. You can also write your own header files.

Types of header files

Only the first 6 headers are important

Header File Name	Syntax	Meaning/ Description
stdio.h	`#include <stdio.h>`	Contains declarations for standard input/output functions
stdlib.h	`#include <stdlib.h>`	Contains declarations for memory allocation and management
string.h	`#include <string.h>`	Contains declarations for string manipulation functions

math.h	`#include <math.h>`	Contains declarations for mathematical functions
time.h	`#include <time.h>`	Contains declarations for functions to manipulate date/time
ctype.h	`#include <ctype.h>`	Contains declarations for functions to manipulate characters
errno.h	`#include <errno.h>`	Contains declarations for error-handling functions/ macros
limits.h	`#include <limits.h>`	Contains declarations for constants related to integer types
assert.h	`#include <assert.h>`	Contains declarations for the assert () macro
float.h	`#include <float.h>`	Contains declarations for constants related to float types

`stdio.h`

1. `printf()` : Used to print formatted output to the console or a file.
2. `scanf()` : Used to read formatted input from the console or a file.
3. `fgets()` : Used to read a line of text from a file or the console.
4. `fopen()` : Used to open a file.
5. `fclose()` : Used to close a file.
6. `fseek()` : Used to set the file position indicator for a file.
7. `fread()` : Used to read data from a file.
8. `fwrite()` : Used to write data to a file.

`stdlib.h`

1. `malloc()` : Used to dynamically allocate memory.
2. `free()` : Used to free dynamically allocated memory.
3. `atoi()` : Used to convert a string to an integer.
4. `atof()` : Used to convert a string to a floating-point number.
5. `rand()` : Used to generate a random number.
6. `qsort()` : Used to sort an array.

`string.h`

1. `strlen()` : Used to get the length of a string.
2. `strcpy()` : Used to copy one string to another.
3. `strcat()` : Used to concatenate two strings.
4. `strstr()` : Used to find a substring in a string.

5. `memset()` : Used to set the value of a block of memory to a specific value.
6. `memcpy()` : Used to copy a block of memory from one location to another.

`math.h`

1. `sin()` : Used to calculate the sine of an angle.
2. `cos()` : Used to calculate the cosine of an angle.
3. `tan()` : Used to calculate the tangent of an angle.
4. `sqrt()` : Used to calculate the square root of a number.
5. `pow()` : Used to raise a number to a power.
6. `ceil()` : Used to round a number up to the nearest integer.
7. `floor()` : Used to round a number down to the nearest integer.

`time.h`

1. `time()` : Used to get the current time in seconds since Jan 1 1970, (nothing special about this exact date).
2. `localtime()` : Used to convert a time value to a local time.
3. `gmtime()` : Used to convert a time value to a UTC time.
4. `mktime()` : Used to convert a local time to a time value.
5. `strftime()` : Used to format a time value as a string.

`ctype.h`

1. `isdigit()` : Used to check if a character is a digit.
2. `isalpha()` : Used to check if a character is an alphabetic character.
3. `islower()` : Used to check if a character is a lowercase letter.
4. `toupper()` : Used to convert a character to uppercase.
5. `tolower()` : Used to convert a character to lowercase.