C

C is a general-purpose programming language created by Dennis Ritchie at the Bell Laboratories in 1972.

Data Types

In C, there are different types of variables (defined with different keywords), for example:

- int - stores integers (whole numbers), without decimals, such as 123
- float - stores floating point numbers, with decimals, such as 19.99
- char - stores single characters, such as ’a’ or ’b’ . Char values are surrounded by single quotes

Syntax

```
#include <stdio.h>

int main() {
  printf("Hello World!");
  return 0;
}
```

Format Specifiers

Format specifiers are used together with the printf() function to tell the compiler what type of data the variable is storing. It is basically a placeholder for the variable value.

A format specifier starts with a percentage sign %, followed by a character.

For example, to output the value of an int variable, you must use the format specifier %d or %i surrounded by double quotes, inside the printf() function.

EXAMPLE 1

```
#include <stdio.h>

int main() {
  int myNum = 15;
  printf("%d", myNum);
  return 0;
}

---------------------------------------------------------------------------------

// output -> 15
```

EXAMPLE 2

```
#include <stdio.h>

int main() {
  // Create variables
```

```
  int myNum = 15;                 // Integer (whole number)
  float myFloatNum = 5.99;      // Floating point number
  char myLetter = 'D';             // Character

  // Print variables
  printf("%d\n", myNum);
  printf("%f\n", myFloatNum);
  printf("%c\n", myLetter);
  return 0;
}

--------------------------------------------------------------------------------

// Output :
// 15
// 5.99
// D
```

To combine both text and a variable, separate them with a comma inside the printf() function:

```
#include <stdio.h>

int main() {
  int myNum = 15;
  printf("My favorite number is: %d", myNum);
  return 0;
}
```

To print different types in a single printf() function, you can use the following:

```
#include <stdio.h>

int main() {
  int myNum = 15;
  char myLetter = 'D';
  printf("My number is %d and my letter is %c", myNum, myLetter);
  return 0;
}
```

Change Variable Values

Note: If you assign a new value to an existing variable, it will overwrite the previous value:

```
#include <stdio.h>

int main() {
  int myNum = 15;

  int myOtherNum = 23;

  // Assign the value of myOtherNum (23) to myNum
  myNum = myOtherNum;

  // myNum is now 23, instead of 15
  printf("%d", myNum);
```

```
  return 0;
}
```

The general rules for naming variables are:

- Names can contain letters, digits and underscores
- Names must begin with a letter or an underscore (_)
- Names are case sensitive (myVar and myvar are different variables)
- Names cannot contain whitespaces or special characters like !, #, %, etc.
- Reserved words (such as int) cannot be used as names

Set Decimal Precision

If you want to remove the extra zeros (set decimal precision), you can use a dot (.) followed by a number that specifies how many digits that should be shown after the decimal point:

```
#include <stdio.h>

int main() {
  float myFloatNum = 3.5;

  printf("%f\n", myFloatNum); // Default will show 6 digits after the decimal point
  printf("%.1f\n", myFloatNum); // Only show 1 digit
  printf("%.2f\n", myFloatNum); // Only show 2 digits
  printf("%.4f", myFloatNum);   // Only show 4 digits
  return 0;
}

--------------------------------------------------------------------------------

// Output
// 3.500000
// 3.5
// 3.50
// 3.5000
```

Constants

If you don't want others (or yourself) to change existing variable values, you can use the const keyword.

This will declare the variable as "constant", which means unchangeable and read-only:

```
const int myNum = 15;     // myNum will always be 15
myNum = 10;               // error: assignment of read-only variable 'myNum'
```

Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + operator to add together two values:

C divides the operators into the following groups:

1. Arithmetic operators

2. Assignment operators
3. Comparison operators
4. Logical operators
5. Bitwise operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the assignment operator (=) to assign the value 10 to a variable called x:

```
int x = 10;
```

A list of all assignment operators:

Operator	***Example***	***Same As***
=	X = 5	X = 5
+=	X +=3	X = x +3
-=	X -=3	X = x - 3
*=	X *= 3	X = x * 3
/=	X /= 3	X = x / 3
%=	X %= 3	X = x% 3

Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

A list of all comparison operators:

==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

You can also test for true or false values with logical operators.

Logical operators are used to determine the logic between variables or values:

If-Else Statements

Use the if statement to specify a block of code to be executed if a condition is true.

```
if (20 > 18) {
  printf("20 is greater than 18");
}
```

Use the else statement to specify a block of code to be executed if the condition is false.

```
int time = 20;
if (time < 18) {
  printf("Good day.");
} else {
  printf("Good evening.");
}
```

Use the else if statement to specify a new condition if the first condition is false.

```
int time = 22;
if (time < 10) {
  printf("Good morning.");
} else if (time < 20) {
  printf("Good day.");
} else {
  printf("Good evening.");
}
```

Short Hand If Else

There is also a short-hand if else, which is known as the ternary operator because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

```
variable = (condition) ? expressionTrue : expressionFalse;
```

```
int time = 20;
(time < 18) ? printf("Good day.") : printf("Good evening.");
```

Switch Case

Instead of writing many if..else statements, you can use the switch statement.

The switch statement selects one of many code blocks to be executed:

```
int day = 4;

switch (day) {
  case 1:
    printf("Monday");
    break;
  case 2:
    printf("Tuesday");
    break;
  case 3:
    printf("Wednesday");
```

```
  break;
 case 4:
  printf("Thursday");
  break;
 case 5:
  printf("Friday");
  break;
 case 6:
  printf("Saturday");
  break;
 case 7:
  printf("Sunday");
  break;
}
```

When C reaches a break keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

The default keyword specifies some code to run if there is no case match:

```
int day = 4;

switch (day) {
 case 6:
  printf("Today is Saturday");
  break;
 case 7:
  printf("Today is Sunday");
  break;
 default:
  printf("Looking forward to the Weekend");
}
```

Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

While Loop

The while loop loops through a block of code as long as a specified condition is true:

```
int i = 0;

while (i < 5) {
 printf("%d\n", i);
 i++;
}
```

Note Do not forget to increase the variable used in the condition (i++), otherwise the loop will never end!

Do-While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

```
int i = 0;

do {
  printf("%d\n", i);
  i++;
}
while (i < 5);
```

For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

```
int i;

for (i = 0; i < 5; i++) {
  printf("%d\n", i);
}
```

Break and Continue

The break statement can also be used to jump out of a loop.

```
int i;

for (i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  printf("%d\n", i);
}
```

Here the loop will end when i will be equal to 4.

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
int i;

for (i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  printf("%d\n", i);
}
```

Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To create an array, define the data type (like int) and specify the name of the array followed by square brackets [].

```
int myNumbers[] = {25, 50, 75, 100};
```

We have now created a variable that holds an array of four integers.

To access an array element, refer to its index number.

Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

```
int myNumbers[] = {25, 50, 75, 100};
printf("%d", myNumbers[0]);

// Outputs 25
```

To change the value of a specific element, refer to the index number:

```
myNumbers[0] = 33;

int myNumbers[] = {25, 50, 75, 100};
myNumbers[0] = 33;

printf("%d", myNumbers[0]);

// Now outputs 33 instead of 25
```

Loop through an array:

You can loop through the array elements with the for loop.

```
int myNumbers[] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
  printf("%d\n", myNumbers[i]);
}
```

Strings

Strings are used for storing text/characters.

For example, "Hello World" is a string of characters.

Unlike many other programming languages, C does not have a String type to easily create string variables. Instead, you must use the char type and create an array of characters to make a string in C:

```
char greetings[] = "Hello World!";
```

Note that you have to use double quotes (" ").

To output the string, you can use the printf() function together with the format specifier %s to tell C that we are now working with strings:

```
char greetings[] = "Hello World!";
printf("%s", greetings);
```

Since strings are actually arrays in C, you can access a string by referring to its index number inside square brackets [].

```
char greetings[] = "Hello World!";
```

```
printf("%c", greetings[0]);

// Output -> H
```

Note that we have to use the %c format specifier to print a single character.

To change the value of a specific character in a string, refer to the index number, and use single quotes:

```
char greetings[] = "Hello World!";
greetings[0] = 'J';
printf("%s", greetings);

// Outputs Jello World! instead of Hello World!
```

Special Characters

```
char txt[] = "We are the so-called "Vikings" from the north.";

// Error
```

The solution to avoid this problem, is to use the backslash escape character.

The backslash (\ ) escape character turns special characters into string characters:

```
char txt[] = "We are the so-called \"Vikings\" from the north.";

// Now this is okk

char txt[] = "It\'s alright.";

// Output -> It's alright.
```

Other popular escape characters in C are:

C also has many useful string functions, which can be used to perform certain operations on strings.

To use them, you must include the <string.h> header file in your program:

```
#include <string.h>
```

For example, to get the length of a string, you can use the strlen() function:

```
char alphabet[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
printf("%d", strlen(alphabet));

// Output -> 26
```

Concatenate Strings

To concatenate (combine) two strings, you can use the strcat() function:

```
char str1[20] = "Hello ";
char str2[] = "World!";
```

```
// Concatenate str2 to str1 (result is stored in str1)
strcat(str1, str2);

// Print str1
printf("%s", str1);
```

Note that the size of str1 should be large enough to store the result of the two strings combined (20 in this example).

To copy the value of one string to another, you can use the strcpy() function:

```
char str1[20] = "Hello World!";
char str2[20];

// Copy str1 to str2
strcpy(str2, str1);

// Print str2
printf("%s", str2);
```

Note that the size of str2 should be large enough to store the copied string (20 in this example).

User Input

To get user input, you can use the scanf() function:

```
// Create an integer variable that will store the number we get from the user
int myNum;

// Ask the user to type a number
printf("Type a number: \n");

// Get and save the number the user types
scanf("%d", &myNum);

// Output the number the user typed
printf("Your number is: %d", myNum);
```

The scanf() function takes two arguments: the format specifier of the variable (%d in the example above) and the reference operator (&myNum), which stores the memory address of the variable.

The scanf() function also allow multiple inputs (an integer and a character in the following example):

```
// Create an int and a char variable
int myNum;
char myChar;

// Ask the user to type a number AND a character
printf("Type a number AND a character and press enter: \n");

// Get and save the number AND character the user types
scanf("%d %c", &myNum, &myChar);

// Print the number
printf("Your number is: %d\n", myNum);

// Print the character
```

```
printf("Your character is: %c\n", myChar);

You can also get a string entered by the user:

// Create a string
char firstName[30];

// Ask the user to input some text
printf("Enter your first name: \n");

// Get and save the text
scanf("%s", firstName);

// Output the text
printf("Hello %s", firstName);
```

When working with strings in scanf(), you must specify the size of the string/array (we used a very high number, 30 in our example, but atleast then we are certain it will store enough characters for the first name), and you don't have to use the reference operator (&).

Multi Word Strings

To use multi word string or full name in a single string like : "Zayn Malik" we can use gets() function.

```
#include<stdio.h>

int main(){
   char str[50];

   printf("Enter the name you want to print : ");
   scanf("%s", str);
   printf("%s", str);

   return 0;
}
```

Instead you can use this

```
#include<stdio.h>

int main(){
   char str[50];

   printf("Enter the name you want to print : ");
   gets(str);
   printf("%s", str);

   return 0;
}
```

Just like we have gets() for taking multi word string input, similarly we can also use puts() to print.

```
#include<stdio.h>

int main(){
   char str[50];
```

```
   printf("Enter the name you want to print : ");
   gets(str);
   puts(str);

   return 0;
}
```

Memory Address

When a variable is created in C, a memory address is assigned to the variable.

The memory address is the location of where the variable is stored on the computer.

When we assign a value to the variable, it is stored in this memory address.

To access it, use the reference operator (&), and the result represents where the variable is stored:

```
int myAge = 43;
printf("%p", &myAge);

// Outputs 0x7ffe5367e044
```

Note : The memory address is in hexadecimal form (0x..). You will probably not get the same result in your program, as this depends on where the variable is stored on your computer.

You should also note that &myAge is often called a "pointer". A pointer basically stores the memory address of a variable as its value. To print pointer values, we use the %p format specifier.

Pointer

```
int myAge = 43; // an int variable

printf("%d", myAge);  // Outputs the value of myAge (43)
printf("%p", &myAge); // Outputs the memory address of myAge (0x7ffe5367e044)
```

- A pointer is a variable that stores the memory address of another variable as its value.
- A pointer variable points to a data type (like int) of the same type, and is created with the * operator.

The address of the variable you are working with is assigned to the pointer:

```
int myAge = 43;     // An int variable
int *ptr = &myAge;  // A pointer variable, with the name ptr, that stores the address of myAge

// Output the value of myAge (43)
printf("%d\n", myAge);

// Output the memory address of myAge (0x7ffe5367e044)
printf("%p\n", &myAge);

// Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);
```

Dereference

You can also get the value of the variable the pointer points to, by using the * operator (the dereference operator):

```
int myAge = 43;     // Variable declaration
int* ptr = &myAge;  // Pointer declaration

// Reference: Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);

// Dereference: Output the value of myAge with the pointer (43)
printf("%d\n", *ptr);
```

Pointers & Arrays

You can also use pointers to access arrays.

```
int myNumbers[4] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
  printf("%d\n", myNumbers[i]);
}
```

Instead of printing the value of each array element, let's print the memory address of each array element:

```
int myNumbers[4] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
  printf("%p\n", &myNumbers[i]);
}

// Output

/*
0x7ffe70f9d8f0
0x7ffe70f9d8f4
0x7ffe70f9d8f8
0x7ffe70f9d8fc
*/
```

To access the rest of the elements in myNumbers, you can increment the pointer/array (+1, +2, etc):

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the value of the second element in myNumbers
printf("%d\n", *(myNumbers + 1));

// Get the value of the third element in myNumbers
printf("%d", *(myNumbers + 2));

// Output

/*
50
75
```

```
*/
```

It is also possible to change the value of array elements with pointers:

```
int myNumbers[4] = {25, 50, 75, 100};

// Change the value of the first element to 13
*myNumbers = 13;

// Change the value of the second element to 17
*(myNumbers +1) = 17;

// Get the value of the first element
printf("%d\n", *myNumbers);

// Get the value of the second element
printf("%d\n", *(myNumbers + 1));

// Output

/*
13
17
*/
```

Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

```
// Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

A function can be called multiple times:

```
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction();
  myFunction();
  myFunction();
```

```
  return 0;
}

// I just got executed!
// I just got executed!
// I just got executed!
```

Parameters and Arguments

Information can be passed to functions as a parameter. Parameters act as variables inside the function. Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma:

```
void myFunction(char name[]) {
  printf("Hello %s\n", name);
}

int main() {
  myFunction("Liam");
  myFunction("Harry");
  myFunction("Zayn");
  myFunction("Louis");
  myFunction("Nial");
  return 0;
}

// Hello Liam
// Hello Harry
// Hello Zayn
// Hello Louis
// Hello Nial
```

Multiple Parameters

Inside the function, you can add as many parameters as you want:

```
void myFunction(char name[], int age) {
  printf("Hello %s. You are %d years old.\n", name, age);
}

int main() {
  myFunction("Liam", 28);
  myFunction("Harry", 27);
  myFunction("Zayn", 30);
  myFunction("Louis", 30);
  myFunction("Nial", 29);
  return 0;
}

// Hello Liam. You are 28 years old.
// Hello Harry. You are 27 years old.
// Hello Zayn. You are 30 years old.
// Hello Louis. You are 30 years old.
// Hello Nial. You are 29 years old.
```

Pass Arrays as Function Parameters

```
void myFunction(int myNumbers[5]) {
  for (int i = 0; i < 5; i++) {
    printf("%d\n", myNumbers[i]);
  }
}

int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  myFunction(myNumbers);
  return 0;
}
```

Return Values

The void keyword indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as int or float, etc.) instead of void, and use the return keyword inside the function:

```
int myFunction(int x) {
  return 5 + x;
}

int main() {
  printf("Result is: %d", myFunction(3));
  return 0;
}

// Outputs 8 (5 + 3)
```

Recursion

Recursion in C is a programming technique where a function calls itself to solve a problem. It's like solving a problem by breaking it down into smaller, similar subproblems and solving those subproblems.

Imagine you want to calculate the factorial of a number. The factorial of a number n (written as n!) is the product of all positive integers from 1 to n. For example, 5! is equal to 5 * 4 * 3 * 2 * 1, which is 120.

```
#include <stdio.h>

// Define a recursive function to calculate factorial
int factorial(int n) {
    // Base case: If n is 0 or 1, return 1
    if (n == 0 || n == 1) {
        return 1;
    }
    // Recursive case: Calculate factorial of (n-1) and multiply it by n
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int number = 5;
    int result = factorial(number);
    printf("%d! = %d\n", number, result);
    return 0;
}
```

Structure

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a member of the structure.

Unlike an array, a structure can contain many different data types (int, float, char, etc.).

```
struct MyStructure {   // Structure declaration
  int myNum;           // Member (int variable)
  char myLetter;       // Member (char variable)
}; // End the structure with a semicolon
```

Create a struct variable with the name "s1":

```
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  struct myStructure s1;
  return 0;
}
```

Access Structure Members

To access members of a structure, use the dot syntax (.):

```
// Create a structure called myStructure
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  // Create a structure variable of myStructure called s1
  struct myStructure s1;

  // Assign values to members of s1
  s1.myNum = 13;
  s1.myLetter = 'B';

  // Print values
  printf("My number: %d\n", s1.myNum);
  printf("My letter: %c\n", s1.myLetter);

  return 0;
}
```

Enumeration (enum)

In C programming, an enumeration (enum) is a user-defined data type that consists of a set of named constant values. Enums provide a way to create a collection of related named constants, which can make your code more readable and maintainable by giving meaningful names to integer values. Each constant within an enum is assigned an integer value by default, starting from 0 and incrementing by 1 for each subsequent constant.

To create an enum, use the enum keyword, followed by the name of the enum, and separate the enum items with a comma:

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
};
```

```
#include <stdio.h>

// Declare an enum named 'Day' representing days of the week
enum Day {
   Sunday,    // 0
   Monday,    // 1
   Tuesday,   // 2
   Wednesday, // 3
   Thursday,  // 4
   Friday,    // 5
   Saturday   // 6
};

int main() {
   // Declare a variable of type 'enum Day'
   enum Day today;

   // Assign a value from the 'Day' enum to the variable
   today = Wednesday;

   // Output the value of 'today'
   printf("Today is day number %d\n", today);

   return 0;
}
```

Note that if you assign a value to one specific item, the next items will update their numbers accordingly:

```
enum Level {
  LOW = 5,
  MEDIUM,    // Now 6
  HIGH       // Now 7
};
```

Enums using Switch Case

```
#include <stdio.h>

// Declare an enum named 'Month' representing months of the year
enum Month {
   January,   // 0
   February,  // 1
   March,     // 2
   April,     // 3
   May,       // 4
   June,      // 5
   July,      // 6
   August,    // 7
```

```
  September, // 8
  October,   // 9
  November,  // 10
  December   // 11
};

int main() {
  // Declare a variable of type 'enum Month'
  enum Month currentMonth;

  // Assign a value from the 'Month' enum to the variable
  currentMonth = August;

  // Use a switch statement to determine the number of days in the current month
  switch (currentMonth) {
    case January:
    case March:
    case May:
    case July:
    case August:
    case October:
    case December:
      printf("This month has 31 days.\n");
      break;

    case April:
    case June:
    case September:
    case November:
      printf("This month has 30 days.\n");
      break;

    case February:
      printf("This month has either 28 or 29 days.\n");
      break;

    default:
      printf("Invalid month.\n");
  }

  return 0;
}
```

File I/O

In C, you can create, open, read, and write to files by declaring a pointer of type FILE, and use the fopen() function:

```
FILE *fptr
fptr = fopen(filename, mode);
```

FILE is basically a data type, and we need to create a pointer variable to work with it (fptr).
To actually open a file, use the fopen() function, which takes two parameters:

Filename - The name of the actual file you want to open (or create), like filename.txt
Mode - A single character, which represents what you want to do with the file (read, write or append):

w -> Writes to a file
a -> Appends new data to a file
r -> Reads from a file

Create a File
To create a file, you can use the w mode inside the fopen() function.
The w mode is used to write to a file. However, if the file does not exist, it will create one for you:

```
FILE *fptr;

// Create a file
fptr = fopen("filename.txt", "w");

// Close the file
fclose(fptr);
```

***NOTE* -** If you want to create the file in a specific folder, just provide an absolute path:

```
fptr = fopen("C:\directoryname\filename.txt", "w");
```

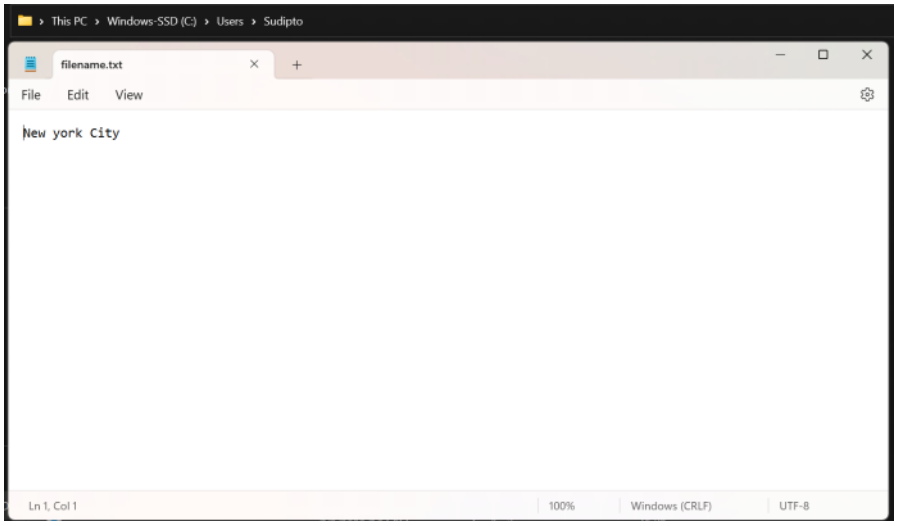
For closing the file use use the fclose() function.
It is considered as good practice, because it makes sure that:

- Changes are saved properly
- Other programs can use the file (if you want)
- Clean up unnecessary memory space

Write To a File

The w mode means that the file is opened for writing. To insert content to it, you can use the fprint() function and add the pointer variable (fptr in our example) and some text:

```
#include <stdio.h>

int main() {
  FILE *fptr;

  // Open a file in writing mode
  fptr = fopen("filename.txt", "w");

  // Write some text to the file
  fprintf(fptr, "New York City");

  // Close the file
  fclose(fptr);

  return 0;
}
```



The program has created a filename.txt file in the location which was the default location for all my C files.

***Note* -** If you write to a file that already exists, the old content is deleted, and the new content is inserted. This is important to know, as you might accidentally erase existing content.

Append Content To a File

If you want to add content to a file without deleting the old content, you can use the a mode.
The a mode appends content at the end of the file:

```
#include <stdio.h>

int main() {
  FILE *fptr;

  // Open a file in append mode
  fptr = fopen("filename.txt", "a");

  // Append some text to the file
  fprintf(fptr, "\nUSA ");

  // Close the file
  fclose(fptr);

  return 0;
}
```

Note - Just like with the w mode; if the file does not exist, the a mode will create a new file with the "appended" content.

Read a File

To read from a file, you can use the r mode:

```
FILE *fptr;

// Open a file in read mode
fptr = fopen("filename.txt", "r");
```

This will make the **filename.txt** opened for reading.

Next, we need to create a string that should be big enough to store the content of the file.
For example, let's create a string that can store up to 100 characters:

```
FILE *fptr;

// Open a file in read mode
fptr = fopen("filename.txt", "r");

// Store the content of the file
char myString[100];
```

In order to read the content of **filename.txt**, we can use the **fgets()** function.
The **fgets()** function takes three parameters:

```
fgets(myString, 100, fptr);
```

1. The first parameter specifies where to store the file content, which will be in the myString array we just created.
2. The second parameter specifies the maximum size of data to read, which should match the size of myString (100).
3. The third parameter requires a file pointer that is used to read the file (fptr in our example).

```
#include <stdio.h>

int main() {
  FILE *fptr;

  // Open a file in read mode
  fptr = fopen("filename.txt", "r");

  // Store the content of the file
  char myString[100];

  // Read the content and store it inside myString
  fgets(myString, 100, fptr);

  // Print file content
  printf("%s", myString);

  // Close the file
  fclose(fptr);

  return 0;
}
```

Note - The fgets function only reads the first line of the file. If you remember, there were two lines of text in filename.txt.
To read every line of the file, you can use a while loop:

```
#include <stdio.h>

int main() {
  FILE *fptr;

  // Open a file in read mode
  fptr = fopen("filename.txt", "r");

  // Store the content of the file
  char myString[100];

  // Read the content and print it
  while(fgets(myString, 100, fptr)) {
```

```
    printf("%s", myString);
  }

  // Close the file
  fclose(fptr);

  return 0;
}
```

Good Practice

If you try to open a file for reading that does not exist, the fopen() function will return NULL.

NOTE - As a good practice, we can use an if statement to test for NULL, and print some text instead (when the file does not exist):

```
#include <stdio.h>

int main() {
  FILE *fptr;

  // Open a file in read mode
  fptr = fopen("loremipsum.txt", "r");

  // Print some text if the file does not exist
  if(fptr == NULL) {
    printf("Not able to open the file.");
  }

  // Close the file
  fclose(fptr);

  return 0;
}
```