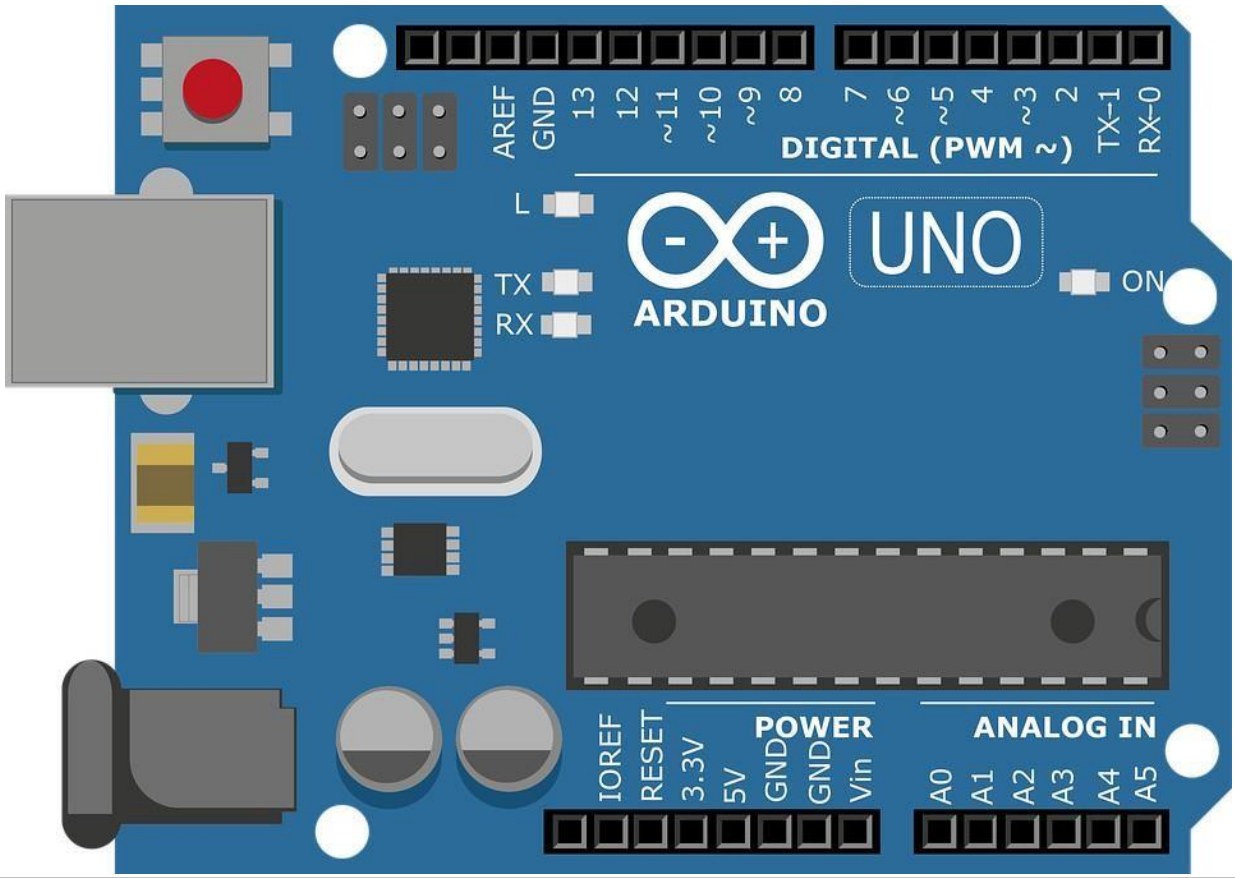
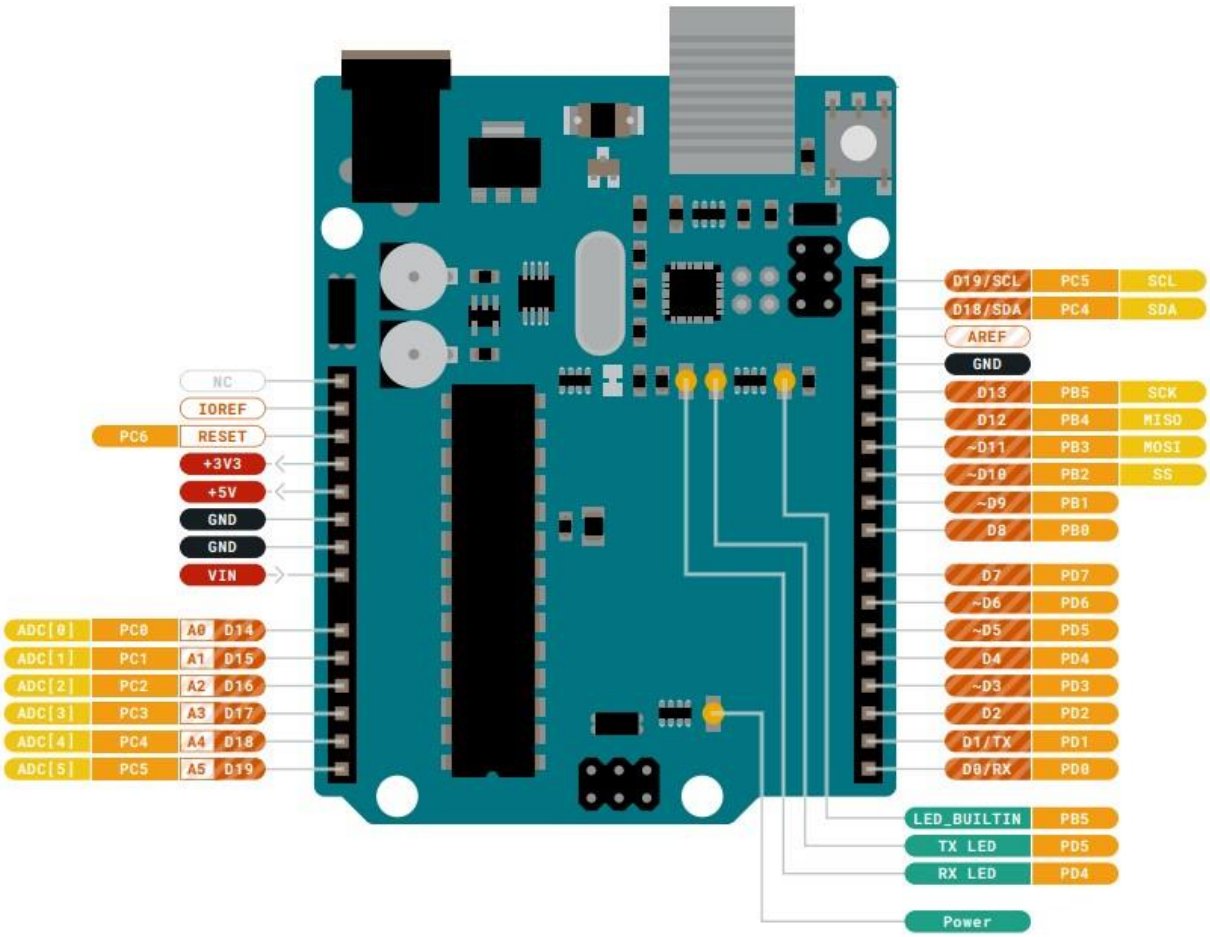
Learning Arduino



Microcontroller	ATmega328P
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	20 Ma

DC Current for 3.3V Pin	50 Ma
Flash Memory	32 KB (ATmega328P) of which 0.5 KB used by boot loader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Clock Speed	16 MHz
LED_BUILTIN	13



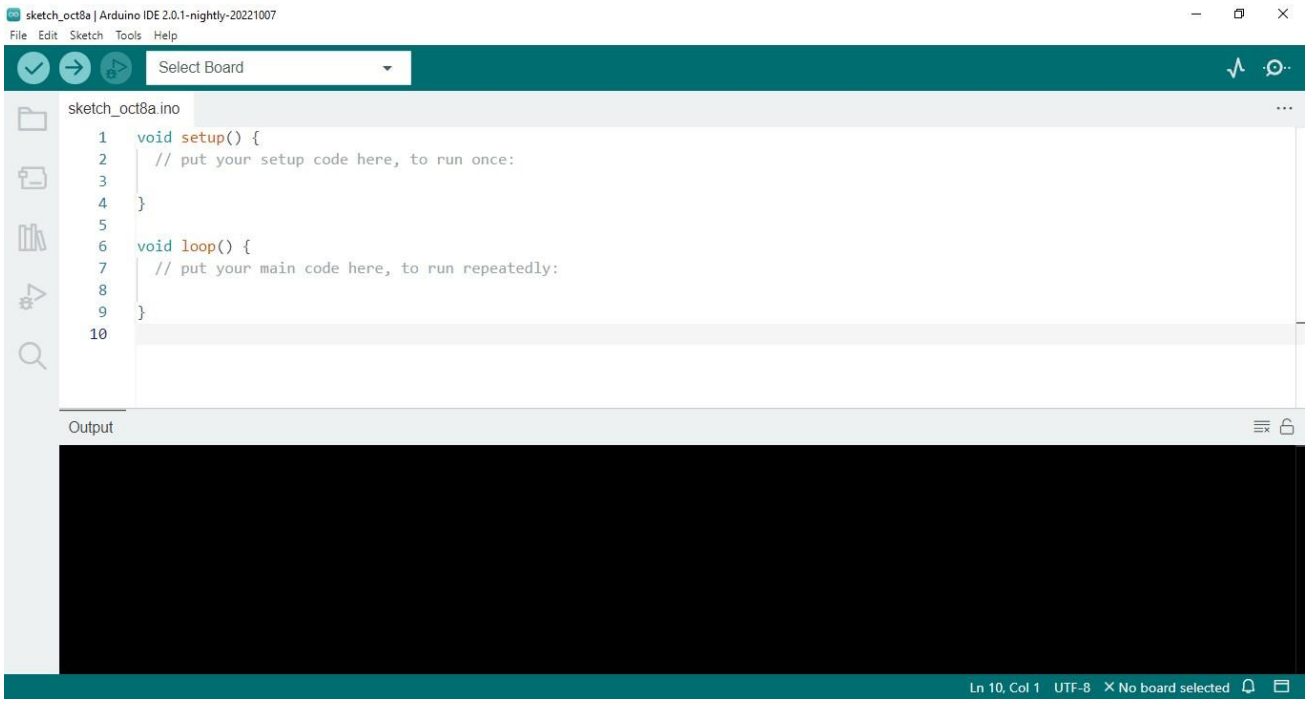
Different Types of Arduino Boards

- Arduino Uno (R3)
- Arduino Nano
- Arduino Micro
- Arduino Due
- LilyPad Arduino Board

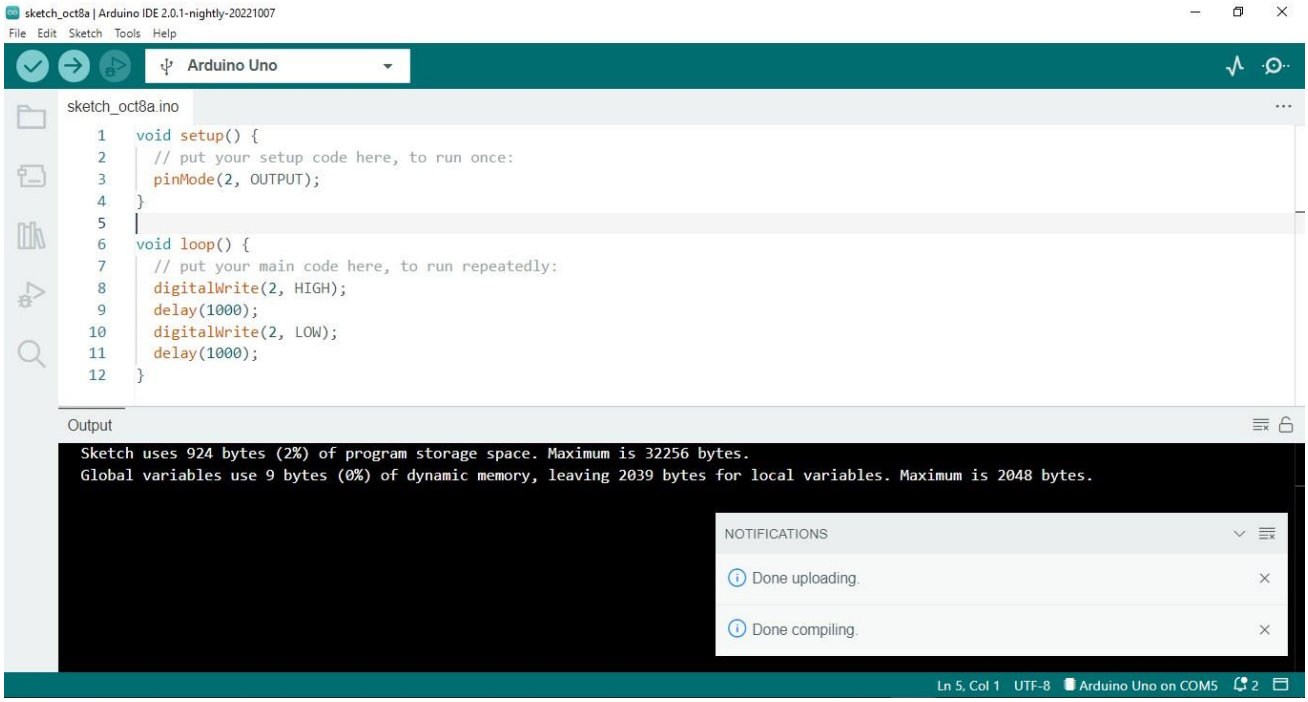
- Arduino Bluetooth
- Arduino Diecimila
- RedBoard Arduino Board
- Arduino Mega (R3) Board
- Arduino Leonardo Board
- Arduino Robot
- Arduino Esplora
- Arduino Pro Mic
- Arduino Ethernet
- Arduino Zero
- Fastest Arduino Board

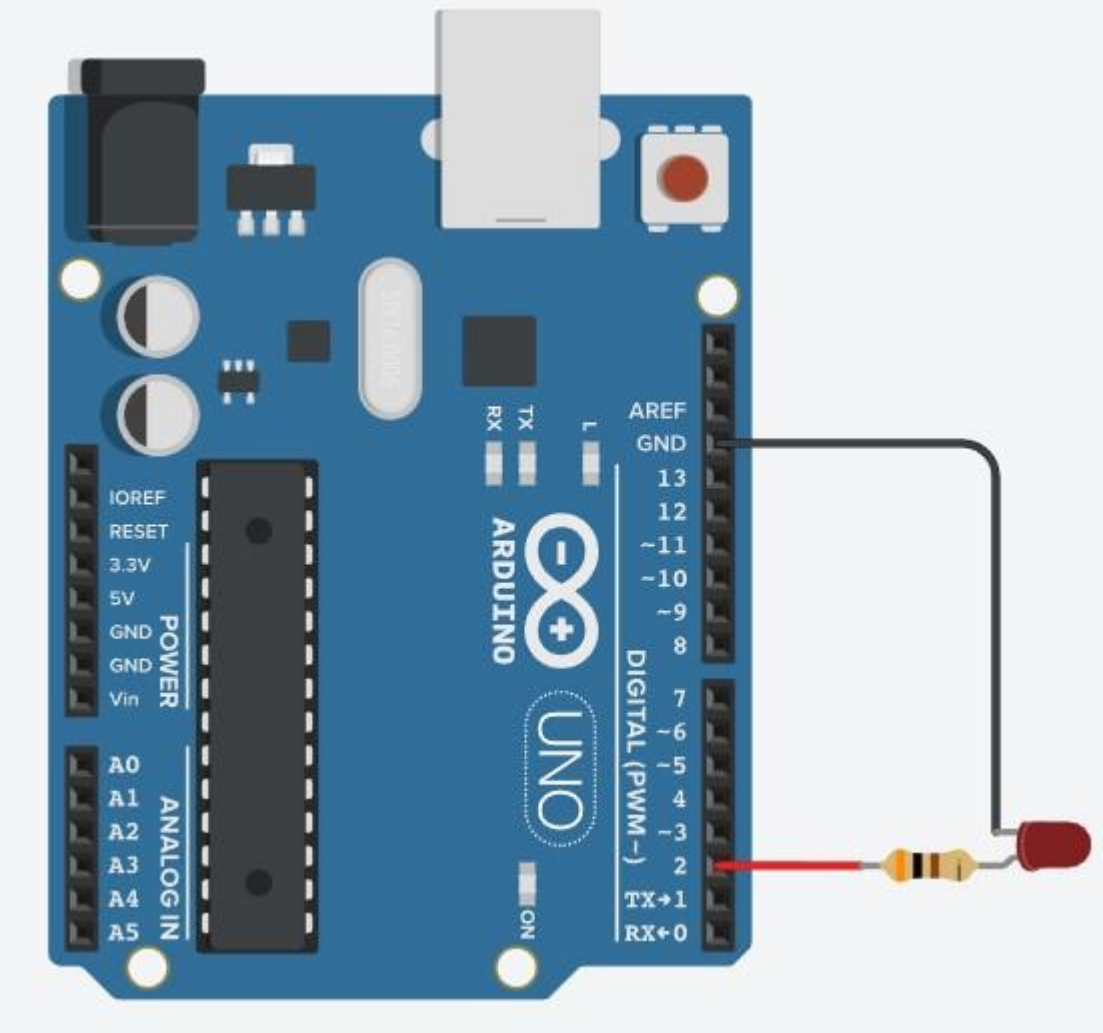
Introduction to Arduino IDE

The Arduino IDE is **an open-source software, which is used to write and upload code to the Arduino boards**. The IDE application is suitable for different operating systems such as Windows, Mac OS X, and Linux. It supports the programming languages C and C++. Here, IDE stands for Integrated Development Environment.



Simple Program LED Blinking

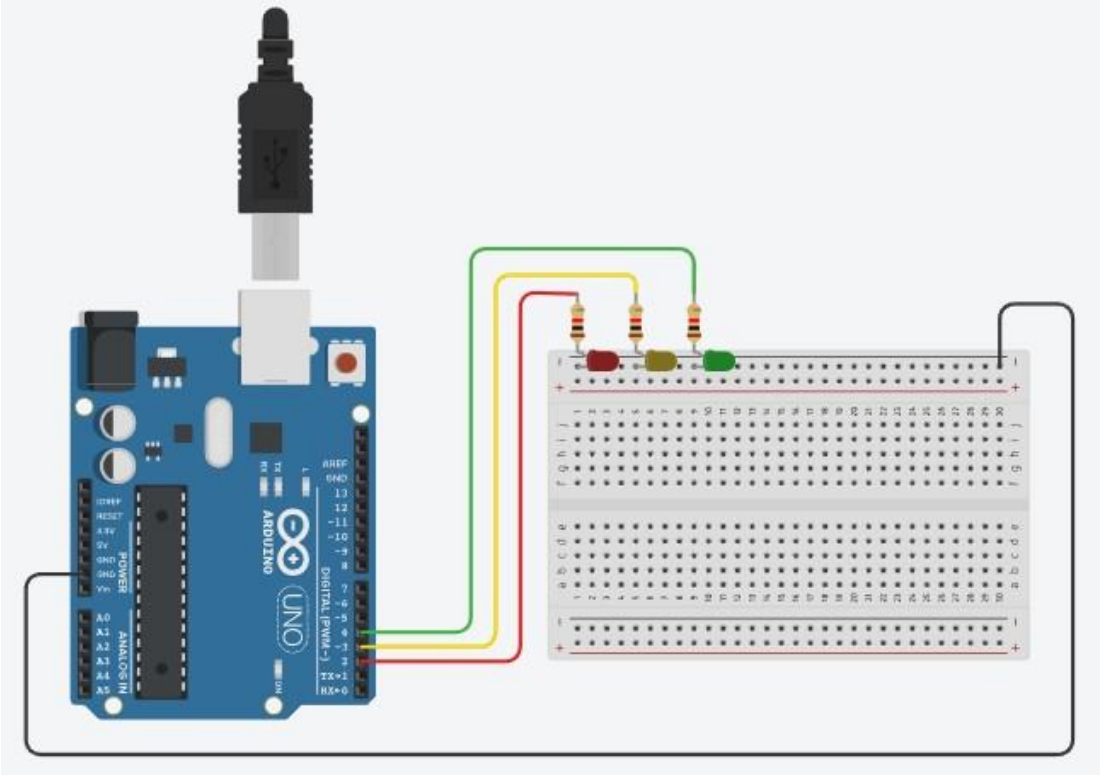




Simple Traffic Light Control Program

Code:

```
int red = 2 ;  int
yellow = 3 ;  int
green = 4 ;
void setup(){  pinMode(red,
        OUTPUT);
        pinMode(yellow, OUTPUT);
        pinMode(green, OUTPUT);
}
void loop(){  digitalWrite(red,
        1);  delay(4000);
        digitalWrite(yellow, 1);
        delay(1000);
        digitalWrite(red, 0);
        digitalWrite(yellow, 0);
        digitalWrite(green, 1);
        delay(5000);
        digitalWrite(green, 0);
}
```



Embedded ‘C’ Language

Embedded C is most popular programming language in software field for developing electronic gadgets. Each processor used in electronic system is associated with embedded software. Embedded C programming plays a key role in performing specific function by the processor.

The language in which Arduino is programmed is a subset of C and it includes only those features of standard C that are supported by the Arduino IDE.

Differences between C and Embedded C:

Parameters	C	Embedded C
GENERAL	• C is a general purpose programming language, which can be used to design any type of desktop based applications. • It is a type of high level language.	• Embedded C is simply an extension C language and it is used to develop microcontroller based applications. It is nothing • but an extension of C.

DEPENDENCY	C language is hardware independent language. C compilers are OS dependent.	Embedded C is fully hardware dependent language. Embedded C are OS independent.
COMPILER	For C language, the standard compilers can be used to compile and execute the program. Popular Compiler to execute a C language program are: • GCC (GNU Compiler collection) • Borland turbo C, • Intel C++	For Embedded C, a specific compilers that are able to generate particular hardware/micro-controller based output is used. Popular Compiler to execute a Embedded C language program are: • Keil compiler • BiPOM ELECTRONIC • Green Hill software
USABILITY AND APPLICATION	C language has a freeformat of program coding. It is specifically used for desktop application. Optimization is normal.	Formatting depends upon the type of microprocessor that is used. It is used for limited resources like RAM and ROM.

	□ It is very easy to read and modify the C language. □ Bug fixing are very easy in a C language program. It □ supports other various programming languages during application. Input □ can be given to the program while it is running. □ Applications of C Program: • Logical programs • System software programs	□ High level of optimization. It □ is not easy to read and modify the Embedded C language. □ Bug fixing is complicated in a Embedded C language program. □ It supports only required processor of the application, and not the programming languages. □ Only the pre-defined input can be given to the running program. □ Applications of Embedded C Program: • DVD • TV • Digital camera

C Introduction

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 language in the world
- If you know C, you will have no problem learning other popular 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 and Python
- C is very versatile; it can be used in both applications and technologies

Difference between C and C++

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

Get Started

It is not necessary to have any prior programming experience.

To start using C, you 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 computer 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.

We will use **Code::Blocks** in our tutorial, which we believe is a good place to start.

You 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 Syntax

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

Example

```
 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.

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 **function** 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.

Line 5: `return 0` ends the `main()` function.

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

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

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

This example uses a single-line comment at the end of 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!
to the screen, and it is amazing */ printf("Hello
World!");
```

C reserved keywords

The table below lists all keywords reserved by the C language. When the current programming language is C or C++, these keywords cannot be abbreviated, used as variable names, or used as any other type of identifiers.

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

C Variables

Variables are containers for storing data values.

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;
```

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

Example

```
int myNum; myNum = 15;
```

Note: If you 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
```

Output Variables

You 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++), you 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, you 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, you 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 = 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);
```

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

Example

```
int myNum = 5;
printf("My favorite number is: %d", myNum);
```

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

Example

```
int myNum = 5; char myLetter = 'D'; printf("My number is %d and
my letter is %c", myNum, myLetter);
```

You will learn more about Data Types in the next chapter.

Add Variables 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);
```

You 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**.

These unique names are called **identifiers**.

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:

```
// Good
```

Example

```
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

Data Types

As explained in the Variables chapter, a variable in C must be a specified **data type**, and you 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 7 decimal digits

Example

`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**, which you will learn more about in a later chapter

Constants

When you don't want others (or yourself) to override existing variable values, 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'
```

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

Example

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

Notes on Constants

When you 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 = 1980;
```

User Input

You 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 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.

Tip: You will learn more about memory addresses and functions in the next chapter.

User Input Strings

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

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 += 3	x = x + 3
-=	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
|=	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.

Note: The return value of a comparison is either true (`1`) or false (`0`).

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	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

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

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	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));
```

Conditions and If Statements

You learned from the operators comparison chapter, 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`

You 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 C 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 you can use if..else if 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."); }
```

You can simply write:

Example

```
int time = 20;
```

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

Switch Statement

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:

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
{
prin
tf("
%d\n
",
i);
```

```
i++;
}
while (i < 5);
```

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

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:

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); }
```

Break

You have already seen the `break` statement used in an earlier chapter 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 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

You 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++; }
```

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. However, you can 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:

Example

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

Access Strings

Since strings are actually arrays in C, you 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!
```

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.

You should also note that you can to create a string with a set of characters. This example will produce the same result as the one above:

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 you do not have to include the `\0` character, as C will do it for you.

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
```

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 will represent where the variable is stored:

Example

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

Note: The memory address is in hexadecimal form (0x..). You probably won't get the same result in your program.

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.

Creating Pointers

You learned from the previous chapter, 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)
```

In the example above, `&myAge` is also known as a **pointer**.

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're 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 you're working with.

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).

However, you 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**.

Why Should I Learn About Pointers? Pointers are important in C, because they give you the ability 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.

Note: Pointers must be handled with care, since it is possible to damage data stored in other memory addresses.

Good To Know: There are three ways to declare pointer variables, but the first way is mostly used:

```
int* myNum; // Most used
int *myNum;
int * myNum;
```

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

You 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, **you should know the size of the array,** in order for the program to store enough memory.

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

C 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.

Predefined Functions

So it turns out you already know what a function is. You 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*) your 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. You will learn more about return values later in the next chapter
- 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. You can add as many parameters as you 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, you can add as many parameters as you 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 you 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.

Return Values

The `void` keyword, used in the previous examples, 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:

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)
```

You 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

You just learned from the previous chapters that you can create and call a function it the following way:

Example

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

A function consist 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.

You 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 chapter 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; }
```

Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

```
#include <stdio.h> int addNumbers(int a, int b);
// function prototype
 int
main()
{     int
n1,n2,sum;
     printf("Enters two numbers:
");     scanf("%d %d",&n1,&n2);
     sum = addNumbers(n1, n2);        // function
call     printf("sum = %d",sum);
     return
0;
}  int addNumbers(int a, int b)         // function
definition
{     int result;     result = a+b;     return result;
// return statement }
```

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.