Sub Topic(s): C Intro, C Syntax, C Output, C Comments, C Variables, C Data Types, C Constants, C Operators, C Booleans, C If...Else, C Switch, C While Loop, C For Loop, C Break/Continue, C Arrays, C Strings, C User Input, C Memory Address, C Pointers, C Functions, C Function parameters, C Function Declaration, C Recursion, C Math Functions, C Create Files, C Write To Files, C Read Files, C Structures, C Enums

What is C?

C is a general-purpose programming language created by Dennis Ritchie at the Bell laboratories in 1972.
It is a very popular language, despite being old.

C is strongly associated with UNIX, as it was developed to write the UNIX operating system.

Why learn C?

- It is one of the most popular programming languages in the world.
- If we know C, we will have no problem learning other programming languages such as Java, Python, C++, C# etc as the syntax is similar.
- C is very fast, compared to other programming languages like Java & Python.
- C is very versatile; it can be used in both technologies & applications.

Difference between c & C++

- C++ was developed as an extension of C, & both languages have almost same syntax.
- The main difference between C & C++ is that C++ support classes & objects, while C does not.

Get Started With C

To start using C, we will need two things:

- A text editor, like Notepad, to write C code.
- A compiler, like gcc, to translate the C code into a language that the compiler will understand.

There are many text editors and compilers to choose from. In this tutorial, we will use an IDE (see below).

C install IDE

An IDE (Integrated Development Environment) is used to edit AND compile the code.
Popular IDE's include Code::Blocks, Eclipse, and Visual Studio. These are all free, and they can be used to both edit and debug C code.
Note: Web-based IDE's can work as well, but functionality is limited.
We will use Code::Blocks in our tutorial, which we believe is a good place to start.
We can find the latest version of Codeblocks at http://www.codeblocks.org/. Download the mingw-setup.exe file, which will install the text editor with a compiler

C QuickStart

Let's create our first C file
Open Codeblocks and go to File > New > Empty File.
Write the following C code and save the file as myfirstprogram.c (File > Save File as):

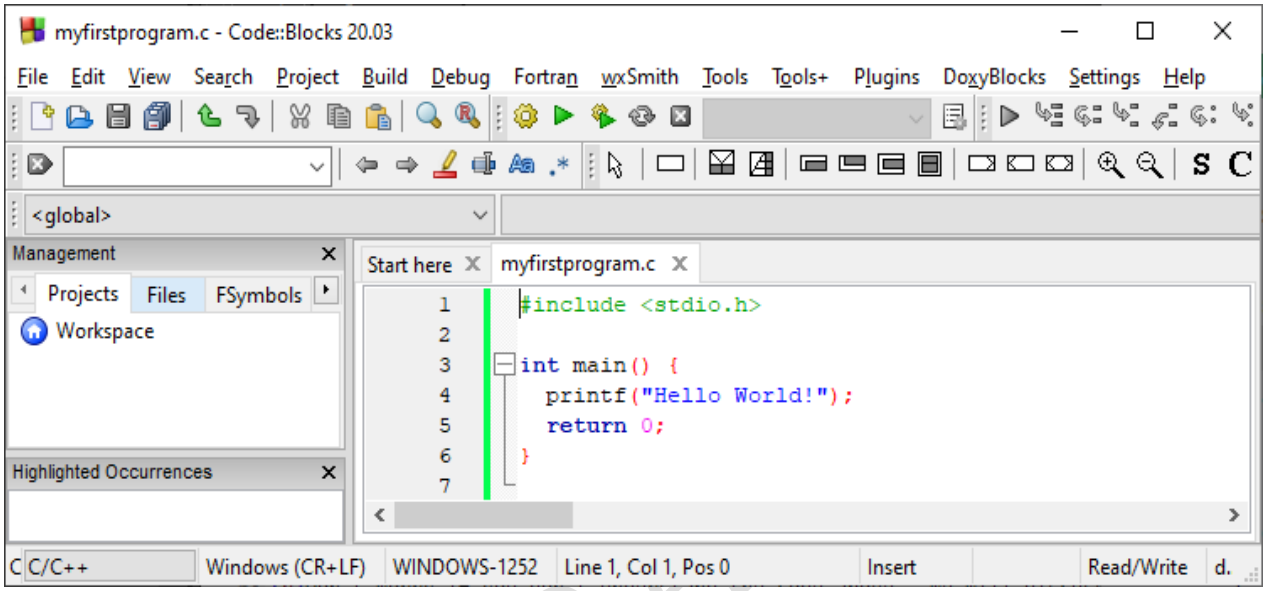
Myfirstprogram.c

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

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

Don't worry if you don't understand the code above - we will discuss it in detail in later chapters. For now, focus on how to run the code.

In Codeblocks, it should look like this:



Then, go to **Build > Build and Run** to run (execute) the program. The result will look something to this:

```
Hello World!
Process returned 0 (0x0) execution time: 0.011 s
Press any key to continue.
```

C Syntax

We have already seen the following code a couple of times in the first chapters. Let's break it down to understand it better:

Example

```
#include<stdio.h>
Int main() {
printf("Hello World!");
return 0;
 }
```

Example explained

Line 1: #include <stdio.h> is a **header file library** that lets us work with input and output functions, such as **printf()** (used in line 4). Header files add functionality to C programs.

> **Note:** Don't worry if you don't understand how #include <stdio.h> works. Just think of it as something that (almost) always appears in your program.

Line 2: A blank line. C ignores white space. But we use it to make the code more readable.

Line 3: 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.

Line 4: printf() is a f**unction** used to output/print text to the screen. In our example it will output "Hello World!".

> **Note that:** Every C statement ends with a semicolon ;
>
> **Note:** The body of int main() could also been written as:
> int main(){printf("Hello World!");return 0;}
>
> **Remember:** The compiler ignores white spaces. However, multiple lines makes the code more readable.

L

Line 6: Do not forget to add the closing curly bracket} to actually end the main function.

Output (Print Text)

To output values or print text in C, we can use the printf() function

Example

```
 #include<stdio.h>
Int main()  {
printf("Hello World!!");
return 0;
 }
```

We can use as many printf() functions as you want. However, note that it does not insert a new line at the end of the output:

Example

```
#include<stdio.h>
Int main()  {
printf("Hello World!");
printf("I am learning C");
return 0;
 }
```

New Lines

To insert a new line, we can use the \n character:

Example

```
#include<stdio.h>
Int main()  {
printf("Hello World!\n");
printf("I am learning C");
return 0;
```

}

We can also output multiple lines with a single printf() function. However, this could make the code harder to read:

Example

```
#include<stdio.h>
Int main()  {
printf("Hello World!\n I am fine\n and very well");
return 0;
```

Tip: Two \n characters after each other will create a blank line

Example

```
#include<stdio.h>
int main()  {
printf("Hello World!\n\n");
printf("I am learning C");
return 0;
 }
```

What is \n exactly?

The newline character (\n) is called an **escape sequence**, and it forces the cursor to change its position to the beginning of the next line on the screen. This results in a new line.

Examples of other valid escape sequences are:

Escape Sequence	Description
\t	Creates a horizontal tab
\\	Inserts a backslash character (\)
\"	Inserts a double quote character

Comments in C

Comments can be used to explain code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Comments can be **singled-lined** or **multi-lined**.

Single-line Comments

Single-line comments start with two forward slashes (//).

Any text between // and the end of the line is ignored by the compiler (will not be executed).

This example uses a single-line comment before a line of code:

Example

```
printf("Hello World!");    //This is a comment
```

C Multi-Line Comments

Multi-line comments start with /* and ends with */.

Any text between /* and */ will be ignored by the compiler:

Example

```
/* The code below will print
The words Hello World! On the screen*/
printf("Hello World");
```

Single or multi-line comments?

It is up to us which we want to use. Normally, we use // for short comments, and /* */ for longer.

Good to know: Before version **C99** (released in 1999), we could only use multi-line comments in C.

C Variables

Variables are containers for storing data values, like numbers and characters.
In C, there are different types of variables (defined with different keywords), for example:

- Int - stores integers (whole numbers), without decimals, such as 123 or -123.
- Float - stores floating point numbers, with decimals, such as 19.99 or -19.99
- Char - stores single characters, such as 'a' or 'B'. Char values are surrounded by **single quotes**

Declaring (Creating) Variables

To create a variable, specify the type and assign it a value:

Syntax

```
type variableName = value;
```

Where type is one of C types (such as int), and variableName is the name of the variable (such as **x** or **myName**). The **equal sign** is used to assign a value to the variable.
So, to create a variable that should **store a number**, look at the following example:

Example

Create a variable called myNum of type int and assign the value 15 to it:

```
int myNum = 15:
```

We can also declare a variable without assigning the value, and assign the value later:

Example

```
// Declare a variable
int myNum;
// Assign a value to the variable
myNum = 15;
```

Output Variables

We learned from the output chapter that you can output values/print text with the printf() function:

Example

```
printf("Hello World!");
```

In many other programming languages (like Python, Java, and C++), we would normally use a print function to display the value of a variable. However, this is not possible in C:

Example

```
int myNum = 15;
printf(myNum);  // Nothing happens
```

To output variables in C, we must get familiar with something called "format specifiers".

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, we must use the format specifier %d or %i surrounded by double quotes, inside the printf() function:

Example

```
int myNum = 15;
printf("%d", myNum);  // Outputs 15
```

To print other types, use %c for char and %f for float:

Example

```
// 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);
```

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

Example

```
int myNum = 15;
printf("My favorite number is: %d", myNum);
```

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

Example

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

Change Variable Values

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

Example

```
int myNum = 15;  // myNum is 15
myNum = 10;  // Now myNum is 10
```

We can also assign the value of one variable to another:

Example

```
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);
```

Or copy values to empty variables:

Example

```
// Create a variable and assign the value 15 to it
int myNum = 15;
// Declare a variable without assigning it a value
int myOtherNum;
// Assign the value of myNum to myOtherNum
myOtherNum = myNum;
// myOtherNum now has 15 as a value
printf("%d", myOtherNum);
```

Add values together

To add a variable to another variable, you can use the + operator:

Example

```
int x = 5;
int y = 6;
int sum = x + y;
printf("%d", sum);
```

Declare Multiple Variables

To declare more than one variable of the same type, use a comma-separated list:

Example

```
int x = 5, y = 6, z = 50;
printf("%d", x + y + z);
```

We can also assign the same value to multiple variables of the same type:

Example

```
int x, y, z;
x = y = z = 50;
printf("%d", x + y + z);
```

C Variable Names

All C variables must be identified with unique names.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code:

Example

```
// Good
int minutesPerHour = 60;
// OK, but not so easy to understand what m actually is
int m = 60;
```

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

Real-Life Example

Often in our examples, we simplify variable names to match their data type (myInt or myNum for int types, myChar for char types etc). This is done to avoid confusion.

However, if we want a real-life example on how variables can be used, take a look at the following, where we have made a program that stores different data of a college student:

Example

```
// Student data
int studentID = 15;
int studentAge = 23;
float studentFee = 75.25;
char studentGrade = 'B';
// Print variables
printf("Student id: %d\n", studentID);
printf("Student age: %d\n", studentAge);
printf("Student fee: %f\n", studentFee);
printf("Student grade: %c", studentGrade);
```

Data Types

As explained in the Variables, a variable in C must be a specified **data type**, and we must use a **format specifier** inside the printf() function to display it:

Example

```
// Create variables
 int myNum = 5;           // 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);
```

Basic Data Types

The data type specifies the size and type of information the variable will store.

In this tutorial, we will focus on the most basic ones:

Data Type	Size	Description
int	2 or 4 bytes	Stores whole numbers, without decimals
float	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits
Double	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits
char	1 byte	Stores a single character/letter/number, or ASCII values

Basic Format Specifiers

There are different format specifiers for each data type. Here are some of them

Format Specifier	Data Type
%d or %i	int
%f	float
%lf	double
%c	char
%s	Used for strings (text), which we will learn more about in a later chapter

Set Decimal Precision

We have probably already noticed that if you print a floating point number, the output will show many digits after the decimal point:

Example

```
Float myFloatNum = 3.5;
double myDoubleNum = 19.99;
printf("%f\n", myFloatNum); // Outputs 3.500000
printf("%lf", myDoubleNum); // Outputs 19.990000
```

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

Example

```
float myFloatNum = 3.5;
printf("%f\n", myFloatNum); // Default will show 6 digits after decimal
printf("%.1f\n", myFloatNum); // Only show 1 digit
printf("%.2f\n", myFloatNum); // Only show 2 digits
printf("%.4f", myFloatNum);   // Only show 4 digits
```

Type Conversion

Sometimes, we have to convert the value of one data type to another type. This is known as **type conversion**.

For example, if we try to divide two integers, 5 by 2, we would expect the result to be 2.5. But since we are working with integers (and not floating-point values), the following example will just output 2:

Example

```
Int x = 5;
int y = 2;
int sum = 5 / 2;
printf("%d", sum); // Outputs 2
```

To get the right result, we need to know how type conversion works.

There are two types of conversion in C:

- **Implicit Conversion (automatically)**
- **Explicit Conversion (manually)**

Implicit Conversion

Implicit conversion is done automatically by the compiler when we assign a value of one type to another.

For example, if we assign an int value to a float type:

Example

```
// Automatic conversion: int to float
float myFloat = 9;
printf("%f", myFloat); // 9.000000
```

As we can see, the compiler automatically converts the int value 9 to a float value of 9.000000.

This can be risky, as we might lose control over specific values in certain situations. Especially if it was the other way around - the following example automatically converts the float value 9.99 to an int value of 9:

Example

```
// Automatic conversion: float to int
int myInt = 9.99;
printf("%d", myInt); // 9
```

What happened to .99? We might want that data in our program! So be careful. It is important that we know how the compiler work in these situations, to avoid unexpected results.

As another example, if we divide two integers: 5 by 2, we know that the sum is 2.5. And as we know from the beginning of this page, if we store the sum as an integer, the result will only display the number 2. Therefore, it would be better to store the sum as a float or a double, right?

Example

```
float sum = 5 / 2;
printf("%f", sum); // 2.000000
```

Why is the result 2.00000 and not 2.5? Well, it is because 5 and 2 are still integers in the division. In this case, we need to manually convert the integer values to floating-point values. (see below).

Explicit Conversion

Explicit conversion is done manually by placing the type in parentheses () in front of the value.

Considering our problem from the example above, we can now get the right result:

Example

```
// Manual conversion: int to float
Float sum = (float) 5 / 2;
printf("%f", sum); // 2.500000
```

We can also place the type in front of a variable:

Example

```
int num1 = 5;
int num2 = 2;
float sum = (float) num1 / num2;
printf("%f", sum); // 2.500000
```

And since we learned about "decimal precision" , we could make the output even cleaner by removing the extra zeros:

Example

```
int num1 = 5;
int num2 = 2;
float sum = (float) num1 / num2;
printf("%.1f", sum); // 2.5
```

C Constants

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

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

Example

```
const int myNum = 15; // myNum will always be 15
```

```
myNum = 10; // error: assignment of read-only variable 'myNum'
```

We should always declare the variable as constant when we have values that are unlikely to change:

Example

```
const int minutesPerHour = 60;
const float PI = 3.14;
```

Notes On Constants

When we declare a constant variable, it must be assigned with a value:

Example

Like this:

```
const int minutesPerHour = 60;
```

This however, **will not work**:

```
const int minutesPerHour;
minutesPerHour = 60; // error
```

Good Practice

Another thing about constant variables, is that it is considered good practice to declare them with uppercase. It is not required, but useful for code readability and common for C programmers:

Example

```
const int BIRTHYEAR = 1910;
```

C Operators

Operators are used to perform operations on variables and values.

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

Example

```
int myNum = 100 + 50;
```

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Example

```
int sum1 = 100 + 50; // 150 (100 + 50)
int sum2 = sum1 + 250;// 400 (150 + 250)
int sum3 = sum2 + sum2; // 800 (400 + 400)
```

C divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	X + Y
-	Subtraction	Subtracts one value from another	X – Y
*	Multiplication	Multiplies two values	X * Y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x

--	Decrement	Decreases the value of a variable by 1	--x

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**:

Example

```
int x = 10;
```

The **addition assignment** operator (+=) adds a value to a variable:

Example

```
int x = 10;
x += 5;
```

A list of all assignment operators:

Operator	Example	Same As
=	X = 5	X = 5
+=	X += 5	X = x + 3
-=	X -= 5	X = x - 3
*=	X *= 5	X = x * 3
/=	X /= 5	X = x / 3
%=	X %= 5	X = x % 3
&=	X &= 5	X = x & 3
|=	X |= 5	X = x | 3
^=	X ^= 5	X = x ^ 3
>>=	X >>= 5	X = x >> 3
<<=	X <<= 5	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.

The return value of a comparison is either 1 or 0, which means **true** (1) or **false** (0). These values are known as **Boolean values**, and we will learn more about them in the Booleans and If..Else chapter.

In the following example, we use the **greater than** operator (>) to find out if 5 is greater than 3:

Example

```
int x = 5;
int y = 3;
printf("%d", x > y); // returns 1 (true) because 5 is greater than 3
```

A list of all comparison operators:

Operator	Name	Example
==	Equal to	x==y
!=	Not equal to	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

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

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

Operator	Name	Description	Example
&&	Logical AND	Returns true if both statements	x < 5 && x < 10

		are true	
||	Logical OR	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical NOT	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Sizeof Operator

The memory size (in bytes) of a data type or a variable can be found with the sizeof operator:

Example

```
int myInt;
float myFloat;
double myDouble;
 char myChar;
 printf("%lu\n", sizeof(myInt));
 printf("%lu\n", sizeof(myFloat));
 printf("%lu\n", sizeof(myDouble));
 printf("%lu\n", sizeof(myChar));
```

Note that we use the %lu format specifer to print the result, instead of %d. It is because the compiler expects the sizeof operator to return a long unsigned int (%lu), instead of int (%d). On some computers it might work with %d, but it is safer to use %lu.

Booleans

Very often, in programming, we will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, C has a bool data type, which is known as **booleans**.

Booleans represent values that are either true or false.

Boolean Variables

In C, the bool type is not a built-in data type, like int or char.

It was introduced in C99, and you must **import** the following header file to use it:

```
#include <stdbool.h>
```

A boolean variable is declared with the bool keyword and can only take the values true or false:

```
bool isProgrammingFun = true;
bool isFishTasty = false;
```

Before trying to print the boolean variables, we should know that boolean values are returned as integers:

- 1 (or any other number that is not 0) represents true
- 0 represents false

Therefore, we must use the %d format specifier to print a boolean value:

Example

```
// Create boolean variables
bool isProgrammingFun = true;
bool isFishTasty = false;
// Return boolean values
printf("%d", isProgrammingFun);   // Returns 1 (true)
 printf("%d", isFishTasty);       // Returns 0 (false)
```

However, it is more common to return a boolean value by **comparing** values and variables.

Comparing Values and Variables

Comparing values are useful in programming, because it helps us to find answers and make decisions.

For example, we can use a comparison operator, such as the **greater than** (>) operator, to compare two values:

Example

```
printf("%d", 10 > 9);  // Returns 1 (true) because 10 is greater than 9
```

From the example above, we can see that return value is a boolean value (1).
We can also compare two variables:

Example

```
int x = 10;
int y = 9;
printf("%d", x > y);
```

In the example below, we use the equal to (==) operator to compare different values:

Example

```
printf("%d", 10 == 10); // Returns 1 (true), because 10 is equal to 10
printf("%d", 10 == 15); // Returns 0 (false), because 10 is not equal to 15
printf("%d", 5 == 55);  // Returns 0 (false) because 5 is not equal to 55
```

We are not limited to only compare numbers. You can also compare boolean variables, or even special structures, like arrays (which you will learn more about in a later chapter):

Example

```
bool isHamburgerTasty = true;
bool isPizzaTasty = true;
// Find out if both hamburger and pizza is tasty
printf("%d", isHamburgerTasty == isPizzaTasty);
```

Remember to include the <stdbool.h> header file when working with bool variables.

Real Life Example

Let's think of a "real life example" where we need to find out if a person is old enough to vote.
In the example below, we use the >= comparison operator to find out if the age (25) is **greater than** OR **equal to** the voting age limit, which is set to 18:

Example

```
int myAge = 25;
int votingAge = 18;
printf("%d", myAge >= votingAge); // Returns 1 (true), meaning 25 year olds are allowed to vote!
```

Right? An even better approach (since we are on a roll now), would be to wrap the code above in an if...else statement, so we can perform different actions depending on the result:

Example

Output "Old enough to vote!" if myAge is **greater than or equal** to 18. Otherwise output "Not old enough to vote.":

```
int myAge = 25;
int votingAge = 18;
if (myAge >= votingAge) {
printf("Old enough to vote!");
} else {
printf("Not old enough to vote.");
 }
```

Booleans are the basis for all comparisons and conditions.

Conditions and If Statements

We have already learned that C supports the usual logical conditions from mathematics:

- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b
- Equal to a == b
- Not Equal to: a != b

We can use these conditions to perform different actions for different decisions.
C has the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false
- Use switch to specify many alternative blocks of code to be executed

The if Statement

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

Syntax

```
if (condition) {
// block of code to be executed if the condition is true
}
```

Note that if is in lowercase letters. Uppercase letters (If or IF) will generate an error.
In the example below, we test two values to find out if 20 is greater than 18. If the condition is true, print some text:

Example

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

We can also test variables:

Example

```
int x = 20;
int y = 18;
If (x > y) {
printf("x is greater than y");
 }
```

Example explained

In the example above we use two variables, **x** and **y**, to test whether x is greater than y (using the > operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

The else Statement

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

Syntax

```
if (condition) {
// block of code to be executed if the condition is true
 } else {
 // block of code to be executed if the condition is false
  }
```

Example

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

```
} else {
printf("Good evening.");
}
// Outputs "Good evening."
```

Example explained

In the example above, time (20) is greater than 18, so the condition is false. Because of this, we move on to the else condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

The else if Statement

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

Syntax

```
if (condition1) {
// block of code to be executed if condition1 is true
} else if (condition2) {   // block of code to be executed if the condition1 is false and condition2 is true
} else {
// block of code to be executed if the condition1 is false and condition2 is false
}
```

Example

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

Example explained

In the example above, time (22) is greater than 10, so the **first condition** is false. The next condition, in the else if statement, is also false, so we move on to the else condition since **condition1** and **condition2** is both false - and print to the screen "Good evening". However, if the time was 14, our program would print "Good day."

Another Example

This example shows how we can use if..else to find out if a number is positive or negative:

Example

```
int myNum = 10; // Is this a positive or negative number?
if (myNum > 0) {
printf("The value is a positive number.");
} else if (myNum < 0) {
printf("The value is a negative number.");
} else {
printf("The value is 0.");
}
```

Short Hand If...Else (Ternary Operator)

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:

Syntax

variable = (condition) ? expressionTrue : expressionFalse;

Instead of writing:

Example

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

We can simply write:

Example

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

It is completely up to us if we want to use the traditional if...else statement or the ternary operator.

Switch Statement

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

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

Syntax

```
switch(expression) {
case x:
// code block
break;
 case y:
// code block
 break;
 default:
// code block
 }
```

This is how it works:

- The switch expression is evaluated once
- The value of the expression is compared with the values of each case
- If there is a match, the associated block of code is executed
- The break statement breaks out of the switch block and stops the execution
- The default statement is optional, and specifies some code to run if there is no case match

The example below uses the weekday number to calculate the weekday name:

Example

```
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;
}
// Outputs "Thursday" (day 4)
```

The break Keyword

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.
A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

The default Keyword

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

Example

```
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");
}
// Outputs "Looking forward to the Weekend"
```

Note: The default keyword must be used as the last statement in the switch, and it does not need a break.

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:

Syntax

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

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
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!

The 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.

Syntax

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

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

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

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

For Loop

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

Syntax

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

Statement 1 is executed (one time) before the execution of the code block.
Statement 2 defines the condition for executing the code block.
Statement 3 is executed (every time) after the code block has been executed.
The example below will print the numbers 0 to 4:

Example

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

Example explained

Statement 1 sets a variable before the loop starts (int i = 0).
Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.
Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Another Example

This example will only print even values between 0 and 10:

Example

```
for (i = 0; i <= 10; i = i + 2) {
printf("%d\n", i);
}
```

Nested Loops

It is also possible to place a loop inside another loop. This is called a **nested loop**.
The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

```
int i, j;
// Outer loop
for (i = 1; i <= 2; ++i) {
printf("Outer: %d\n", i);  // Executes 2 times
// Inner loop
for (j = 1; j <= 3; ++j) {
printf(" Inner: %d\n", j);  // Executes 6 times (2 * 3)
}
}
```

Break

We have already seen the break statement used in an earlier of this tutorial. It was used to "jump out" of a switch statement.
The break statement can also be used to jump out of a loop.

This example jumps out of the for loop when i is equal to 4:

Example

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

Continue

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.
This example skips the value of 4:

Example

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

Break and Continue in While Loop

We can also use break and continue in while loops:

Break Example

```
int i = 0;
while (i < 10) {
if (i == 4) {
break;
```

```
}
printf("%d\n", i);
i++;
}
```

Continue Example

```
int i = 0;
while (i < 10) {
if (i == 4) {
 i++;
continue;
}
printf("%d\n", i);
  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 []**.

To insert values to it, use a comma-separated list, inside curly braces:

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

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

Access the Elements of an Array

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.

This statement accesses the value of the **first element [0]** in myNumbers:

Example

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

Change an Array Element

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

Example

```
myNumbers[0] = 33;
```

Example

```
int myNumbers[] = {25, 50, 75, 100};
myNumbers[0] = 33;
printf("%d", myNumbers[0]);
// Now outputs 33 instead of 25
```

Loop Through an Array

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

The following example outputs all elements in the myNumbers array:

Example

```
int myNumbers[] = {25, 50, 75, 100};
int i;
for (i = 0; i < 4; i++) {
printf("%d\n", myNumbers[i]);
 }
```

Set Array Size

Another common way to create arrays, is to specify the size of the array, and add elements later:

Example

```
// Declare an array of four integers:
int myNumbers[4];
// Add elements
myNumbers[0] = 25;
myNumbers[1] = 50;
myNumbers[2] = 75;
myNumbers[3] = 100;
```

Using this method, **we would know the size of the array**, in order for the program to store enough memory.

We are not able to change the size of the array after creation.

Multidimensional Arrays

In the previous chapter, we learned about arrays, which is also known as **single dimension arrays**. These are great, and something we will use a lot while programming in C. However, if we want to store data as a tabular form, like a table with rows and columns, we need to get familiar with **multidimensional arrays**.

A multidimensional array is basically an array of arrays.

Arrays can have any number of dimensions. In this tutorial, we will introduce the most common; two-dimensional arrays (2D).

Two-Dimensional Arrays

A 2D array is also known as a matrix (a table of rows and columns).

To create a 2D array of integers, take a look at the following example:

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
```

The first dimension represents the number of rows **[2],** while the second dimension represents the number of columns **[3].** The values are placed in row-order, and can be visualized like this:

	COLUMN 0	COLUMN 1	COLUMN 2
ROW 0	1	4	2
ROW 1	3	6	8

Access the Elements of a 2D Array

To access an element of a two-dimensional array, we must specify the index number of both the row and column.

This statement accesses the value of the element in the first row (0) and third column (2) of the matrix array.

Example

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
 printf("%d", matrix[0][2]);  // Outputs 2
```

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

Change Elements in a 2D Array

To change the value of an element, refer to the index number of the element in each of the dimensions:

The following example will change the value of the element in the **first row (0)** and **first column (0):**

Example

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
matrix[0][0] = 9;
```

```
printf("%d", matrix[0][0]);  // Now outputs 9 instead of 1
```

Loop Through a 2D Array

To loop through a multi-dimensional array, we need one loop for each of the array's dimensions.

The following example outputs all elements in the **matrix** array:

```
Example
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
int i, j;
for (i = 0; i < 2; i++) {
for (j = 0; j < 3; j++) {
printf("%d\n", matrix[i][j]);
 }
}
```

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, we must use the char type and create an array of characters to make a string in C:

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

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

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

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

Access Strings

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

This example prints the **first character (0)** in **greetings**:

```
Example
char greetings[] = "Hello World!";
printf("%c", greetings[0]);
```

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

Modify Strings

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

```
Example
char greetings[] = "Hello World!";
greetings[0] = 'J';
printf("%s", greetings);
// Outputs Jello World! instead of Hello World!
```

Loop Through a String

We can also loop through the characters of a string, using a for loop:

```
Example
char carName[] = "Volvo";
int i;
for (i = 0; i < 5; ++i) {
printf("%c\n", carName[i]);
}
```

Another Way of Creating Strings

In the examples above, we used a "string literal" to create a string variable. This is the easiest way to create a string in C.

We should also note that we can create a string with a set of characters. This example will produce the same result as the example in the beginning of this tutorial:

Example

```
char greetings[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!', '\0'};
printf("%s", greetings);
```

Why do we include the \0 character at the end? This is known as the "null terminating character", and must be included when creating strings using this method. It tells C that this is the end of the string.

Differences

The difference between the two ways of creating strings, is that the first method is easier to write, and we do not have to include the \0 character, as C will do it for us

You should note that the size of both arrays is the same: They both have **13 characters** (space also counts as a character by the way), including the \0 character:

Example

```
char greetings[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!', '\0'};
char greetings2[] = "Hello World!";
printf("%lu\n", sizeof(greetings));   // Outputs 13
printf("%lu\n", sizeof(greetings2));  // Outputs 13
```

Strings - Special Characters

Because strings must be written within quotes, C will misunderstand this string, and generate an error:

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

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

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

Escape Character	Result	Description
\’	‘	Single Quote
\”	“	Double Quote
\\	\	Backslash

The sequence \" inserts a double quote in a string:

Example

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

The sequence \' inserts a single quote in a string:

Example

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

The sequence \\ nserts a single backslash in a string:

Example

```
char txt[] = "The character \\ is called backslash.";
```

Other popular escape characters in C are:

Escape Character	Result
\n	New Line
\t	Tab
\0	Null

String Functions

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

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

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

String Length

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

Example

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

In the Strings tutorial, we used sizeof to get the size of a string/array. Note that sizeof and strlen behaves differently, as sizeof also includes the \0 character when counting:

Example

```
char alphabet[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
printf("%d", strlen(alphabet));   // 26
printf("%d", sizeof(alphabet));   // 27
```

It is also important that we know that sizeof will always return the memory size (in bytes), and not the actual string length:

Example

```
char alphabet[50] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
printf("%d", strlen(alphabet));   // 26
printf("%d", sizeof(alphabet));   // 50
```

Concatenate Strings

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

Example

```
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 our example).

Copy Strings

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

Example

```
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 our example).

Compare Strings

To compare two strings, we can use the strcmp() function.

It returns 0 if the two strings are equal, otherwise a value that is not 0:

Example

```
char str1[] = "Hello";
char str2[] = "Hello";
char str3[] = "Hi";
// Compare str1 and str2, and print the result
printf("%d\n", strcmp(str1, str2));  // Returns 0 (the strings are equal)
// Compare str1 and str3, and print the result
```

```
printf("%d\n", strcmp(str1, str3));  // Returns -4 (the strings are not equal)
```

User Input

We have already learned that printf() is used to **output values** in C.
To get **user input**, you can use the scanf() function:

Example

```
Output a number entered by the user:
// Create an int variable that will store  number we get from 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.
Tip: We will learn more about memory addresses and functions in the next tutorial.

Multiple Inputs

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

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);
```

Take String Input

We can also get a string entered by the user:

Example

```
Output the name of a 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);
```

Note: When working with strings in scanf(), we 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 we don't have to use the reference operator (&).

However, the scanf() function has some limitations: it considers space (whitespace, tabs, etc) as a terminating character, which means that it can only display a single word (even if we type many words). For example:

Example

```
char fullName[30];
printf("Type your full name: \n");
scanf("%s", &fullName);
printf("Hello %s", fullName);
// Type your full name: John Doe
// Hello John
```

From the example above, we would expect the program to print "John Doe", but it only prints "John".

That's why, when working with strings, we often use the fgets() function to **read a line of text**. Note that we must include the following arguments: the name of the string variable, sizeof(*string_name*), and stdin:

Example

```
char fullName[30];
printf("Type your full name: \n");
fgets(fullName, sizeof(fullName), stdin);
printf("Hello %s", fullName);
// Type your full name: John Doe
// Hello John Doe
```

Use the scanf() function to get a single word as input, and use fgets() for multiple words

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:

Example

```
int myAge = 43;
printf("%p", &myAge); // Outputs 0x7ffe5367e044
```

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

We 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.

We will learn much more about pointers in the next tutorial.

Why is it useful to know the memory address?

Pointers are important in C, because they allow us to manipulate the data in the computer's memory - **this can reduce the code and improve the performance**.

Pointers are one of the things that make C stand out from other programming languages, like Python and Java.

Creating Pointers

We learned from the previous tutorial, that we can get the **memory address** of a variable with the reference operator &:

Example

```
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 we are working with is assigned to the pointer:

Example

```
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);
```

Example explained
Create a pointer variable with the name ptr, that points to an int variable (myAge). Note that the type of the pointer has to match the type of the variable we're working with (int in our example).
Use the & operator to store the memory address of the myAge variable, and assign it to the pointer.
Now, ptr holds the value of myAge's memory address.

Dereference

In the example above, we used the pointer variable to get the memory address of a variable (used together with the & **reference** operator).
We can also get the value of the variable the pointer points to, by using the * operator (the **dereference operator**):

Example

```
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);
```

Note that the * sign can be confusing here, as it does two different things in our code:

- When used in declaration (int* ptr), it creates a **pointer variable**.
- When not used in declaration, it act as a **dereference operator**.

Good To Know: There are two ways to declare pointer variables in C:

```
int* myNum;
int *myNum;
```

Notes on Pointers
Pointers are one of the things that make C stand out from other programming languages, like Python and Java.
They are important in C, because they allow us to manipulate the data in the computer's memory. This can reduce the code and improve the performance. If you are familiar with data structures like lists, trees and graphs, you should know that pointers are especially useful for implementing those. And sometimes we even have to use pointers, for example when working with files.
But be careful; pointers must be handled with care, since it is possible to damage data stored in other memory addresses.

Pointers & Arrays

We can also use pointers to access arrays.

Consider the following array of integers:

Example

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

We learned from the arrays tutorial that we can loop through the array elements with a for loop:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
int i;
for (i = 0; i < 4; i++) {
printf("%d\n", myNumbers[i]);
 }
```

Result of above program:

```
25
50
75
100
```

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

Example

```
int myNumbers[4] = {25, 50, 75, 100};
int i;
for (i = 0; i < 4; i++) {
printf("%p\n", &myNumbers[i]);
}
```

Result: of the above program is:

```
0x7ffe70f9d8f0
0x7ffe70f9d8f4
0x7ffe70f9d8f8
0x7ffe70f9d8fc
```

Note that the last number of each of the elements' memory address is different, with an addition of 4.

It is because the size of an int type is typically 4 bytes, remember:

Example

```
// Create an int variable
int myInt;
// Get the memory size of an int
printf("%lu", sizeof(myInt));
```

Result of the above program is:

```
4
```

So from the "memory address example" above, we can see that the compiler reserves 4 bytes of memory for each array element, which means that the entire array takes up 16 bytes (4 * 4) of memory storage:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
// Get the size of the myNumbers array
printf("%lu", sizeof(myNumbers));
```

Result of the above program is:

```
16
```

How Are Pointers Related to Arrays

Ok, so what's the relationship between pointers and arrays? Well, in C, the **name of an array**, is actually a **pointer** to the **first element** of the array.
Confused? Let's try to understand this better, and use our "memory address example" above again.
The **memory address** of the **first element** is the same as the **name of the array**:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
// Get the memory address of the myNumbers array
printf("%p\n", myNumbers);
// Get the memory address of the first array element
printf("%p\n", &myNumbers[0]);
```

Result of the above program is:

```
0x7ffe70f9d8f0
0x7ffe70f9d8f0
```

This basically means that we can work with arrays through pointers!

How? Since myNumbers is a pointer to the first element in myNumbers, we can use the * operator to access it:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
// Get the value of the first element in myNumbers
printf("%d", *myNumbers);
```

Result of the aboive program is:

```
25
```

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

Example

```
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));
// and so on..
```

Result of the above program is:

```
50
75
```

or loop through it:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
int *ptr = myNumbers;
 int i;
 for (i = 0; i < 4; i++) {
 printf("%d\n", *(ptr + i));
  }
```

Result of the above program is:

```
25
50
75
100
```

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

Example

```
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));
```

Result of the above program is:

```
13
17
```

This way of working with arrays might seem a bit excessive. Especially with simple arrays like in the examples above. However, for large arrays, it can be much more efficient to access and manipulate arrays with pointers.

It is also considered faster and easier to access two-dimensional arrays with pointers.

And since strings are actually arrays, we can also use pointers to access strings.

For now, it's great that we know how this works. But like we specified in the previous tutorial; **pointers must be handled with care**, since it is possible to overwrite other data stored in memory.

C Functions

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

We 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.

Predefined Functions

So it turns out we already know what a function is. We have been using it the whole time while studying this tutorial!

For example, main() is a function, which is used to execute code, and printf() is a function; used to output/print text to the screen:

Example

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

Create a Function

To create (often referred to as declare) our own function, specify the name of the function, followed by parentheses () and curly brackets {}:

Syntax

```
void myFunction() {
// code to be executed
 }
```

Example Explained

- myFunction() is the name of the function
- void means that the function does not have a return value. We will learn more about return values later in the next tutorial.
- Inside the function (the body), add code that defines what the function should do

Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed when they are called.

To call a function, write the function's name followed by two parentheses () and a semicolon ;
In the following example, myFunction() is used to print a text (the action), when it is called:

Example

Inside main, call myFunction():

```
// 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:

```
Example
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. We can add as many parameters as we want, just separate them with a comma:

Syntax

```
 returnType functionName(parameter1, parameter2, parameter3) {
// code to be executed
 }
```

The following function that takes a string of characters with **name** as parameter. When the function is called, we pass along a name, which is used inside the function to print "Hello" and the name of each person.

Example

```
void myFunction(char name[]) {
printf("Hello %s\n", name);
}
int main() {
 myFunction("Liam");
  myFunction("Jenny");
 myFunction("Anja");
  return 0;
 }
// Hello Liam
```

```
// Hello Jenny
// Hello Anja
```

When a **parameter** is passed to the function, it is called an **argument**. So, from the example above: name is a **parameter**, while Liam, Jenny and Anja are **arguments.**

Multiple Parameters

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

Example

```
void myFunction(char name[], int age) {
printf("Hello %s. You are %d years old.\n", name, age);
}
int main() {
myFunction("Liam", 3);
myFunction("Jenny", 14);
myFunction("Anja", 30);
return 0;
}
// Hello Liam. You are 3 years old.
// Hello Jenny. You are 14 years old.
// Hello Anja. You are 30 years old.
```

Note that when we are working with multiple parameters, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Pass Arrays as Function Parameters

We can also pass arrays to a function:

Example

```
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;
}
```

Example Explained

The function (myFunction) takes an array as its parameter (int myNumbers[5]), and loops through the array elements with the for loop.

When the function is called inside main(), we pass along the myNumbers array, which outputs the array elements.

Note that when we call the function, we only need to use the name of the array when passing it as an argument myFunction(myNumbers). However, the full declaration of the array is needed in the function parameter (int myNumbers[5]).

Return Values

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

Example

```
int myFunction(int x) {
```

```
 return 5 + x;
 }
 int main() {
 printf("Result is: %d", myFunction(3));
 return 0;
}
// Outputs 8 (5 + 3)
```

This example returns the sum of a function with **two parameters**:

Example

```
int myFunction(int x, int y) {
 return x + y;
 }
 int main() {
 printf("Result is: %d", myFunction(5, 3));
  return 0;
 }
  // Outputs 8 (5 + 3)
```

We can also store the result in a variable:

Example

```
 int myFunction(int x, int y) {
  return x + y;
 }
 int main() {
 int result = myFunction(5, 3);
 printf("Result is = %d", result);
 return 0;
 }
// Outputs 8 (5 + 3)
```

Function Declaration and Definition

We just learned from the previous tutorial that we can create and call a function in the following way:

Example

```
// Create a function
void myFunction() {
printf("I just got executed!");
}
int main() {
myFunction(); // call the function
 return 0;
 }
```

A function consists of two parts:

- **Declaration:** the function's name, return type, and parameters (if any)
- **Definition:** the body of the function (code to be executed)

```
void myFunction() { // declaration
// the body of the function (definition)
 }
```

For code optimization, it is recommended to separate the declaration and the definition of the function.

We will often see C programs that have function declaration above main(), and function definition below main(). This will make the code better organized and easier to read:

Example

```
// Function declaration
void myFunction();
// The main method
int main() {
 myFunction();  // call the function
return 0;
}
// Function definition
void myFunction() {
printf("I just got executed!");
}
```

Another Example

If we use the example from the previous tutorial regarding function parameters and return values:

Example

```
int myFunction(int x, int y) {
return x + y;
 }
int main() {
 int result = myFunction(5, 3);
 printf("Result is = %d", result);
 return 0;
 }
// Outputs 8 (5 + 3)
```

It is considered good practice to write it like this instead:

Example

```
// Function declaration
int myFunction(int, int);
 // The main method
int main() {
 int result = myFunction(5, 3); // call the function
 printf("Result is = %d", result);
 return 0;
 }
// Function definition
int myFunction(int x, int y) {
return x + y;
 }
```

Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

Example

```
int sum(int k);
int main() {
```

```
int result = sum(10);
printf("%d", result);
return 0;
}
int sum(int k) {
if (k > 0) {
return k + sum(k - 1);
 } else {
return 0;
 }
 }
```

Example Explained

When the sum() function is called, it adds parameter k to the sum of all numbers smaller than k and returns the result. When k becomes 0, the function just returns 0. When running, the program follows these steps:

10 + sum(9)
10 + (9 + sum(8))
10 + (9 + (8 + sum(7)))
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0

Since the function does not call itself when k is 0, the program stops there and returns the result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

Math Functions

There is also a list of math functions available, that allows us to perform mathematical tasks on numbers.

To use them, we must include the math.h header file in our program:

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

Square Root

To find the square root of a number, use the sqrt() function:

Example

```
printf("%f", sqrt(16));
```

Round a Number

The ceil() function rounds a number upwards to its nearest integer, and the floor() method rounds a number downwards to its nearest integer, and returns the result:

Example

```
printf("%f", ceil(1.4));
printf("%f", floor(1.4));
```

Power

The pow() function returns the value of x to the power of y (xy):

Example

```
printf("%f", pow(4, 3));
```

Other Math Functions

A list of other popular math functions (from the <math.h> library) can be found in the table below:

Function	Description
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x
asin(x)	Returns the arcsine of x
atan(x)	Returns the arctangent of x
cbrt(x)	Returns the cube root of x
cos(x)	Returns the cosine of x
exp(x)	Returns the value of E^x
sin(x)	Returns the sine of x (x is in radians)
tan(x)	Returns the tangent of an angle

File Handling

In C, we 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). For now, this line is not important. It's just something we need when working with files.

To actually open a file, use the fopen() function, which takes two parameters:

Parameter	Description
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

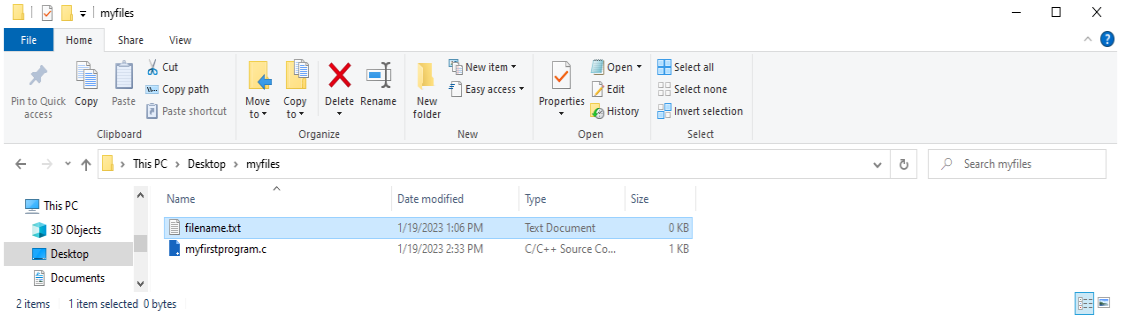
To create a file, we can use the w mode inside the fopen() function.

The w mode is used to write to a file. **However**, if the file does not exist, it will create one for us:

Example

```
FILE *fptr;
// Create a file
fptr = fopen("filename.txt", "w");
// Close the file
fclose(fptr);
```

Note: The file is created in the same directory as our other C files, if nothing else is specified. On our computer, it looks like this:



Tip: If we want to create the file in a specific folder, just provide an absolute path:

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

Closing the file

Did you notice the fclose() function in our example above?

This will close the file when we are done with it.

It is considered as good practice, because it makes sure that:

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

In the next tutorial, we will learn how to write content to a file and read from it.

Write To a File

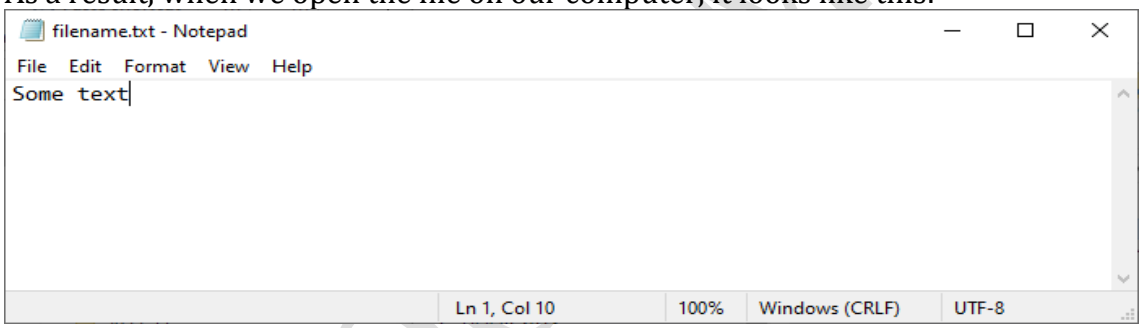
Let's use the w mode from the previous tutorial again, and write something to the file we just created.

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

Example

```
FILE *fptr;
// Open a file in writing mode
fptr = fopen("filename.txt", "w");
// Write some text to the file
fprintf(fptr, "Some text");
// Close the file
fclose(fptr);
```

As a result, when we open the file on our computer, it looks like this:



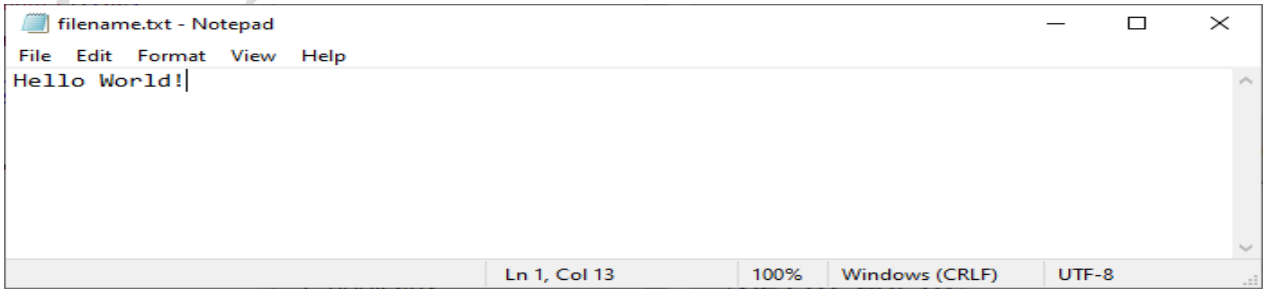
Note: If we write to a file that already exists, the old content is deleted, and the new content is inserted. This is important to know, as we might accidentally erase existing content.

For example:

Example

```
fprintf(fptr, "Hello World!");
```

As a result, when we open the file on our computer, it says "Hello World!" instead of "Some text":



Append Content To a File

If we want to add content to a file without deleting the old content, we can use the a mode.

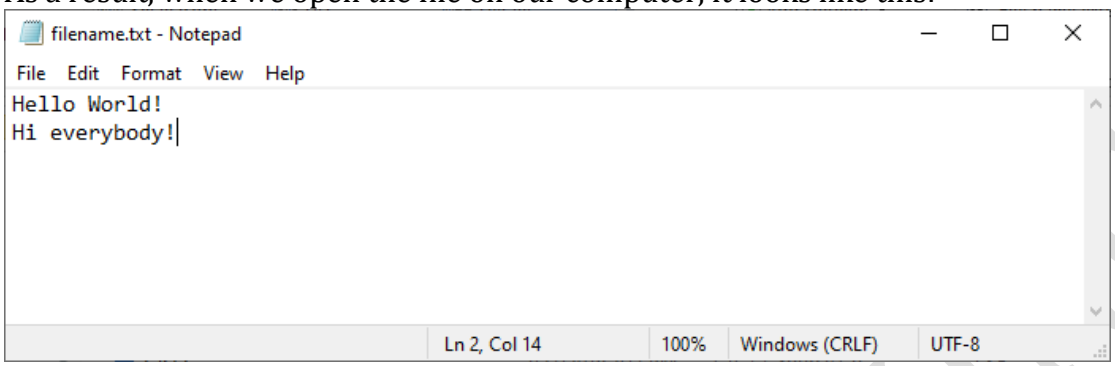
The a mode appends content at the end of the file:

Example

```
FILE *fptr;
```

```
// Open a file in append mode
fptr = fopen("filename.txt", "a");
// Append some text to the file
fprintf(fptr, "\nHi everybody!");
// Close the file
fclose(fptr);
```

As a result, when we open the file on our computer, it looks like this:



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

In the previous tutorial, we wrote to a file using w and a modes inside the fopen() function.

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

Example

```
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:

Example

```
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:

Example

```
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).

Now, we can print the string, which will output the content of the file:

Example

```
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 the file content
printf("%s", myString);
// Close the file
fclose(fptr);
```

The output of the above program is:

```
Hello World!
```

Note: The fgets function only reads the first line of the file. If we remember, there were two lines of text in filename.txt.

To read every line of the file, we can use a while loop:

Example

```
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);
```

The output of the above program is:

```
Hello World!
Hi everybody!
```

Good Practice

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

Tip: 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):

Example

```
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);
```

If the file does not exist, the following text is printed as the output:

```
Not able to open the file.
```

With this in mind, we can create a more sustainable code if we use our "read a file" example above again:

Example

If the file exist, read the content and print it. If the file does not exist, print a message:

```
FILE *fptr;
// Open a file in read mode
fptr = fopen("filename.txt", "r");
// Store the content of the file
```

```
char myString[100];
// If the file exist
 if(fptr != NULL) {
// Read the content and print it
while(fgets(myString, 100, fptr)) {
 printf("%s", myString);
 }
// If the file does not exist
} else {
printf("Not able to open the file.");
}
// Close the file
 fclose(fptr);
```

The output of the above program is:

```
Hello World!
Hi everybody!
```

Structures

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.).

Create a Structure

We can create a structure by using the struct keyword and declare each of its members inside curly braces:

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

To access the structure, we must create a variable of it.
Use the struct keyword inside the main() method, followed by the name of the structure and then the name of the structure variable:

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 (.):

Example

```
// 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;
 }
```

Now we can easily create multiple structure variables with different values, using just one structure:

Example

```
// Create different struct variables
 struct myStructure s1;
 struct myStructure s2;
// Assign values to different struct variables
 s1.myNum = 13;
 s1.myLetter = 'B';
 s2.myNum = 20;
s2.myLetter = 'C';
```

What About Strings in Structures?

Remember that strings in C are actually an array of characters, and unfortunately, we can't assign a value to an array like this:

Example

```
struct myStructure {
int myNum;
char myLetter;
 char myString[30];  // String
};
int main() {
struct myStructure s1;
 // Trying to assign a value to the string
  s1.myString = "Some text";
// Trying to print the value
   printf("My string: %s", s1.myString);
  return 0;
  }
```

The above program will show the following error:

```
prog.c:12:15: error: assignment to expression with array type
```

However, there is a solution for this! We can use the strcpy() function and assign the value to s1.myString, like this:

Example

```
 struct myStructure {
 int myNum;
 char myLetter;
char myString[30]; // String
 };
int main() {
struct myStructure s1;
  // Assign a value to the string using the strcpy function
 strcpy(s1.myString, "Some text");
 // Print the value
```

```
printf("My string: %s", s1.myString);
return 0;
}
```

Result of above program is:

My string: Some text

Simpler Syntax

We can also assign values to members of a structure variable at declaration time, in a single line.

Just insert the values in a comma-separated list inside curly braces {}. Note that we don't have to use the strcpy() function for string values with this technique:

Example

```
// Create a structure
struct myStructure {
int myNum;
char myLetter;
char myString[30];
};
int main() {
// Create a structure variable and assign values to it
struct myStructure s1 = {13, 'B', "Some text"};
// Print values
printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);
return 0;
}
```

Note: The order of the inserted values must match the order of the variable types declared in the structure (13 for int, 'B' for char, etc).

Copy Structures

We can also assign one structure to another.

In the following example, the values of s1 are copied to s2:

Example

```
struct myStructure s1 = {13, 'B', "Some text"};
struct myStructure s2;
s2 = s1;
```

Modify Values

If we want to change/modify a value, we can use the dot syntax (.).

And to modify a string value, the strcpy() function is useful again:

Example

```
struct myStructure {
int myNum;
char myLetter;
char myString[30];
};
int main() {
// Create a structure variable and assign values to it
struct myStructure s1 = {13, 'B', "Some text"};
// Modify values
s1.myNum = 30;
s1.myLetter = 'C';
strcpy(s1.myString, "Something else");
```

```
// Print values
printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);
return 0;
 }
```

Modifying values are especially useful when we copy structure values:

Example

```
// Create a structure variable and assign values to it
struct myStructure s1 = {13, 'B', "Some text"};
// Create another structure variable
 struct myStructure s2;
// Copy s1 values to s2
s2 = s1;
// Change s2 values
 s2.myNum = 30;
s2.myLetter = 'C';
strcpy(s2.myString, "Something else");
// Print values
printf("%d %c %s\n", s1.myNum, s1.myLetter, s1.myString);
printf("%d %c %s\n", s2.myNum, s2.myLetter, s2.myString);
```

Ok, so, how are structures useful?

Imagine we have to write a program to store different information about Cars, such as brand, model, and year. What's great about structures is that we can create a single "Car template" and use it for every cars you make. See below for a real life example.

Real Life Example

Use a structure to store different information about Cars:

Example

```
struct Car {
char brand[50];
char model[50];
 int year;
 };
int main() {
 struct Car car1 = {"BMW", "X5", 1999};
 struct Car car2 = {"Ford", "Mustang", 1969};
struct Car car3 = {"Toyota", "Corolla", 2011};
 printf("%s %s %d\n", car1.brand, car1.model, car1.year);
 printf("%s %s %d\n", car2.brand, car2.model, car2.year);
 printf("%s %s %d\n", car3.brand, car3.model, car3.year);
return 0;
}
```

C Enumeration (enum)

An **enum** is a special type that represents a group of constants (unchangeable values).

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
 };
```

Note that the last item does not need a comma.

It is not required to use uppercase, but often considered as good practice.

Enum is short for "enumerations", which means "specifically listed".
To access the enum, we must create a variable of it.
Inside the main() method, specify the enum keyword, followed by the name of the enum (Level) and then the name of the enum variable (myVar in this example):

```
enum Level myVar;
```

Now that we have created an enum variable (myVar), we can assign a value to it.
The assigned value must be one of the items inside the enum (LOW, MEDIUM or HIGH):

```
enum Level myVar = MEDIUM;
```

By default, the first item (LOW) has the value 0, the second (MEDIUM) has the value 1, etc.
If we now try to print myVar, it will output 1, which represents MEDIUM:

```
int main() {
// Create an enum variable and assign a value to it
 enum Level myVar = MEDIUM;
// Print the enum variable
printf("%d", myVar);
 return 0;
```

Change Values

As we know, the first item of an enum has the value 0. The second has the value 1, and so on.
To make more sense of the values, we can easily change them:

```
enum Level {
 LOW = 25,
MEDIUM = 50,
 HIGH = 75
  };
printf("%d", myVar); // Now outputs 50
```

Note that if we 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
};
```

Enum in a Switch Statement

Enums are often used in switch statements to check for corresponding values:

```
 enum Level {
LOW = 1,
MEDIUM,
HIGH
};
int main() {
enum Level myVar = MEDIUM;
switch (myVar) {
case 1:
printf("Low Level");
break;
case 2:
printf("Medium level");
break;
case 3:
printf("High level");
 break;
```

```
 }
return 0;
}
```

Why And When To Use Enums?

Enums are used to give names to constants, which makes the code easier to read and maintain.

Use enums when we have values that we know aren't going to change, like month days, days, colors, deck of cards, etc.