

*For everyone who contributed to the  
publication of this book*

## Table of Contents

C Programming Language .....	1
C++ Programming Language .....	136
C# Programming Language .....	229
Reference .....	351

## List of Abbreviation

Abbreviation	Meaning
ECMA	European Computer Manufacturers Association
ISO	<b>International Standards Organization</b>
CLI	Command Line Interface
C#	<b>C-Sharp</b>

# Introduction to Book

**C** is a general-purpose programming language that is extremely popular, simple, and flexible. It is machine-independent, structured programming language which is used extensively in various applications.

**C** is middle-level programming language which was developed at **Bell Lab** in **1972** by **Dennis Ritchie**. **C language** combines the features of Low level as well as High-level Language. Hence its considered a middle-level Language.

**C** is a high-level classical type programming language that allows you to develop firmware and portable applications. The **C language** was developed with an objective of writing system software. It is an ideal language for developing firmware systems.

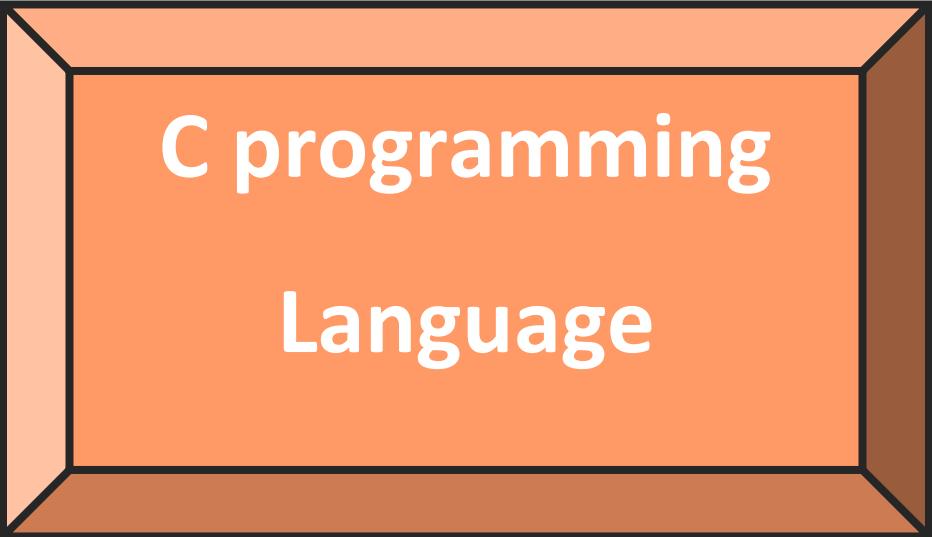
**C++** is a computer programming language that contains the feature of C programming language as well as Simula67(a first object Oriented language). **C++** introduced the concept of Class and Objects.

It encapsulates high and low-level language features. So, it is seen as an intermediate level language. Earlier it was called "C with classes" as it had all the properties of the C language.

**C#** is a general-purpose, modern and object-oriented programming language pronounced as "C sharp". It was developed by Microsoft led by Anders Hejlsberg and his team within the .Net initiative and was approved by the European Computer Manufacturers Association (**ECMA**) and International Standards Organization (**ISO**).

**C#** was designed for complex computational logic and was mainly designed for Command Line Interface (**CLI**) that deals with executable code in a runtime environment. The term **C#** is pronounced as C-Sharp, which runs on Dot Net Framework. The latest version of **C#** is **C# 7.0**.

In this book, three important languages are explained: **C**, **C ++**, and **C #**. The basics of these three languages will be explained, which can benefit those who want to enter the programming world.



**C programming**

**Language**

# C Programming Language

## C INTRODUCTION

- C Keywords and Identifiers

### Character set

A character set is a set of alphabets, letters and some special characters that are valid in C language.

### Alphabets

Uppercase: A B C .....	X Y Z
Lowercase: a b c .....	x y z

C accepts both lowercase and uppercase alphabets as variables and functions.

### Digits

0 1 2 3 4 5 6 7 8 9
---------------------

### Special Characters

Special Characters in C Programming

,	<	>	.	-
(	)	;	\$	:
%	[	]	#	?
'	&	{	}	"
^	!	*	/	
-	\	*	+	

# C Programming Language

## White space Characters

Blank space, newline, horizontal tab, carriage, return and form feed.

## C Keywords

**Keywords** are predefined, reserved words used in programming that have special meanings to the **compiler**. **Keywords** are part of the syntax and they cannot be used as an identifier. For example:

```
int money;
```

Here, **int** is a keyword that indicates **money** is a variable of type **int (integer)**.

As **C** is a case sensitive language, all keywords must be written in **lowercase**. Here is a list of all keywords allowed in **ANSI C**.

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

# C Programming Language

## C Identifiers

**Identifier** refers to name given to entities such as **variables**, **functions**, **structures** etc.

Identifiers must be **unique**. They are created to give a unique name to an entity to identify it during the execution of the program. For example:

```
int money;  
double accountBalance;
```

Here, **money** and **accountBalance** are identifiers.

Also remember, identifier names must be **different from keywords**. You cannot use **int** as an identifier because **int** is a keyword.

## Rules for naming identifiers

1. A **valid identifier** can have letters (both uppercase and lowercase letters), digits and underscores.
2. The first letter of an identifier should be either a **letter** or an **underscore**.
3. You cannot use keywords as identifiers.
4. There is no rule on how long an identifier can be. However, you may run into problems in some compilers if the identifier is **longer than 31 characters**.

You can choose any name as an identifier if you follow the above rule, however, give meaningful names to identifiers that make sense.

## • C Variables, Constants and Literals

### Variables

In programming, a variable is a **container** (storage area) to hold data.

To indicate the storage area, each variable should be given a **unique name** (identifier). Variable names are just the symbolic representation of a **memory location**. For example:

# C Programming Language

```
int playerScore = 95;
```

Here, **playerScore** is a variable of **int** type. Here, the variable is assigned an integer value **95**.

The value of a variable can be changed, hence the name variable.

```
char ch = 'a';
// some code
ch = '1';
```

## Rules for naming a variable

1. A variable name can only have letters (both uppercase and lowercase letters), digits and underscore.
2. The first letter of a variable should be either a letter or an underscore.
3. There is no rule on how long a variable name (identifier) can be. However, you may run into problems in some compilers if the variable name is longer than 31 characters.

**Note:** You should always try to give meaningful names to variables. For example: **firstName** is a better variable name than **fn**.

**C** is a strongly typed language. This means that the variable type cannot be changed once it is declared. For example:

```
int number = 5;      // integer variable
number = 5.5;        // error
double number;       // error
```

Here, the type of **number** variable is **int**. You cannot assign a floating-point (decimal) value **5.5** to this variable. Also, you cannot redefine the data type of the variable to double. By the way, to store the decimal values in C, you need to declare its type to either double or float.

# C Programming Language

## Literals

**Literals** are data used for representing **fixed values**. They can be used directly in the code.

For example: 1, 2.5, 'c' etc.

Here, 1, 2.5 and 'c' are literals. Why? You cannot assign different values to these terms.

### 1. Integers

An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

decimal (base 10)

octal (base 8)

hexadecimal (base 16)

#### For example:

```
Decimal: 0, -9, 22 etc  
Octal: 021, 077, 033 etc  
Hexadecimal: 0x7f, 0x2a, 0x521 etc
```

In C programming, octal starts with a **0**, and hexadecimal starts with a **0x**.

### 2. Floating-point Literals

A **floating-point** literal is a numeric literal that has either a fractional form or an exponent form. For example:

```
-2.0  
0.0000234  
-0.22E-5
```

**Note:** E-5 =  $10^{-5}$

### 3. Characters

A **character literal** is created by enclosing a single character inside single quotation marks. For example: 'a', 'm', 'F', '2', '}' etc.

# C Programming Language

## 4. Escape Sequences

Sometimes, it is necessary to use characters that cannot be typed or has special meaning in **C programming**. For example: newline(enter), tab, question mark etc.

In order to use these characters, escape sequences are used.

Escape Sequences	
Escape Sequences	Character
\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Horizontal tab
\v	Vertical tab
\\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\0	Null character

For example: \n is used for a newline. The backslash \\ causes escape from the normal way the characters are handled by the compiler.

## 5. String Literals

A **string literal** is a sequence of characters enclosed in double-quote marks. For example:

# C Programming Language

```
"good"           //string constant  
""              //null string constant  
"      "         //string constant of six white space  
"x"              //string constant having a single character.  
"Earth is round\n"    //prints string with a newline
```

## Constants

If you want to define a variable whose value cannot be changed, you can use the **const** keyword. This will create a constant. For example,

```
const double PI = 3.14;
```

Notice, we have added keyword **const**.

Here, **PI** is a symbolic constant; its value cannot be changed.

```
const double PI = 3.14;  
PI = 2.9; //Error
```

You can also define a constant using the **#define** preprocessor directive.

## • C Data Types

In **C programming**, data types are declarations for variables. This determines the type and **size of data** associated with variables. For example,

```
int myVar;
```

Here, **myVar** is a variable of **int (integer)** type. The size of int is **4 bytes**.

## Basic types

Here's a table containing commonly used types in **C programming** for quick access.

# C Programming Language

Type	Size (bytes)	Format Specifier
int	at least 2, usually 4	%d
char	1	%c
float	4	%f
double	8	%lf
short int	2 usually	%hd
unsigned int	at least 2, usually 4	%u
long int	at least 4, usually 8	%li
long long int	at least 8	%lli
unsigned long int	at least 4	%lu
unsigned long long int	at least 8	%llu
signed char	1	%c
unsigned char	1	%c
long double	at least 10, usually 12 or 16	%Lf

## int

Integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, 0, -5, 10

We can use int for declaring an integer variable.

```
int id;
```

Here, **id** is a variable of type **integer**.

You can declare multiple variables at once in **C programming**. For example,

# C Programming Language

```
int id, age;
```

The size of int is usually **4 bytes (32 bits)**. And, it can take **2<sup>32</sup>** distinct states from - **2147483648** to **2147483647**.

## float and double

**float** and **double** are used to hold real numbers.

```
float salary;
double price;
```

In **C**, **floating-point numbers** can also be represented in **exponential**. For example,

```
float normalizationFactor = 22.442e2;
```

What's the difference between **float** and **double**?

The size of **float** (single precision float data type) is **4 bytes**. And the size of **double** (double precision float data type) is **8 bytes**.

## char

Keyword **char** is used for declaring **character** type variables. For example,

```
char test = 'h';
```

The size of the character variable is **1 byte**.

## void

**void** is an incomplete type. It means "nothing" or "no type". You can think of void as absent.

For example, if a function is not returning anything, its return type should be **void**.

**Note** that, you cannot create variables of void type.

# C Programming Language

## short and long

If you need to use a **large number**, you can use a type specifier **long**. Here's how:

```
long a;
long long b;
long double c;
```

Here variables **a** and **b** can store **integer values**. And, **c** can store a **floating-point** number.

If you are sure, only a small integer (**[−32,767, +32,767]** range) will be used, you can use **short**.

```
short d;
```

You can always check the size of a variable using the **sizeof()** operator.

```
#include <stdio.h>
int main() {
    short a;
    long b;
    long long c;
    long double d;

    printf("size of short = %d bytes\n", sizeof(a));
    printf("size of long = %d bytes\n", sizeof(b));
    printf("size of long long = %d bytes\n", sizeof(c));
    printf("size of long double= %d bytes\n", sizeof(d));
    return 0;
}
```

## signed and unsigned

In C, signed and unsigned are type modifiers. You can alter the data storage of a data type by using them. For example,

```
unsigned int x;
int y;
```

# C Programming Language

Here, the variable x can hold only zero and positive values because we have used the unsigned modifier.

Considering the size of int is 4 bytes, variable y can hold values from  $-2^{31}$  to  $2^{31}-1$ , whereas variable x can hold values from 0 to  $2^{32}-1$ .

Other data types defined in C programming are:

- **bool Type**
- **Enumerated type**
- **Complex types**

## Derived Data Types

**Data types** that are derived from fundamental data types are derived types. For example: **arrays, pointers, function types, structures**, etc.

- **C Input Output (I/O)**

## C Output

In C programming, **printf()** is one of the main output function. The function sends formatted output to the screen. For example,

### Example 1: C Output

```
#include <stdio.h>
int main()
{
    // Displays the string inside quotations
    printf("C Programming");
    return 0;
}
```

### Output

```
C Programming
```

# C Programming Language

## How does this program work?

- All valid C programs must contain the **main()** function. The code execution begins from the start of the **main()** function.
- The **printf()** is a library function to send formatted output to the screen. The function prints the string inside quotations.
- To use **printf()** in our program, we need to include **stdio.h** header file using the **#include <stdio.h>** statement.
- The **return 0;** statement inside the **main()** function is the "**Exit status**" of the program. It's optional.

## Example 2: Integer Output

```
#include <stdio.h>
int main()
{
    int testInteger = 5;
    printf("Number = %d", testInteger);
    return 0;
}
```

## Output

```
Number = 5
```

We use **%d** format specifier to print **int** types. Here, the **%d** inside the quotations will be replaced by the value of **testInteger**.

## Example 3: float and double Output

# C Programming Language

```
#include <stdio.h>
int main()
{
    float number1 = 13.5;
    double number2 = 12.4;

    printf("number1 = %f\n", number1);
    printf("number2 = %lf", number2);
    return 0;
}
```

## Output

```
number1 = 13.500000
number2 = 12.400000
```

To print float, we use **%f** format specifier. Similarly, we use **%lf** to print double values.

## Example 4: Print Characters

```
#include <stdio.h>
int main()
{
    char chr = 'a';
    printf("character = %c.", chr);
    return 0;
}
```

## Output

```
character = a
```

To print **char**, we use **%c** format specifier.

## C Input

# C Programming Language

In C programming, **scanf()** is one of the commonly used function to take **input** from the user. The **scanf()** function reads formatted input from the standard input such as **keyboards**.

## Example 5: Integer Input/Output

```
#include <stdio.h>
int main()
{
    int testInteger;
    printf("Enter an integer: ");
    scanf("%d", &testInteger);
    printf("Number = %d", testInteger);
    return 0;
}
```

## Output

```
Enter an integer: 4
Number = 4
```

Here, we have used **%d** format specifier inside the **scanf()** function to take **int input** from the user. When the user enters an integer, it is stored in the **testInteger** variable.

**Notice**, that we have used **&testInteger** inside **scanf()**. It is because **&testInteger** gets the address of **testInteger**, and the value entered by the user is stored in that address.

## Example 6: Float and Double Input/Output

# C Programming Language

```
#include <stdio.h>
int main()
{
    float num1;
    double num2;

    printf("Enter a number: ");
    scanf("%f", &num1);
    printf("Enter another number: ");
    scanf("%lf", &num2);

    printf("num1 = %f\n", num1);
    printf("num2 = %lf", num2);

    return 0;
}
```

## Output

```
Enter a number: 12.523
Enter another number: 10.2
num1 = 12.523000
num2 = 10.200000
```

We use **%f** and **%lf** format specifier for float and double respectively.

## Example 7: C Character I/O

```
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c", &chr);
    printf("You entered %c.", chr);
    return 0;
}
```

## Output

# C Programming Language

```
Enter a character: g  
You entered g.
```

When a character is entered by the user in the above program, the character itself is not stored. Instead, an integer value (ASCII value) is stored.

And when we display that value using **%c** text format, the entered character is displayed. If we use **%d** to display the character, its ASCII value is printed.

## Example 8: ASCII Value

```
#include <stdio.h>  
int main()  
{  
    char chr;  
    printf("Enter a character: ");  
    scanf("%c", &chr);  
  
    // When %c is used, a character is displayed  
    printf("You entered %c.\n", chr);  
  
    // When %d is used, ASCII value is displayed  
    printf("ASCII value is % d.", chr);  
    return 0;  
}
```

## Output

```
Enter a character: g  
You entered g.  
ASCII value is 103.
```

## I/O Multiple Values

Here's how you can take multiple inputs from the user and display them.

# C Programming Language

```
#include <stdio.h>
int main()
{
    int a;
    float b;

    printf("Enter integer and then a float: ");

    // Taking multiple inputs
    scanf("%d%f", &a, &b);

    printf("You entered %d and %f", a, b);
    return 0;
}
```

## Output

```
Enter integer and then a float: -3
3.4
You entered -3 and 3.400000
```

## Format Specifiers for I/O

As you can see from the above examples, we use

- **%d** for int.
- **%f** for float.
- **%lf** for double.
- **%c** for char.

Here's a list of commonly used C data types and their format specifiers.

# C Programming Language

Data Type	Format Specifier
int	%d
char	%c
float	%f
double	%lf
short int	%hd
unsigned int	%u
long int	%li
long long int	%lli
unsigned long int	%lu
unsigned long long int	%llu
signed char	%c
unsigned char	%c
long double	%Lf

## • C Programming Operators

An **operator** is a symbol that operates on a value or a variable. For example: **+** is an operator to perform addition.

C has a wide range of operators to perform various operations.

## C Arithmetic Operators

An **arithmetic operator** performs mathematical operations such as addition, subtraction, multiplication, division etc. on numerical values (constants and variables).

# C Programming Language

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

## Example 1: Arithmetic Operators

```
// Working of arithmetic operators
#include <stdio.h>
int main()
{
    int a = 9,b = 4, c;

    c = a+b;
    printf("a+b = %d \n",c);
    c = a-b;
    printf("a-b = %d \n",c);
    c = a*b;
    printf("a*b = %d \n",c);
    c = a/b;
    printf("a/b = %d \n",c);
    c = a%b;
    printf("Remainder when a divided by b = %d \n",c);

    return 0;
}
```

## Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

The operators **+**, **-** and **\*** computes addition, subtraction, and multiplication respectively as you might have expected.

# C Programming Language

In normal calculation,  $9/4 = 2.25$ . However, the output is 2 in the program. It is because both the variables a and b are integers. Hence, the output is also an integer. The compiler neglects the term after the decimal point and shows answer 2 instead of 2.25.

The modulo operator **%** computes the remainder. When a=9 is divided by b=4, the remainder is 1. The **%** operator can only be used with **integers**. Suppose a = 5.0, b = 2.0, c = 5 and d = 2. Then in C programming,

```
// Either one of the operands is a floating-point number  
a/b = 2.5  
a/d = 2.5  
c/b = 2.5  
  
// Both operands are integers  
c/d = 2
```

## C Increment and Decrement Operators

C programming has two operators **increment ++** and **decrement --** to change the value of an operand (constant or variable) by 1.

**Increment ++** increases the value by 1 whereas **decrement --** decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

### Example 2: Increment and Decrement Operators

```
// Working of increment and decrement operators  
#include <stdio.h>  
int main()  
{  
    int a = 10, b = 100;  
    float c = 10.5, d = 100.5;  
  
    printf("++a = %d \n", ++a);  
    printf("--b = %d \n", --b);  
    printf("++c = %f \n", ++c);  
    printf("--d = %f \n", --d);  
  
    return 0;  
}
```

# C Programming Language

## Output

```
++a = 11
--b = 99
++c = 11.500000
++d = 99.500000
```

Here, the operators **++** and **--** are used as prefixes. These two operators can also be used as postfixes like **a++** and **a--**.

## C Assignment Operators

An **assignment operator** is used for assigning a value to a variable. The most common assignment operator is **=**.

Operator	Example	Same as
<b>=</b>	<b>a = b</b>	<b>a = b</b>
<b>+=</b>	<b>a += b</b>	<b>a = a+b</b>
<b>-=</b>	<b>a -= b</b>	<b>a = a-b</b>
<b>*=</b>	<b>a *= b</b>	<b>a = a*b</b>
<b>/=</b>	<b>a /= b</b>	<b>a = a/b</b>
<b>%=</b>	<b>a %= b</b>	<b>a = a%b</b>

## Example 3: Assignment Operators

# C Programming Language

```
// Working of assignment operators
#include <stdio.h>
int main()
{
    int a = 5, c;

    c = a;      // c is 5
    printf("c = %d\n", c);
    c += a;     // c is 10
    printf("c = %d\n", c);
    c -= a;     // c is 5
    printf("c = %d\n", c);
    c *= a;     // c is 25
    printf("c = %d\n", c);
    c /= a;     // c is 5
    printf("c = %d\n", c);
    c %= a;     // c = 0
    printf("c = %d\n", c);

    return 0;
}
```

## Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

## C Relational Operators

A **relational operator** checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in **decision making** and **loops**.

# C Programming Language

Operator	Meaning of Operator	Example
==	Equal to	<code>5 == 3</code> is evaluated to 0
>	Greater than	<code>5 &gt; 3</code> is evaluated to 1
<	Less than	<code>5 &lt; 3</code> is evaluated to 0
!=	Not equal to	<code>5 != 3</code> is evaluated to 1
>=	Greater than or equal to	<code>5 &gt;= 3</code> is evaluated to 1
<=	Less than or equal to	<code>5 &lt;= 3</code> is evaluated to 0

## Example 4: Relational Operators

```
// Working of relational operators
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;

    printf("%d == %d is %d \n", a, b, a == b);
    printf("%d == %d is %d \n", a, c, a == c);
    printf("%d > %d is %d \n", a, b, a > b);
    printf("%d > %d is %d \n", a, c, a > c);
    printf("%d < %d is %d \n", a, b, a < b);
    printf("%d < %d is %d \n", a, c, a < c);
    printf("%d != %d is %d \n", a, b, a != b);
    printf("%d != %d is %d \n", a, c, a != c);
    printf("%d >= %d is %d \n", a, b, a >= b);
    printf("%d >= %d is %d \n", a, c, a >= c);
    printf("%d <= %d is %d \n", a, b, a <= b);
    printf("%d <= %d is %d \n", a, c, a <= c);

    return 0;
}
```

# C Programming Language

## Output

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

## C Logical Operators

An expression containing **logical operator** returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in **decision making** in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression <code>((c==5) &amp;&amp; (d&gt;5))</code> equals to 0.
	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression <code>((c==5)    (d&gt;5))</code> equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression <code>!(c==5)</code> equals to 0.

## Example 5: Logical Operators

# C Programming Language

```
// Working of logical operators

#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;

    result = (a == b) && (c > b);
    printf("(a == b) && (c > b) is %d \n", result);
    result = (a == b) && (c < b);
    printf("(a == b) && (c < b) is %d \n", result);
    result = (a == b) || (c < b);
    printf("(a == b) || (c < b) is %d \n", result);
    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) is %d \n", result);
    result = !(a != b);
    printf("!(a != b) is %d \n", result);
    result = !(a == b);
    printf("!(a == b) is %d \n", result);

    return 0;
}
```

## Output

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

## Explanation of logical operator program

- $(a == b) \&\& (c > 5)$  evaluates to 1 because both operands  $(a == b)$  and  $(c > b)$  is 1 (true).
- $(a == b) \&\& (c < b)$  evaluates to 0 because operand  $(c < b)$  is 0 (false).
- $(a == b) || (c < b)$  evaluates to 1 because  $(a = b)$  is 1 (true).
- $(a != b) || (c < b)$  evaluates to 0 because both operand  $(a != b)$  and  $(c < b)$  are 0 (false).
- $!(a != b)$  evaluates to 1 because operand  $(a != b)$  is 0 (false). Hence,  $!(a != b)$  is 1 (true).
- $!(a == b)$  evaluates to 0 because  $(a == b)$  is 1 (true). Hence,  $!(a == b)$  is 0 (false).

# C Programming Language

## C Bitwise Operators

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
-	Bitwise complement
<<	Shift left
>>	Shift right

## Other Operators

### Comma Operator

Comma operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

### The sizeof operator

The **sizeof** is a unary operator that returns the size of data (constants, variables, array, structure, etc).

### Example 6: sizeof Operator

# C Programming Language

```
#include <stdio.h>
int main()
{
    int a;
    float b;
    double c;
    char d;
    printf("Size of int=%lu bytes\n",sizeof(a));
    printf("Size of float=%lu bytes\n",sizeof(b));
    printf("Size of double=%lu bytes\n",sizeof(c));
    printf("Size of char=%lu byte\n",sizeof(d));

    return 0;
}
```

## Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
```

# C Programming Language

## C FLOW CONTROL

- C if...else Statement

### C if Statement

The syntax of the if statement in C programming is:

```
if (test expression)
{
    // statements to be executed if the test expression is true
}
```

### How if statement works?

The **if statement** evaluates the test expression inside the **parenthesis ()**.

If the test expression is evaluated to true, statements inside the body of if are executed.

If the test expression is evaluated to false, statements inside the body of if are not executed.

Expression is true.

```
int test = 5;

if (test < 10)
{
    // codes
}

// codes after if
```

Expression is false.

```
int test = 5;

if (test > 10)
{
    // codes
}

// codes after if
```

### Example 1: if statement

# C Programming Language

```
// Program to display a number if it is negative

#include <stdio.h>
int main() {
    int number;

    printf("Enter an integer: ");
    scanf("%d", &number);

    // true if number is less than 0
    if (number < 0) {
        printf("You entered %d.\n", number);
    }

    printf("The if statement is easy.");
    return 0;
}
```

## Output 1

```
Enter an integer: -2
You entered -2.
The if statement is easy.
```

When the user enters **-2**, the test expression **number<0** is evaluated to true. Hence, You entered **-2** is displayed on the screen.

## Output 2

```
Enter an integer: 5
The if statement is easy.
```

When the user enters **5**, the test expression **number<0** is evaluated to false and the statement inside the body of **if** is not executed.

## C if...else Statement

The **if statement** may have an optional else block. The syntax of the **if..else** statement is:

# C Programming Language

```
if (test expression) {  
    // statements to be executed if the test expression is true  
}  
else {  
    // statements to be executed if the test expression is false  
}
```

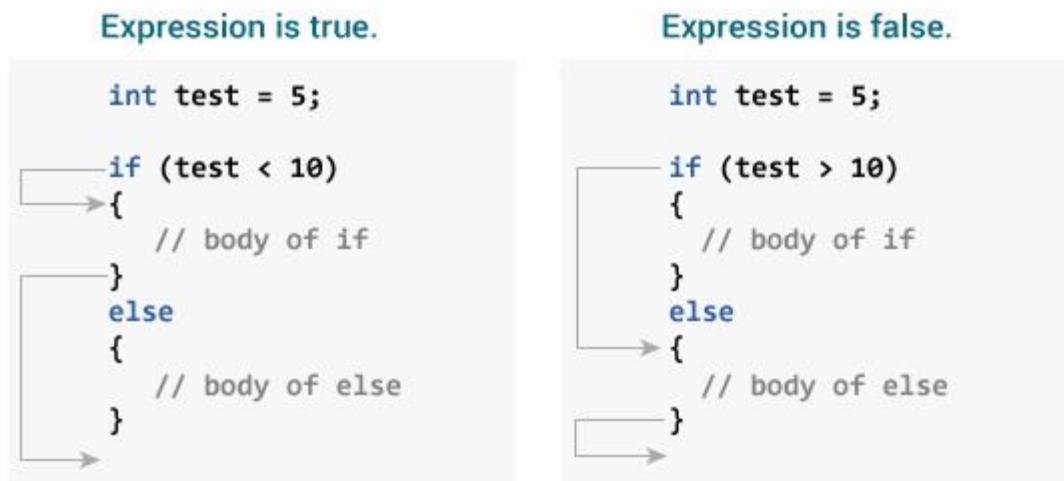
## How if...else statement works?

If the test expression is evaluated to **true**,

- statements inside the body of **if** are executed.
- statements inside the body of else are skipped from execution.

If the test expression is evaluated to false,

- statements inside the body of else are executed
- statements inside the body of if are skipped from execution.



## Example 2: if...else statement

# C Programming Language

```
// Check whether an integer is odd or even

#include <stdio.h>
int main() {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);

    // True if the remainder is 0
    if (number%2 == 0) {
        printf("%d is an even integer.", number);
    }
    else {
        printf("%d is an odd integer.", number);
    }

    return 0;
}
```

## Output

```
Enter an integer: 7
7 is an odd integer.
```

When the user enters **7**, the test expression `number%2==0` is evaluated to false. Hence, the statement inside the body of `else` is executed.

## C if...else Ladder

The **if...else** statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.

The **if...else ladder** allows you to check between multiple test expressions and execute different statements.

## Syntax of if...else Ladder

# C Programming Language

```
if (test expression1) {
    // statement(s)
}
else if(test expression2) {
    // statement(s)
}
else if (test expression3) {
    // statement(s)
}
.
.
else {
    // statement(s)
}
```

## Example 3: C if...else Ladder

```
// Program to relate two integers using =, > or < symbol

#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    //checks if the two integers are equal.
    if(number1 == number2) {
        printf("Result: %d = %d", number1, number2);
    }

    //checks if number1 is greater than number2.
    else if (number1 > number2) {
        printf("Result: %d > %d", number1, number2);
    }

    //checks if both test expressions are false
    else {
        printf("Result: %d < %d", number1, number2);
    }

    return 0;
}
```

# C Programming Language

## Output

```
Enter two integers: 12  
23  
Result: 12 < 23
```

## Nested if...else

It is possible to include an **if...else** statement inside the body of another **if...else** statement.

### Example 4: Nested if...else

This program given below relates two integers using either **<**, **>** and **=** similar to the **if...else** ladder's example. However, we will use a **nested if...else** statement to solve this problem.

```
#include <stdio.h>  
int main() {  
    int number1, number2;  
    printf("Enter two integers: ");  
    scanf("%d %d", &number1, &number2);  
  
    if (number1 >= number2) {  
        if (number1 == number2) {  
            printf("Result: %d = %d", number1, number2);  
        }  
        else {  
            printf("Result: %d > %d", number1, number2);  
        }  
    }  
    else {  
        printf("Result: %d < %d", number1, number2);  
    }  
  
    return 0;  
}
```

If the body of an if...else statement has only one statement, you do not need to use brackets **{}**.

# C Programming Language

For example, this code

```
if (a > b) {  
    print("Hello");  
}  
print("Hi");
```

is equivalent to

```
if (a > b)  
    print("Hello");  
print("Hi");
```

- **C for Loop**

In programming, a loop is used to repeat a block of code until the specified condition is met.

**C programming has three types of loops:**

1. for loop
2. while loop
3. do...while loop

## for Loop

The syntax of the for loop is:

```
for (initializationStatement; testExpression; updateStatement)  
{  
    // statements inside the body of loop  
}
```

## How for loop works?

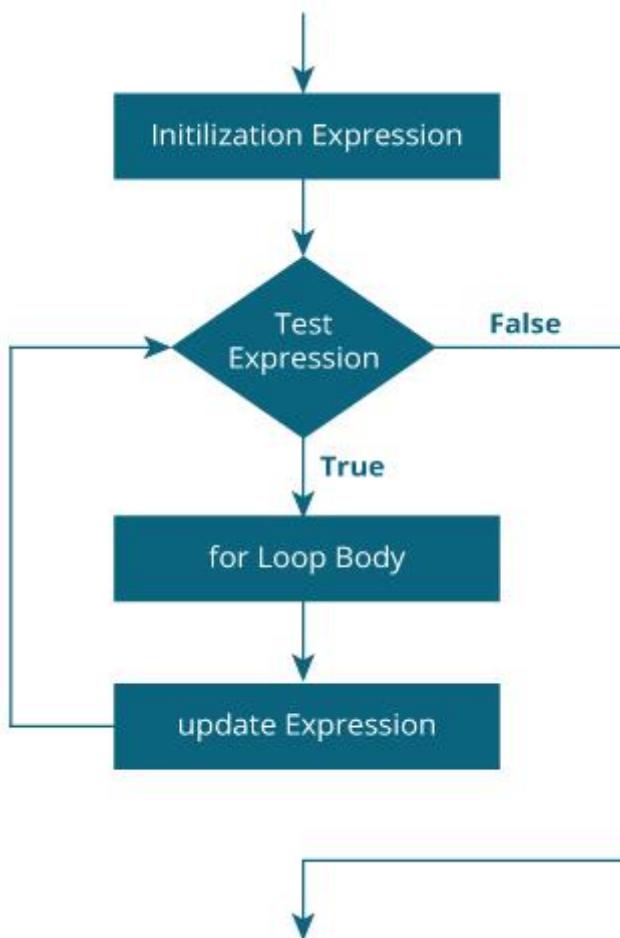
The initialization statement is executed only once.

Then, the test expression is evaluated. If the test expression is evaluated to false, the for loop is terminated.

# C Programming Language

However, if the test expression is evaluated to true, statements inside the body of for loop are executed, and the update expression is updated. Again, the test expression is evaluated. This process goes on until the test expression is false. When the test expression is false, the loop terminates.

## for loop Flowchart



## Example 1: for loop

# C Programming Language

```
// Print numbers from 1 to 10
#include <stdio.h>

int main() {
    int i;

    for (i = 1; i < 11; ++i)
    {
        printf("%d ", i);
    }
    return 0;
}
```

## Output

```
1 2 3 4 5 6 7 8 9 10
```

1. **i** is initialized to **1**.
2. The test expression **i < 11** is evaluated. Since **1** less than **11** is true, the body of for loop is executed. This will print the **1** (value of i) on the screen.
3. The update statement **++i** is executed. Now, the value of **i** will be **2**. Again, the test expression is evaluated to **true**, and the body of for loop is executed. This will print **2** (value of i) on the screen.
4. Again, the update statement **++i** is executed and the test expression **i < 11** is evaluated. This process goes on until **i** becomes **11**.
5. When **i** becomes **11**, **i < 11** will be **false**, and the for loop terminates.

## Example 2: for loop

# C Programming Language

```
// Program to calculate the sum of first n natural numbers
// Positive integers 1,2,3...n are known as natural numbers

#include <stdio.h>
int main()
{
    int num, count, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    // for loop terminates when num is less than count
    for(count = 1; count <= num; ++count)
    {
        sum += count;
    }

    printf("Sum = %d", sum);

    return 0;
}
```

## Output

```
Enter a positive integer: 10
Sum = 55
```

The value entered by the user is stored in the variable num. Suppose, the user entered **10**. The count is initialized to **1** and the test expression is evaluated. Since the test expression **count<=num** (1 less than or equal to 10) is true, the body of for loop is executed and the value of sum will equal to **1**.

Then, the update statement **++count** is executed and the count will equal to **2**. Again, the test expression is evaluated. Since **2** is also less than **10**, the test expression is evaluated to true and the body of for loop is executed. Now, the sum will equal **3**.

This process goes on and the sum is calculated until the count reaches **11**.

When the count is **11**, the test expression is evaluated to **0** (false), and the loop terminates.

# C Programming Language

Then, the value of sum is printed on the screen.

## • C while and do...while Loop

### while loop

The syntax of the while loop is:

```
while (testExpression)
{
    // statements inside the body of the loop
}
```

### How while loop works?

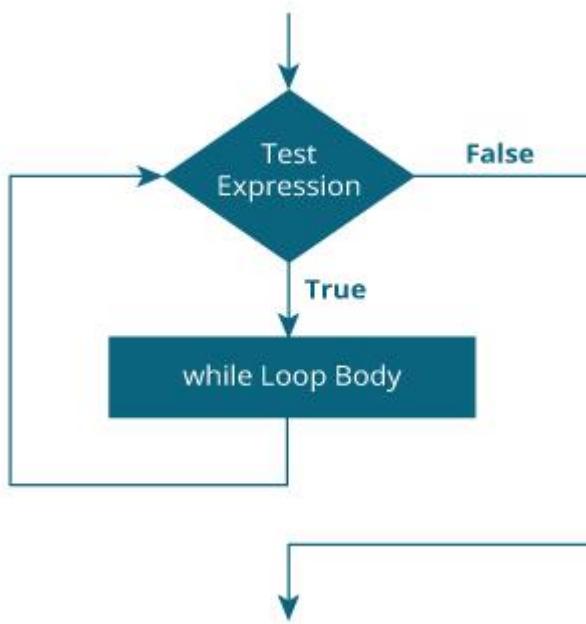
The while loop evaluates the test expression inside the parenthesis () .

If the test expression is true, statements inside the body of while loop are executed. Then, the test expression is evaluated again.

The process goes on until the test expression is evaluated to false.

If the test expression is false, the loop terminates (ends).

### Flowchart of while loop



# C Programming Language

## Example 1: while loop

```
// Print numbers from 1 to 5

#include <stdio.h>
int main()
{
    int i = 1;

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

    return 0;
}
```

## Output

```
1
2
3
4
5
```

Here, we have initialized i to 1.

1. When i is 1, the test expression  $i \leq 5$  is true. Hence, the body of the while loop is executed. This prints 1 on the screen and the value of i is increased to 2.
2. Now, i is 2, the test expression  $i \leq 5$  is again true. The body of the while loop is executed again. This prints 2 on the screen and the value of i is increased to 3.

# C Programming Language

- This process goes on until i becomes 6. When i is 6, the test expression  $i \leq 5$  will be false and the loop terminates.

## do...while loop

The **do..while** loop is similar to the while loop with one important difference. The body of do...while loop is executed at least once. Only then, the test expression is evaluated.

The syntax of the do...while loop is:

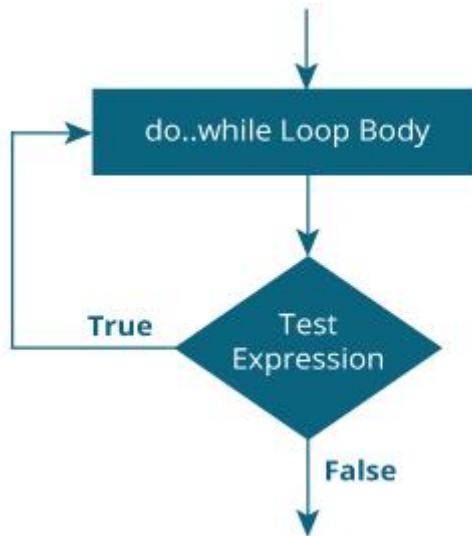
```
do
{
    // statements inside the body of the loop
}
while (testExpression);
```

## How do...while loop works?

- The body of do...while loop is executed once. Only then, the test expression is evaluated.
- If the test expression is true, the body of the loop is executed again and the test expression is evaluated.
- This process goes on until the test expression becomes false.
- If the test expression is false, the loop ends.

## Flowchart of do...while Loop

# C Programming Language



## Example 2: do...while loop

```
// Program to add numbers until the user enters zero

#include <stdio.h>
int main()
{
    double number, sum = 0;

    // the body of the loop is executed at least once
    do
    {
        printf("Enter a number: ");
        scanf("%lf", &number);
        sum += number;
    }
    while(number != 0.0);

    printf("Sum = %.2lf",sum);

    return 0;
}
```

# C Programming Language

## Output

```
Enter a number: 1.5
Enter a number: 2.4
Enter a number: -3.4
Enter a number: 4.2
Enter a number: 0
Sum = 4.70
```

- **C break and continue**

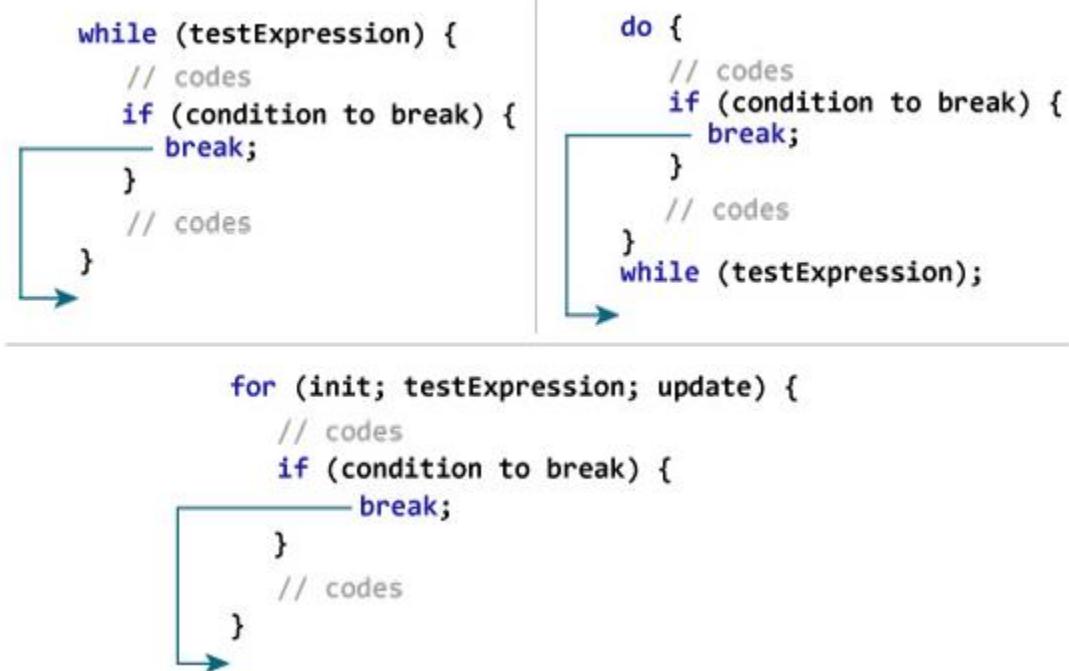
### C break

The break statement ends the loop immediately when it is encountered. Its syntax is:

```
break;
```

The break statement is almost always used with **if...else** statement inside the loop.

### How break statement works?



# C Programming Language

## Example 1: break statement

```
// Program to calculate the sum of a maximum of 10 numbers
// If a negative number is entered, the loop terminates

# include <stdio.h>
int main()
{
    int i;
    double number, sum = 0.0;

    for(i=1; i <= 10; ++i)
    {
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);

        // If the user enters a negative number, the loop ends
        if(number < 0.0)
        {
            break;
        }

        sum += number; // sum = sum + number;
    }

    printf("Sum = %.2lf",sum);

    return 0;
}
```

## Output

```
Enter a n1: 2.4
Enter a n2: 4.5
Enter a n3: 3.4
Enter a n4: -3
Sum = 10.30
```

This program calculates the sum of a maximum of 10 numbers. Why a maximum of 10 numbers? It's because if the user enters a negative number, the break statement is executed. This will end the for loop, and the sum is displayed.

# C Programming Language

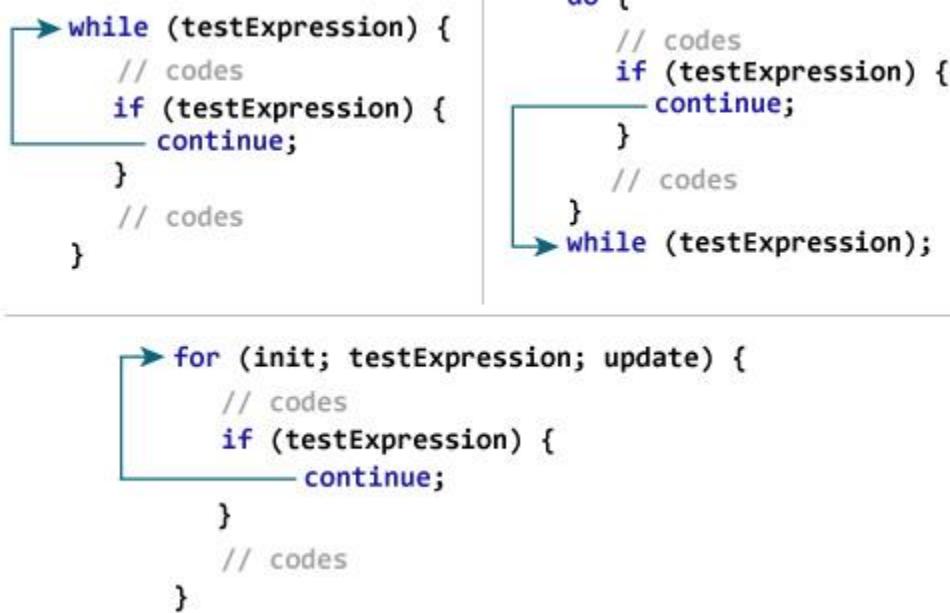
## C continue

The **continue statement** skips the current iteration of the loop and continues with the next iteration. Its syntax is:

```
continue;
```

The **continue statement** is almost always used with the **if...else** statement.

## How continue statement works?



## Example 2: continue statement

# C Programming Language

```
// Program to calculate the sum of a maximum of 10 numbers
// Negative numbers are skipped from the calculation

# include <stdio.h>
int main()
{
    int i;
    double number, sum = 0.0;

    for(i=1; i <= 10; ++i)
    {
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);

        if(number < 0.0)
        {
            continue;
        }

        sum += number; // sum = sum + number;
    }

    printf("Sum = %.2lf",sum);

    return 0;
}
```

## Output

```
Enter a n1: 1.1
Enter a n2: 2.2
Enter a n3: 5.5
Enter a n4: 4.4
Enter a n5: -3.4
Enter a n6: -45.5
Enter a n7: 34.5
Enter a n8: -4.2
Enter a n9: -1000
Enter a n10: 12
Sum = 59.70
```

# C Programming Language

In this program, when the user enters a positive number, the sum is calculated using sum **`+= number;`** statement. When the user enters a negative number, the continue statement is executed and it skips the negative number from the calculation.

## • C switch Statement

The **switch statement** allows us to execute one code block among many alternatives.

You can do the same thing with the **if...else..if** ladder. However, the syntax of the switch statement is much easier to read and write.

## Syntax of switch...case

```
switch (expression)
{
    case constant1:
        // statements
        break;

    case constant2:
        // statements
        break;

    .
    .

    default:
        // default statements
}
```

## How does the switch statement work?

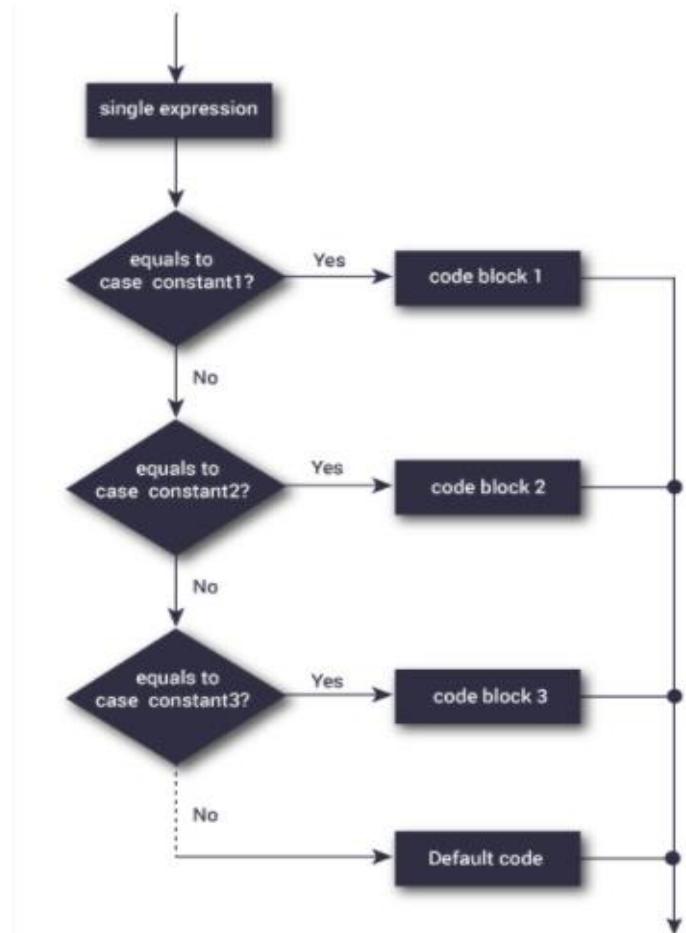
The expression is evaluated once and compared with the values of each case label.

- If there is a match, the corresponding statements after the matching label are executed.  
For example, if the value of the expression is equal to constant2, statements after case constant2: are executed until break is encountered.
- If there is no match, the default statements are executed.
- If we do not use **break**, all statements after the matching label are executed.

# C Programming Language

By the way, the **default** clause inside the switch statement is **optional**.

## switch Statement Flowchart



## Example: Simple Calculator

```
/ Program to create a simple calculator
#include <stdio.h>
int main() {
    char operator;
    double n1, n2;
    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operator);
    printf("Enter two operands: ");
    scanf("%lf %lf", &n1, &n2);

    switch(operator)
    {
```

# C Programming Language

```
case '+':  
    printf("%.1lf + %.1lf = %.1lf", n1, n2, n1+n2);  
    break;  
  
case '-':  
    printf("%.1lf - %.1lf = %.1lf", n1, n2, n1-n2);  
    break;  
  
case '*':  
    printf("%.1lf * %.1lf = %.1lf", n1, n2, n1*n2);  
    break;  
  
case '/':  
    printf("%.1lf / %.1lf = %.1lf", n1, n2, n1/n2);  
    break;  
  
// operator doesn't match any case constant +, -, *, /  
default:  
    printf("Error! operator is not correct");  
}  
return 0;  
}
```

## Output

```
Enter an operator (+, -, *, /): -  
Enter two operands: 32.5  
12.4  
32.5 - 12.4 = 20.1
```

The **-** operator entered by the user is stored in the operator variable. And, two operands **32.5** and **12.4** are stored in variables n1 and n2 respectively.

Since the operator is **-**, the control of the program jumps to

```
printf("%.1lf - %.1lf = %.1lf", n1, n2, n1-n2);
```

Finally, the **break** statement terminates the switch statement.

# C Programming Language

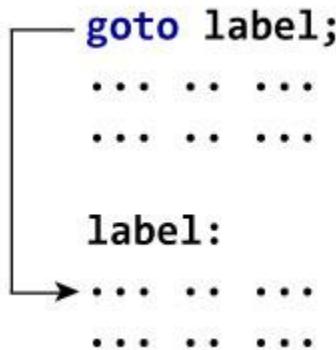
## • C goto Statement

The **goto** statement allows us to transfer control of the program to the specified label.

### Syntax of goto Statement

```
goto label;  
... ... ...  
... ... ...  
label:  
statement;
```

The label is an **identifier**. When the **goto** statement is encountered, the control of the program jumps to label: and starts executing the code.



### Example: goto Statement

```
// Program to calculate the sum and average of positive numbers  
// If the user enters a negative number, the sum and average are displayed.  
  
#include <stdio.h>  
  
int main() {  
    const int maxInput = 100;  
    int i;  
    double number, average, sum = 0.0;  
  
    for (i = 1; i <= maxInput; ++i) {  
        printf("%d. Enter a number: ", i);  
        scanf("%lf", &number);  
        if (number < 0) {  
            break;  
        }  
        sum += number;  
    }  
    if (i > 1) {  
        average = sum / i;  
        printf("The sum is %lf and the average is %lf\n", sum, average);  
    }  
}
```

# C Programming Language

```
// go to jump if the user enters a negative number
if (number < 0.0) {
    goto jump;
}
sum += number;
}

jump:
average = sum / (i - 1);
printf("Sum = %.2f\n", sum);
printf("Average = %.2f", average);

return 0;
}
```

## Output

```
1. Enter a number: 3
2. Enter a number: 4.3
3. Enter a number: 9.3
4. Enter a number: -2.9
Sum = 16.60
Average = 5.53
```

## Reasons to avoid goto

The use of **goto** statement may lead to code that is buggy and hard to follow. For example,

```
one:
for (i = 0; i < number; ++i)
{
    test += i;
    goto two;
}
two:
if (test > 5) {
    goto three;
}
.......
```

Also, the **goto** statement allows you to do **bad stuff** such as jump out of the scope.

# C Programming Language

That being said, **goto** can be useful sometimes. For example: to break from **nested loops**.

## Should you use goto?

If you think the use of **goto** statement simplifies your program, you can use it. That being said, **goto** is rarely useful and you can create any **C** program without using **goto** altogether.

# C Programming Language

## C FUNCTIONS

A **function** is a block of code that performs a specific task.

Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:

create a circle function

create a color function

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Types of function

There are two types of function in C programming:

- **Standard library functions.**
- **User-defined functions.**

### Standard library functions

The **standard library functions** are **built-in functions** in C programming.

These functions are defined in **header files**. For example,

The **printf()** is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the **stdio.h** header file. Hence, to use the **printf()** function, we need to include the **stdio.h** header file using **#include <stdio.h>**.

The **sqrt()** function calculates the square root of a number. The function is defined in the **math.h** header file.

### User-defined function

You can also create functions as per your need. Such functions created by the user are known as **user-defined functions**.

# C Programming Language

## How user-defined function works?

```
#include <stdio.h>
void functionName()
{
    .... ....
    .... ....
}

int main()
{
    .... ....
    .... ....

    functionName();

    .... ....
    .... ....
}
```

The execution of a C program begins from the **main()** function.

When the compiler encounters **functionName();**, control of the program jumps to

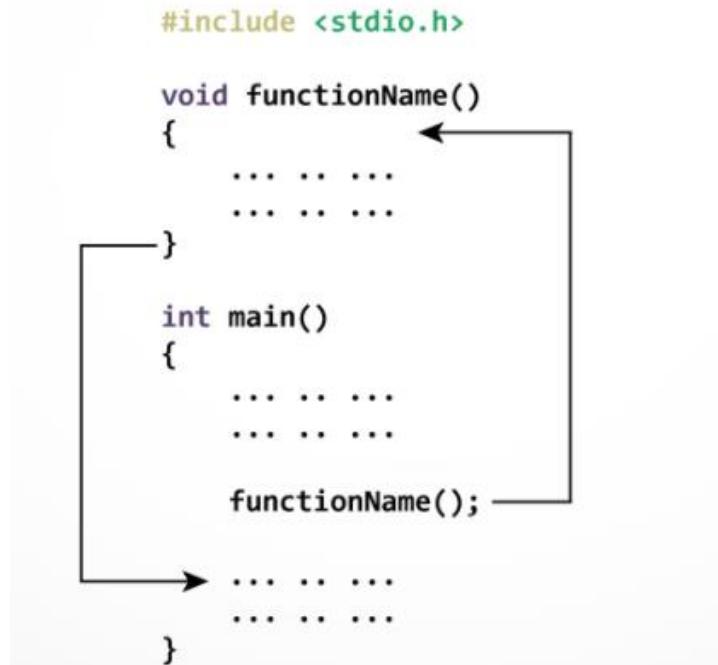
```
void functionName()
```

And, the compiler starts executing the codes inside **functionName()**.

The control of the program jumps back to the **main()** function once code inside the function definition is executed.

# C Programming Language

## How function works in C programming?



**Note**, function names are identifiers and should be unique.

### Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

### • C User-defined functions

A **function** is a block of code that performs a specific task.

C allows you to define functions according to your need. These functions are known as **user-defined functions**. For example:

Suppose, you need to create a circle and color it depending upon the radius and color.

You can create two functions to solve this problem:

# C Programming Language

- createCircle() function
- color() function

## Example: User-defined function

Here is an example to add two integers. To perform this task, we have created an user-defined **addNumbers()**.

```
#include <stdio.h>
int addNumbers(int a, int b);           // function prototype

int main()
{
    int n1,n2,sum;

    printf("Enter 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
}
```

## 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.

# C Programming Language

## Syntax of function prototype

```
returnType functionName(type1 argument1, type2 argument2, ...);
```

In the above example, **int addNumbers(int a, int b);** is the function prototype which provides the following information to the compiler:

1. **name** of the function is **addNumbers()**.
2. **return type** of the function is **int**.
3. two **arguments** of type int are passed to the function.

The function prototype is not needed if the user-defined function is defined before the **main()** function.

## Calling a function

Control of the program is transferred to the **user-defined function** by calling it.

## Syntax of function call

In the above example, the function call is made using **addNumbers(n1, n2);** statement inside the **main()** function.

## Function definition

**Function definition** contains the block of code to perform a specific task. In our example, adding two numbers and returning it.

## Syntax of function definition

```
returnType functionName(type1 argument1, type2 argument2, ...)
{
    //body of the function
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

# C Programming Language

## Passing arguments to a function

In programming, **argument** refers to the variable passed to the function. In the above example, two variables **n1** and **n2** are passed during the function call.

The **parameters** **a** and **b** accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

### How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ...
    ...
    sum = addNumbers(n1, n2);
    ...
}

int addNumbers(int a, int b)
{
    ...
    ...
}
```

The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.

If **n1** is of char type, **a** also should be of char type. If **n2** is of float type, variable **b** also should be of float type.

- + A function can also be called without passing an argument.

## Return Statement

The **return statement** terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

# C Programming Language

In the above example, the value of the result variable is returned to the main function. The sum variable in the **main()** function is assigned this value.

## Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ...
    sum = addNumbers(n1, n2);
    ...
}

int addNumbers(int a, int b)
{
    ...
    return result;
}
```

sum = result

## Syntax of return statement

```
return (expression);
```

## For example,

```
return a;
return (a+b);
```

The type of value returned from the function and the return type specified in the function prototype and function definition must match.

# C Programming Language

## • Types of User-defined Functions in C Programming

These 4 programs below check whether the integer entered by the user is a prime number or not.

The output of all these programs below is the same, and we have created a **user-defined function** in each example. However, the approach we have taken in each example is different.

### Example 1: No arguments passed and no return value

```
#include <stdio.h>
void checkPrimeNumber();

int main()
{
    checkPrimeNumber();      // argument is not passed
    return 0;
}

// return type is void meaning doesn't return any value
void checkPrimeNumber()
{
    int n, i, flag = 0;
    printf("Enter a positive integer: ");
    scanf("%d",&n);

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
        {
            flag = 1;
        }
    }
    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
        printf("%d is a prime number.", n);
}
```

# C Programming Language

The **checkPrimeNumber()** function takes input from the user, checks whether it is a prime number or not and displays it on the screen.

The empty parentheses in **checkPrimeNumber();** statement inside the **main()** function indicates that **no argument** is passed to the function.

The return type of the function is **void**. Hence, no value is returned from the function.

## Example 2: No arguments passed but a return value

```
#include <stdio.h>
int getInteger();
int main()
{
    int n, i, flag = 0;
    // no argument is passed
    n = getInteger();

    for(i=2; i<=n/2; ++i)
    {
        if(n%i==0){
            flag = 1;
            break;
        }
    }
    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
        printf("%d is a prime number.", n);
    return 0;
}
// returns integer entered by the user
int getInteger()
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d",&n);

    return n;
}
```

# C Programming Language

The empty parentheses in the `n = getInteger();` statement indicates that no argument is passed to the function. And, the value returned from the function is assigned to `n`.

Here, the `getInteger()` function takes input from the user and returns it. The code to check whether a number is prime or not is inside the `main()` function.

## Example 3: Argument passed but no return value

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);
int main()
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n is passed to the function
    checkPrimeAndDisplay(n);

    return 0;
}
// return type is void meaning doesn't return any value
void checkPrimeAndDisplay(int n)
{
    int i, flag = 0;
    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0){
            flag = 1;
            break;
        }
    }
    if(flag == 1)
        printf("%d is not a prime number.",n);
    else
        printf("%d is a prime number.", n);
}
```

# C Programming Language

The integer value entered by the user is passed to the **checkPrimeAndDisplay()** function. Here, the **checkPrimeAndDisplay()** function checks whether the argument passed is a prime number or not and displays the appropriate message.

## Example 4: Argument passed and a return value

```
#include <stdio.h>
int checkPrimeNumber(int n);

int main()
{
    int n, flag;
    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n is passed to the checkPrimeNumber() function
    // the returned value is assigned to the flag variable
    flag = checkPrimeNumber(n);
    if(flag == 1)
        printf("%d is not a prime number",n);
    else
        printf("%d is a prime number",n);

    return 0;
}

// int is returned from the function
int checkPrimeNumber(int n)
{
    int i;
    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
            return 1;
    }
    return 0;
}
```

The input from the user is passed to the **checkPrimeNumber()** function.

The **checkPrimeNumber()** function checks whether the passed argument is prime or not.

# C Programming Language

If the **passed argument** is a **prime number**, the function returns 0. If the passed argument is a non-prime number, the function returns 1. The return value is assigned to the flag variable.

Depending on whether flag is **0** or **1**, an appropriate message is printed from the **main()** function.

## C Recursion

A function that calls itself is known as a **recursive function**. And, this technique is known as **recursion**.

### How recursion works?

```
void recurse()
{
    ...
    recurse();
    ...
}

int main()
{
    ...
    recurse();
    ...
}
```

#### How does recursion work?

```
void recurse() ←
{
    ...
    recurse(); ————— recursive call
    ...
}

int main()
{
    ...
    recurse(); —————
    ...
}
```

# C Programming Language

The **recursion** continues until some condition is met to prevent it.

To prevent infinite recursion, **if...else statement** (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

## Example: Sum of Natural Numbers Using Recursion

```
#include <stdio.h>
int sum(int n);

int main() {
    int number, result;

    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);

    printf("sum = %d", result);
    return 0;
}

int sum(int n) {
    if (n != 0)
        // sum() function calls itself
        return n + sum(n-1);
    else
        return n;
}
```

## Output

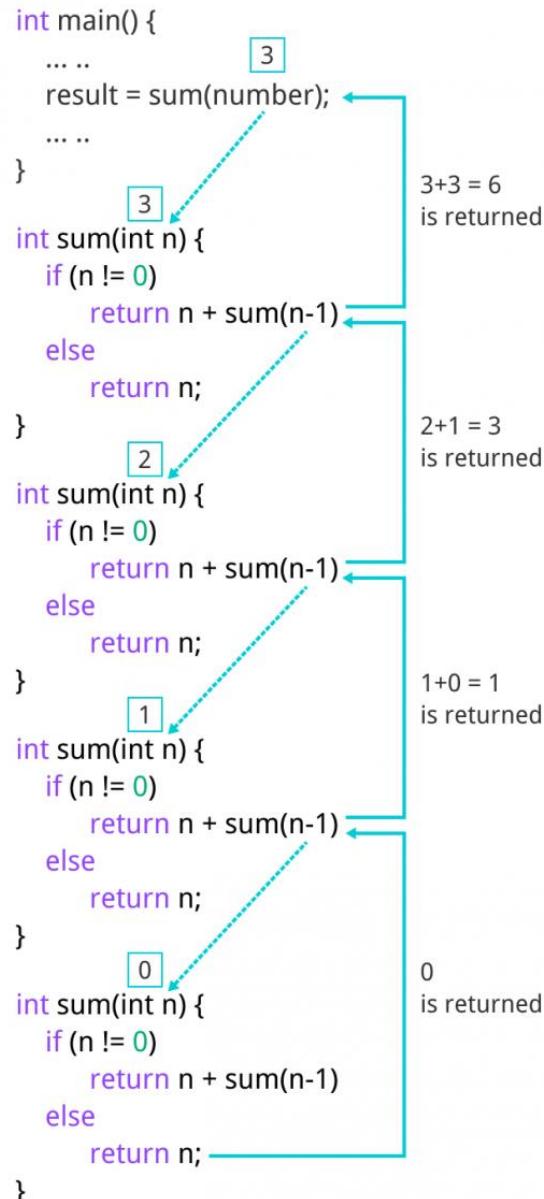
```
Enter a positive integer:3
sum = 6
```

Initially, the **sum()** is called from the **main()** function with number passed as an argument.

Suppose, the value of n inside **sum()** is 3 initially. During the next function call, 2 is passed to the **sum()** function. This process continues until n is equal to 0.

# C Programming Language

When **n** is equal to 0, the if condition fails and the else part is executed returning the sum of integers ultimately to the **main()** function.



## Advantages and Disadvantages of Recursion

**Recursion** makes program elegant. However, if performance is vital, use loops instead as recursion is usually much slower.

# C Programming Language

That being said, recursion is an important concept. It is frequently used in data structure and algorithms. For example, it is common to use recursion in problems such as tree traversal.

## • C Storage Class

Every variable in C programming has two properties: type and storage class.

Type refers to the data type of a variable. And, storage class determines the scope, visibility and lifetime of a variable.

There are 4 types of storage class:

- automatic
- external
- static
- register

## Local Variable

The variables declared inside a block are automatic or local variables. The local variables exist only inside the block in which it is declared.

Let's take an example.

```
#include <stdio.h>

int main(void) {

    for (int i = 0; i < 5; ++i) {
        printf("C programming");
    }

    // Error: i is not declared at this point
    printf("%d", i);
    return 0;
}
```

# C Programming Language

When you run the above program, you will get an error undeclared identifier i. It's because i is declared inside the for loop block. Outside of the block, it's undeclared.

Let's take another example.

```
int main() {
    int n1; // n1 is a local variable to main()
}

void func() {
    int n2; // n2 is a local variable to func()
}
```

In the above example, n1 is local to main() and n2 is local to func().

This means you cannot access the n1 variable inside func() as it only exists inside main().

Similarly, you cannot access the n2 variable inside main() as it only exists inside func().

## Global Variable

Variables that are declared outside of all functions are known as external or global variables. They are accessible from any function inside the program.

### Example 1: Global Variable

```
#include <stdio.h>
void display();

int n = 5; // global variable
int main()
{
    ++n;
    display();
    return 0;
}
void display()
{
    ++n;
    printf("n = %d", n);
}
```

# C Programming Language

## Output

```
n = 7
```

Suppose, a global variable is declared in file1. If you try to use that variable in a different file file2, the compiler will complain. To solve this problem, keyword extern is used in file2 to indicate that the external variable is declared in another file.

## Register Variable

The register keyword is used to declare register variables. Register variables were supposed to be faster than local variables.

However, modern compilers are very good at code optimization, and there is a rare chance that using register variables will make your program faster.

Unless you are working on embedded systems where you know how to optimize code for the given application, there is no use of register variables.

## Static Variable

A **static variable** is declared by using the static keyword. For example;

```
static int i;
```

The value of a static variable persists until the end of the program.

## Example 2: Static Variable

```
#include <stdio.h>
void display();
int main()
{
    display();
    display();
}
void display()
{
```

# C Programming Language

```
static int c = 1;  
c += 5;  
printf("%d ",c);  
}
```

## Output

```
6 11
```

During the first function call, the value of c is initialized to 1. Its value is increased by 5.

Now, the value of c is 6, which is printed on the screen.

During the second function call, c is not initialized to 1 again. It's because c is a static variable. The value c is increased by 5. Now, its value will be 11, which is printed on the screen.

# C Programming Language

## C PROGRAMMING ARRAYS

- ### C Arrays

An array is a variable that can store multiple values. For example, if you want to store 100 integers, you can create an array for it.

```
int data[100];
```

### How to declare an array?

```
dataType arrayName[arraySize];
```

For example,

```
float mark[5];
```

Here, we declared an array, mark, of **floating-point type**. And its size is 5. Meaning, it can hold 5 floating-point values.

It's important to note that the size and type of an array cannot be changed once it is declared.

### Access Array Elements

You can access elements of an array by indices.

Suppose you declared an array mark as above. The first element is **mark[0]**, the second element is **mark[1]** and so on.

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]

# C Programming Language

## Few keynotes:

- **Arrays** have 0 as the first index, not 1. In this example, **mark[0]** is the first element.
- If the size of an array is **n**, to access the last element, the **n-1** index is used. In this example, **mark[4]**
- Suppose the starting address of **mark[0]** is **2120d**. Then, the address of the **mark[1]** will be **2124d**. Similarly, the address of **mark[2]** will be **2128d** and so on. This is because the size of a **float** is **4 bytes**.

## How to initialize an array?

It is possible to initialize an array during declaration. For example,

```
int mark[5] = {19, 10, 8, 17, 9};
```

You can also initialize an array like this.

```
int mark[] = {19, 10, 8, 17, 9};
```

Here, we haven't specified the size. However, the compiler knows its size is 5 as we are initializing it with 5 elements.

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
19	10	8	17	9

Here,

```
mark[0] is equal to 19  
mark[1] is equal to 10  
mark[2] is equal to 8  
mark[3] is equal to 17
```

# C Programming Language

mark[4] is equal to 9

## Change Value of Array elements

```
int mark[5] = {19, 10, 8, 17, 9}

// make the value of the third element to -1
mark[2] = -1;

// make the value of the fifth element to 0
mark[4] = 0;
```

## Input and Output Array Elements

Here's how you can take input from the user and store it in an array element.

```
// take input and store it in the 3rd element
scanf("%d", &mark[2]);

// take input and store it in the ith element
scanf("%d", &mark[i-1]);
```

Here's how you can print an individual element of an array.

```
// print the first element of the array
printf("%d", mark[0]);

// print the third element of the array
printf("%d", mark[2]);

// print ith element of the array
printf("%d", mark[i-1]);
```

## Example 1: Array Input/Output

```
// Program to take 5 values from the user and store them in an array
// Print the elements stored in the array
#include <stdio.h>
```

# C Programming Language

```
int main() {
    int values[5];

    printf("Enter 5 integers: ");

    // taking input and storing it in an array
    for(int i = 0; i < 5; ++i) {
        scanf("%d", &values[i]);
    }

    printf("Displaying integers: ");

    // printing elements of an array
    for(int i = 0; i < 5; ++i) {
        printf("%d\n", values[i]);
    }
    return 0;
}
```

## Output

```
Enter 5 integers: 1
-3
34
0
3
Displaying integers: 1
-3
34
0
3
```

Here, we have used a for loop to take 5 inputs from the user and store them in an array.

Then, using another for loop, these elements are displayed on the screen.

## Example 2: Calculate Average

```
// Program to find the average of n numbers using arrays
```

# C Programming Language

```
#include <stdio.h>
int main()
{
    int marks[10], i, n, sum = 0, average;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    for(i=0; i<n; ++i)
    {
        printf("Enter number%d: ", i+1);
        scanf("%d", &marks[i]);

        // adding integers entered by the user to the sum variable
        sum += marks[i];
    }

    average = sum/n;
    printf("Average = %d", average);

    return 0;
}
```

## Output

```
Enter n: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39
```

Here, we have computed the average of **n** numbers entered by the user.

## Access elements out of its bound!

Suppose you declared an array of 10 elements. Let's say,

# C Programming Language

```
int testArray[10];
```

You can access the array elements from **testArray[0]** to **testArray[9]**.

Now let's say if you try to access **testArray[12]**. The element is not available. This may cause unexpected output (undefined behavior). Sometimes you might get an error and some other time your program may run correctly. Hence, you should never access elements of an array outside of its bound.

- **C Multidimensional Arrays**

In C programming, you can create an array of arrays. These arrays are known as multidimensional arrays. For example,

```
float x[3][4];
```

Here, **x** is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as a table with 3 rows and each row has 4 columns.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Similarly, you can declare a **three-dimensional (3d) array**. For example,

```
float y[2][4][3];
```

Here, the array **y** can hold 24 elements.

## Initializing a multidimensional array

# C Programming Language

Here is how you can initialize two-dimensional and three-dimensional arrays:

## Initialization of a 2d array

```
// Different ways to initialize two-dimensional array

int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};
```

## Initialization of a 3d array

You can initialize a **three-dimensional array** in a similar way like a **two-dimensional array**. Here's an example,

```
int test[2][3][4] = {
    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

## Example 1: Two-dimensional array to store and print values

```
// C program to store temperature of two cities of a week and display it.

#include <stdio.h>
const int CITY = 2;
const int WEEK = 7;
int main()
{
    int temperature[CITY][WEEK];

    // Using nested loop to store values in a 2d array
    for (int i = 0; i < CITY; ++i)
    {
        for (int j = 0; j < WEEK; ++j)
        {
            printf("City %d, Day %d: ", i + 1, j + 1);
            scanf("%d", &temperature[i][j]);
        }
    }
}
```

# C Programming Language

```
    }
}

printf("\nDisplaying values: \n\n");

// Using nested loop to display values of a 2d array
for (int i = 0; i < CITY; ++i)
{
    for (int j = 0; j < WEEK; ++j)
    {
        printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
    }
}
return 0;
}
```

## Output

```
City 1, Day 1: 33
City 1, Day 2: 34
City 1, Day 3: 35
City 1, Day 4: 33
City 1, Day 5: 32
City 1, Day 6: 31
City 1, Day 7: 30
City 2, Day 1: 23
City 2, Day 2: 22
City 2, Day 3: 21
City 2, Day 4: 24
City 2, Day 5: 22
City 2, Day 6: 25
City 2, Day 7: 26
```

Displaying values:

```
City 1, Day 1 = 33
City 1, Day 2 = 34
City 1, Day 3 = 35
City 1, Day 4 = 33
City 1, Day 5 = 32
City 1, Day 6 = 31
City 1, Day 7 = 30
```

# C Programming Language

```
City 2, Day 1 = 23
City 2, Day 2 = 22
City 2, Day 3 = 21
City 2, Day 4 = 24
City 2, Day 5 = 22
City 2, Day 6 = 25
City 2, Day 7 = 26
```

## Example 2: Sum of two matrices

```
// C program to find the sum of two matrices of order 2*2

#include <stdio.h>
int main()
{
    float a[2][2], b[2][2], result[2][2];

    // Taking input using nested for loop
    printf("Enter elements of 1st matrix\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
    {
        printf("Enter a%d%d: ", i + 1, j + 1);
        scanf("%f", &a[i][j]);
    }

    // Taking input using nested for loop
    printf("Enter elements of 2nd matrix\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
    {
        printf("Enter b%d%d: ", i + 1, j + 1);
        scanf("%f", &b[i][j]);
    }

    // adding corresponding elements of two arrays
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
    {
        result[i][j] = a[i][j] + b[i][j];
    }
```

# C Programming Language

```
// Displaying the sum
printf("\nSum Of Matrix:");

for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
{
    printf("%.1f\t", result[i][j]);

    if (j == 1)
        printf("\n");
}
return 0;
}
```

## Output

```
Enter elements of 1st matrix
Enter a11: 2;
Enter a12: 0.5;
Enter a21: -1.1;
Enter a22: 2;
Enter elements of 2nd matrix
Enter b11: 0.2;
Enter b12: 0;
Enter b21: 0.23;
Enter b22: 23;

Sum Of Matrix:
2.2      0.5
-0.9     25.0
```

## Example 3: Three-dimensional array

```
// C Program to store and print 12 values entered by the user

#include <stdio.h>
int main()
{
    int test[2][3][2];
```

# C Programming Language

```
printf("Enter 12 values: \n");

for (int i = 0; i < 2; ++i)
{
    for (int j = 0; j < 3; ++j)
    {
        for (int k = 0; k < 2; ++k)
        {
            scanf("%d", &test[i][j][k]);
        }
    }
}

// Printing values with proper index.

printf("\nDisplaying values:\n");
for (int i = 0; i < 2; ++i)
{
    for (int j = 0; j < 3; ++j)
    {
        for (int k = 0; k < 2; ++k)
        {
            printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
        }
    }
}

return 0;
}
```

## Output

```
Enter 12 values:
1
2
3
4
5
6
7
```

# C Programming Language

```
8
9
10
11
12

Displaying Values:
test[0][0][0] = 1
test[0][0][1] = 2
test[0][1][0] = 3
test[0][1][1] = 4
test[0][2][0] = 5
test[0][2][1] = 6
test[1][0][0] = 7
test[1][0][1] = 8
test[1][1][0] = 9
test[1][1][1] = 10
test[1][2][0] = 11
test[1][2][1] = 12
```

- Pass arrays to a function in C

## Passing individual array elements

Passing array elements to a function is similar to **passing variables to a function**.

### Example 1: Passing an array

```
#include <stdio.h>
void display(int age1, int age2)
{
    printf("%d\n", age1);
    printf("%d\n", age2);
}

int main()
{
    int ageArray[] = {2, 8, 4, 12};
```

# C Programming Language

```
// Passing second and third elements to display()
display(ageArray[1], ageArray[2]);
return 0;
}
```

## Output

```
8
4
```

## Example 2: Passing arrays to functions

```
// Program to calculate the sum of array elements by passing to a function

#include <stdio.h>
float calculateSum(float age[]);

int main() {
    float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18};

    // age array is passed to calculateSum()
    result = calculateSum(age);
    printf("Result = %.2f", result);
    return 0;
}

float calculateSum(float age[]) {

    float sum = 0.0;

    for (int i = 0; i < 6; ++i) {
        sum += age[i];
    }

    return sum;
}
```

## Output

# C Programming Language

```
Result = 162.50
```

To pass an entire array to a function, only the name of the array is passed as an argument.

```
result = calculateSum(age);
```

However, notice the use of **[]** in the function definition.

```
float calculateSum(float age[]) {  
    ...  
}
```

This informs the compiler that you are passing a one-dimensional array to the function.

## Passing Multidimensional Arrays to a Function

To pass multidimensional arrays to a function, only the name of the array is passed to the function(similar to one-dimensional arrays).

### Example 3: Passing two-dimensional arrays

```
#include <stdio.h>  
void displayNumbers(int num[2][2]);  
int main()  
{  
    int num[2][2];  
    printf("Enter 4 numbers:\n");  
    for (int i = 0; i < 2; ++i)  
        for (int j = 0; j < 2; ++j)  
            scanf("%d", &num[i][j]);  
  
    // passing multi-dimensional array to a function  
    displayNumbers(num);  
    return 0;  
}  
  
void displayNumbers(int num[2][2])  
{  
    printf("Displaying:\n");
```

# C Programming Language

```
for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 2; ++j) {
        printf("%d\n", num[i][j]);
    }
}
```

## Output

Enter 4 numbers:

2  
3  
4  
5

Displaying:

2  
3  
4  
5

**Note:** In C programming, you can pass arrays to functions, however, you cannot return arrays from functions.

# C Programming Language

## C PROGRAMMING POINTERS

### C Pointers

Pointers are powerful features of C and C++ programming. Before we learn pointers, let's learn about addresses in C programming.

### Address in C

If you have a variable **var** in your program, **&var** will give you its address in the memory. We have used address numerous times while using the **scanf()** function.

```
scanf("%d", &var);
```

Here, the value entered by the user is stored in the address of var variable. Let's take a working example.

```
#include <stdio.h>
int main()
{
    int var = 5;
    printf("var: %d\n", var);

    // Notice the use of & before var
    printf("address of var: %p", &var);
    return 0;
```

# C Programming Language

```
}
```

## Output

```
var: 5  
address of var: 2686778
```

**Note:** You will probably get a different address when you run the above code.

## C Pointers

Pointers (pointer variables) are special variables that are used to **store addresses rather than values**.

### Pointer Syntax

Here is how we can declare pointers.

```
int* p;
```

Here, we have declared a pointer **p** of **int** type. You can also declare pointers in these ways.

```
int *p1;  
int * p2;
```

Let's take another example of declaring pointers.

```
int* p1, p2;
```

Here, we have declared a pointer p1 and a normal variable p2.

### Assigning addresses to Pointers

Let's take an example.

# C Programming Language

```
int* pc, c;  
c = 5;  
pc = &c;
```

Here, **5** is assigned to the **c** variable. And, the address of **c** is assigned to the **pc** pointer.

## Get Value of Thing Pointed by Pointers

To get the value of the thing pointed by the pointers, we use the **\*** operator. For example:

```
int* pc, c;  
c = 5;  
pc = &c;  
printf("%d", *pc); // Output: 5
```

Here, the address of **c** is assigned to the **pc** pointer. To get the value stored in that address, we used **\*pc**.

**Note:** In the above example, **pc** is a pointer, not **\*pc**. You cannot and should not do something like **\*pc = &c;**

By the way, **\*** is called the **dereference operator** (when working with pointers). It operates on a pointer and gives the value stored in that pointer.

## Changing Value Pointed by Pointers

Let's take an example.

```
int* pc, c;  
c = 5;  
pc = &c;  
c = 1;  
printf("%d", c); // Output: 1  
printf("%d", *pc); // Output: 1
```

We have assigned the address of **c** to the **pc** pointer.

Then, we changed the value of **c** to **1**. Since **pc** and the address of **c** is the same, **\*pc** gives us **1**.

# C Programming Language

Let's take another example.

```
int* pc, c;
c = 5;
pc = &c;
*pc = 1;
printf("%d", *pc); // Output: 1
printf("%d", c); // Output: 1
```

We have assigned the address of **c** to the **pc** pointer.

Then, we changed **\*pc** to **1** using **\*pc = 1;**. Since **pc** and the address of **c** is the same, **c** will be equal to **1**.

Let's take one more example.

```
int* pc, c, d;
c = 5;
d = -15;

pc = &c; printf("%d", *pc); // Output: 5
pc = &d; printf("%d", *pc); // Output: -15
```

Initially, the address of **c** is assigned to the **pc** pointer using **pc = &c;**. Since **c** is **5**, **\*pc** gives us **5**.

Then, the address of **d** is assigned to the **pc** pointer using **pc = &d;**. Since **d** is **-15**, **\*pc** gives us **-15**.

## Example: Working of Pointers

Let's take a working example.

```
#include <stdio.h>
int main()
{
    int* pc, c;
    c = 22;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c); // 22
```

# C Programming Language

```
pc = &c;
printf("Address of pointer pc: %p\n", pc);
printf("Content of pointer pc: %d\n\n", *pc); // 22

c = 11;
printf("Address of pointer pc: %p\n", pc);
printf("Content of pointer pc: %d\n\n", *pc); // 11

*pc = 2;
printf("Address of c: %p\n", &c);
printf("Value of c: %d\n\n", c); // 2
return 0;
}
```

## Output

```
Address of c: 2686784
Value of c: 22
```

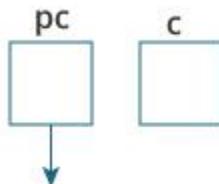
```
Address of pointer pc: 2686784
Content of pointer pc: 22
```

```
Address of pointer pc: 2686784
Content of pointer pc: 11
```

```
Address of c: 2686784
Value of c: 2
```

## Explanation of the program

1. int\* pc, c;

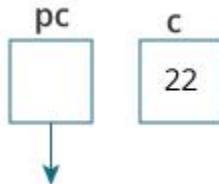


Here, a pointer **pc** and a normal variable **c**, both of type int, is created.

# C Programming Language

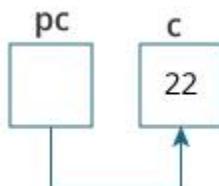
Since **pc** and **c** are not initialized at initially, pointer **pc** points to either no address or a random address. And, variable **c** has an address but contains random garbage value.

2. **c = 22;**



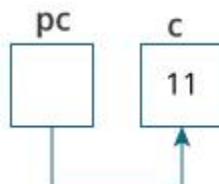
This assigns **22** to the variable **c**. That is, **22** is stored in the memory location of variable **c**.

3. **pc = &c;**



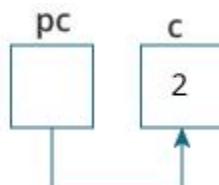
This assigns the address of variable **c** to the pointer **pc**.

4. **c = 11;**



This assigns 11 to variable **c**.

5. **\*pc = 2;**



This change the value at the memory location pointed by the pointer **pc** to 2.

## Common mistakes when working with pointers

# C Programming Language

Suppose, you want pointer pc to point to the address of c. Then,

```
int c, *pc;

// pc is address but c is not
pc = c; // Error

// &c is address but *pc is not
*pc = &c; // Error

// both &c and pc are addresses
pc = &c;

// both c and *pc values
*pc = c;
```

Here's an example of pointer syntax beginners often find confusing.

```
#include <stdio.h>
int main() {
    int c = 5;
    int *p = &c;

    printf("%d", *p); // 5
    return 0;
}
```

**Why didn't we get an error when using int \*p = &c;?**

It's because

```
int *p = &c;
```

is equivalent to

```
int *p:
p = &c;
```

In both cases, we are creating a pointer p (not \*p) and assigning &c to it.

To avoid this confusion, we can use the statement like this:

# C Programming Language

```
int* p = &c;
```

## • Relationship Between Arrays and Pointers

An **array** is a block of sequential data. Let's write a program to print addresses of array elements.

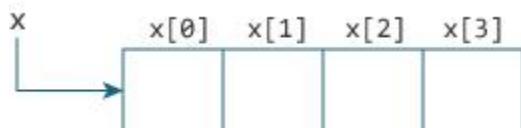
```
#include <stdio.h>
int main() {
    int x[4];
    int i;
    for(i = 0; i < 4; ++i) {
        printf("&x[%d] = %p\n", i, &x[i]);
    }
    printf("Address of array x: %p", x);
    return 0;
}
```

## Output

```
&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448
```

There is a difference of 4 bytes between two consecutive elements of array **x**. It is because the size of **int** is **4 bytes** (on our compiler).

**Notice** that, the address of **&x[0]** and **x** is the same. It's because the variable name **x** points to the first element of the array.



From the above example, it is clear that **&x[0]** is equivalent to **x**. And, **x[0]** is equivalent to **\*x**.

# C Programming Language

Similarly,

**&x[1]** is equivalent to **x+1** and **x[1]** is equivalent to **\*(x+1)**.

**&x[2]** is equivalent to **x+2** and **x[2]** is equivalent to **\*(x+2)**.

...

Basically, **&x[i]** is equivalent to **x+i** and **x[i]** is equivalent to **\*(x+i)**.

## Example 1: Pointers and Arrays

```
#include <stdio.h>
int main() {
    int i, x[6], sum = 0;
    printf("Enter 6 numbers: ");
    for(i = 0; i < 6; ++i) {
        // Equivalent to scanf("%d", &x[i]);
        scanf("%d", x+i);

        // Equivalent to sum += x[i]
        sum += *(x+i);
    }
    printf("Sum = %d", sum);
    return 0;
}
```

When you run the program, the output will be:

```
Enter 6 numbers: 2
3
4
4
12
4
Sum = 29
```

Here, we have declared an array **x** of **6** elements. To access elements of the array, we have used pointers.

# C Programming Language

In most contexts, array names decay to pointers. In simple words, array names are converted to pointers. That's the reason why you can use pointers to access elements of arrays. However, you should remember that pointers and arrays are not the same.

## Example 2: Arrays and Pointers

```
#include <stdio.h>
int main() {
    int x[5] = {1, 2, 3, 4, 5};
    int* ptr;

    // ptr is assigned the address of the third element
    ptr = &x[2];

    printf("*ptr = %d \n", *ptr);    // 3
    printf("*(ptr+1) = %d \n", *(ptr+1)); // 4
    printf("*(ptr-1) = %d", *(ptr-1)); // 2

    return 0;
}
```

When you run the program, the output will be:

```
*ptr = 3
*(ptr+1) = 4
*(ptr-1) = 2
```

In this example, **&x[2]**, the address of the third element, is assigned to the **ptr** pointer. Hence, **3** was displayed when we printed **\*ptr**.

And, printing **\*(ptr+1)** gives us the fourth element. Similarly, printing **\*(ptr-1)** gives us the second element.

## • C Call by Reference: Using pointers

In C programming, it is also possible to **pass addresses** as **arguments** to functions.

To accept these addresses in the function definition, we can use pointers. It's because pointers are used to store addresses. Let's take an example:

# C Programming Language

## Example: Call by reference

```
#include <stdio.h>
void swap(int *n1, int *n2);

int main()
{
    int num1 = 5, num2 = 10;

    // address of num1 and num2 is passed
    swap( &num1, &num2);

    printf("num1 = %d\n", num1);
    printf("num2 = %d", num2);
    return 0;
}

void swap(int* n1, int* n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

When you run the program, the output will be:

```
num1 = 10
num2 = 5
```

The address of **num1** and **num2** are passed to the **swap()** function using **swap(&num1, &num2);**.

Pointers **n1** and **n2** accept these arguments in the function definition.

```
void swap(int* n1, int* n2) {
    ...
}
```

# C Programming Language

When `*n1` and `*n2` are changed inside the `swap()` function, `num1` and `num2` inside the `main()` function are also changed.

Inside the `swap()` function, `*n1` and `*n2` swapped. Hence, `num1` and `num2` are also swapped.

**Notice** that, `swap()` is not returning anything; its return type is `void`.

This technique is known as call by reference in C programming.

## Example 2: Passing Pointers to Functions

```
#include <stdio.h>

void addOne(int* ptr) {
    (*ptr)++; // adding 1 to *ptr
}

int main()
{
    int* p, i = 10;
    p = &i;
    addOne(p);

    printf("%d", *p); // 11
    return 0;
}
```

Here, the value stored at `p`, `*p`, is 10 initially.

We then passed the pointer `p` to the `addOne()` function. The `ptr` pointer gets this address in the `addOne()` function.

Inside the function, we increased the value stored at `ptr` by 1 using `(*ptr)++;`. Since `ptr` and `p` pointers both have the same address, `*p` inside `main()` is also 11.

## • C Dynamic Memory Allocation

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

# C Programming Language

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as **dynamic memory** allocation in C programming.

To allocate memory dynamically, library functions are **malloc()**, **calloc()**, **realloc()** and **free()** are used. These functions are defined in the **<stdlib.h>** header file.

## C malloc()

The name "**malloc**" stands for memory allocation.

The **malloc()** function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.

### Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

### Example

```
ptr = (float*) malloc(100 * sizeof(float));
```

The above statement allocates **400 bytes** of memory. It's because the size of float is **4 bytes**. And, the pointer **ptr** holds the address of the first byte in the allocated memory.

The expression results in a **NULL** pointer if the memory cannot be allocated.

## C calloc()

The name "**calloc**" stands for contiguous allocation.

The **malloc()** function allocates memory and leaves the memory uninitialized. Whereas, the **calloc()** function allocates memory and initializes all bits to zero.

### Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

### Example:

# C Programming Language

```
ptr = (float*) calloc(25, sizeof(float));
```

The above statement allocates contiguous space in memory for 25 elements of type float.

## C free()

Dynamically allocated memory created with either **calloc()** or **malloc()** doesn't get freed on their own. You must explicitly use **free()** to release the space.

### Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by **ptr**.

## Example 1: malloc() and free()

```
// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) malloc(n * sizeof(int));

    // if memory cannot be allocated
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i)
    {
```

# C Programming Language

```
    scanf("%d", ptr + i);
    sum += *(ptr + i);
}

printf("Sum = %d", sum);

// deallocated the memory
free(ptr);

return 0;
}
```

Here, we have dynamically allocated the memory for **n** number of **int**.

## Example 2: calloc() and free()

```
// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }
}
```

# C Programming Language

```
printf("Sum = %d", sum);
free(ptr);
return 0;
}
```

## C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the **realloc()** function.

### Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, **ptr** is reallocated with a new size **x**.

### Example 3: realloc()

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

int main()
{
    int *ptr, i, n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Addresses of previously allocated memory: ");
    for(i = 0; i < n1; ++i)
        printf("%u\n", ptr + i);

    printf("\nEnter the new size: ");
    scanf("%d", &n2);

    // Relocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("Addresses of newly allocated memory: ");
```

# C Programming Language

```
for(i = 0; i < n2; ++i)
    printf("%u\n", ptr + i);

free(ptr);

return 0;
}
```

When you run the program, the output will be:

```
Enter size: 2
Addresses of previously allocated memory:26855472
26855476

Enter the new size: 4
Addresses of newly allocated memory:26855472
26855476
26855480
26855484
```

## C PROGRAMMING STRINGS

In **C programming**, a string is a sequence of characters terminated with a null character **\0**. For example:

```
char c[] = "c string";
```

When the compiler encounters a sequence of characters enclosed in the **double quotation** marks, it appends a null character **\0** at the end by default.

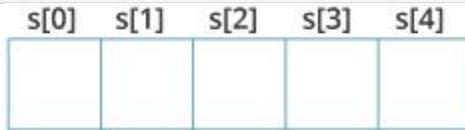
c		s	t	r	i	n	g	\0
---	--	---	---	---	---	---	---	----

## How to declare a string?

Here's how you can declare strings:

# C Programming Language

```
char s[5];
```

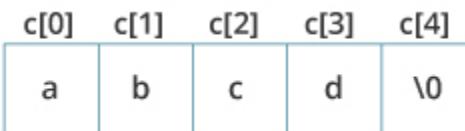


Here, we have declared a string of 5 characters.

## How to initialize strings?

You can initialize strings in a number of ways.

```
char c[] = "abcd";  
  
char c[50] = "abcd";  
  
char c[] = {'a', 'b', 'c', 'd', '\0'};  
  
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```



Let's take another example:

```
char c[5] = "abcde";
```

Here, we are trying to assign **6** characters (the last character is '**\0**') to a char array having 5 characters. This is bad and you should never do this.

## Assigning Values to Strings

Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For example,

```
char c[100];  
c = "C programming"; // Error! array type is not assignable.
```

**Note:** Use the strcpy() function to copy the string instead.

# C Programming Language

## Read String from the user

You can use the **scanf()** function to read a string.

The **scanf()** function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).

### Example 1: scanf() to read a string

```
#include <stdio.h>
int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```

### Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Even though **Dennis Ritchie** was entered in the above program, only "**Ritchie**" was stored in the name string. It's because there was a space after **Dennis**.

## How to read a line of text?

You can use the **fgets()** function to read a line of string. And, you can use **puts()** to display the string.

### Example 2: fgets() and puts()

```
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin); // read string
```

# C Programming Language

```
printf("Name: ");
puts(name);    // display string
return 0;
}
```

## Output

```
Enter name: Tom Hanks
Name: Tom Hanks
```

Here, we have used **fgets()** function to read a string from the user.

**fgets(name, sizeof(name), stdin);** // read string

The **sizeof(name)** results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the name string.

To print the string, we have used **puts(name);**.

**Note:** The **gets()** function can also be used to take input from the user. However, it is removed from the C standard.

It's because **gets()** allows you to input any length of characters. Hence, there might be a buffer overflow.

## Passing Strings to Functions

Strings can be passed to a function in a similar way as arrays.

### Example 3: Passing string to a Function

```
#include <stdio.h>
void displayString(char str[]);

int main()
{
    char str[50];
    printf("Enter string: ");
    fgets(str, sizeof(str), stdin);
    displayString(str);      // Passing string to a function.
    return 0;
}
```

# C Programming Language

```
}

void displayString(char str[])
{
    printf("String Output: ");
    puts(str);
}
```

## Strings and Pointers

Similar like arrays, string names are "**decayed**" to pointers. Hence, you can use pointers to manipulate elements of the string. We recommended you to check C Arrays and Pointers before you check this example.

### Example 4: Strings and Pointers

```
#include <stdio.h>

int main(void) {
    char name[] = "Harry Potter";

    printf("%c", *name);      // Output: H
    printf("%c", *(name+1));  // Output: a
    printf("%c", *(name+7));  // Output: o

    char *namePtr;

    namePtr = name;
    printf("%c", *namePtr);   // Output: H
    printf("%c", *(namePtr+1)); // Output: a
    printf("%c", *(namePtr+7)); // Output: o
}
```

## Commonly Used String Functions

**strlen()** - calculates the length of a string

**strcpy()** - copies a string to another

**strcmp()** - compares two strings

**strcat()** - concatenates two strings

# C Programming Language

## C **strlen()**

The **strlen()** function calculates the length of a given string.

The **strlen()** function takes a string as an argument and returns its length. The returned value is of type long **int**.

It is defined in the **<string.h>** header file.

### Example: C **strlen()** function

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[20] = "Program";
    char b[20] = {'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};

    printf("Length of string a = %ld \n", strlen(a));
    printf("Length of string b = %ld \n", strlen(b));

    return 0;
}
```

### Output

```
Length of string a = 7
Length of string b = 7
```

**Note** that the **strlen()** function doesn't count the null character **\0** while calculating the length.

## C **strcpy()**

The function prototype of **strcpy()** is:

```
char* strcpy(char* destination, const char* source);
```

- The **strcpy()** function copies the string pointed by source (including the null character) to the destination.

# C Programming Language

- The **strcpy()** function also returns the copied string.  
The **strcpy()** function is defined in the **string.h** header file.

## Example: C strcpy()

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[20] = "C programming";
    char str2[20];
    // copying str1 to str2
    strcpy(str2, str1);
    puts(str2); // C programming
    return 0;
}
```

## Output

```
C programming
```

**Note:** When you use **strcpy()**, the size of the destination string should be large enough to store the copied string. Otherwise, it may result in undefined behavior.

## C strcmp()

The **strcmp()** function compares two strings and returns **0** if both strings are identical.

### C strcmp() Prototype

```
int strcmp (const char* str1, const char* str2);
```

The **strcmp()** function takes two strings and returns an integer.

The **strcmp()** compares two strings character by character.

# C Programming Language

If the first character of two strings is equal, the next character of two strings are compared. This continues until the corresponding characters of two strings are different or a null character '\0' is reached.

It is defined in the **string.h** header file.

## Return Value from strcmp()

RETURN VALUE	REMARKS
<b>0</b>	if both strings are identical (equal)
<b>NEGATIVE</b>	if the ASCII value of the first unmatched character is less than second.
<b>POSITIVE INTEGER</b>	if the ASCII value of the first unmatched character is greater than second.

## Example: C strcmp() function

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

int main()
{
    char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";
    int result;

    // comparing strings str1 and str2
    result = strcmp(str1, str2);
    printf("strcmp(str1, str2) = %d\n", result);

    // comparing strings str1 and str3
    result = strcmp(str1, str3);
    printf("strcmp(str1, str3) = %d\n", result);

    return 0;
}
```

## Output

```
strcmp(str1, str2) = 32
```

# C Programming Language

```
strcmp(str1, str3) = 0
```

The first unmatched character between string **str1** and **str2** is third character. The **ASCII** value of '**c**' is 99 and the ASCII value of '**C**' is 67. Hence, when strings **str1** and **str2** are compared, the return value is 32.

When strings **str1** and **str3** are compared, the result is **0** because both strings are identical.

## C **strcat()**

In C programming, the **strcat()** function concatenates (joins) two strings.

The function definition of **strcat()** is:

```
char *strcat(char *destination, const char *source)
```

It is defined in the **string.h** header file.

### **strcat()** arguments

As you can see, the **strcat()** function takes two arguments:

**destination** - destination string

**source** - source string

The **strcat()** function concatenates the **destination** string and the **source** string, and the result is stored in the **destination** string.

### Example: C **strcat()** function

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[100] = "This is ", str2[] = "programiz.com";

    // concatenates str1 and str2
    // the resultant string is stored in str1.
    strcat(str1, str2);
```

# C Programming Language

```
    puts(str1);
    puts(str2);

    return 0;
}
```

## Output

```
This is programiz.com

programiz.com
```

**Note:** When we use **strcat()**, the size of the destination string should be large enough to store the resultant string. If not, we will get the segmentation fault error.

## STRUCT AND UNION

In **C programming**, a **struct** (or structure) is a collection of **variables** (can be of different types) under a single name.

### How to define structures?

Before you can create **structure variables**, you need to define its **data type**. To define a struct, the **struct** keyword is used.

### Syntax of struct

```
struct structureName
{
```

# C Programming Language

```
dataType member1;  
dataType member2;  
...  
};
```

Here is an example:

```
struct Person  
{  
    char name[50];  
    int citNo;  
    float salary;  
};
```

Here, a derived type **struct Person** is defined. Now, you can create variables of this type.

## Create struct variables

When a **struct** type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create **variables**.

Here's how we create structure variables:

```
struct Person  
{  
    char name[50];  
    int citNo;  
    float salary;  
};  
  
int main()  
{  
    struct Person person1, person2, p[20];  
    return 0;  
}
```

# C Programming Language

Another way of creating a struct variable is:

```
struct Person
{
    char name[50];
    int citNo;
    float salary;
} person1, person2, p[20];
```

In both cases, two variables person1, person2, and an array variable p having 20 elements of type struct Person are created.

## Access members of a structure

There are two types of operators used for accessing members of a structure.

. - Member operator

-> - Structure pointer operator

Suppose, you want to access the **salary** of **person2**. Here's how you can do it.

```
person2.salary
```

## Example: Add two distances

```
// Program to add two distances (feet-inch)
#include <stdio.h>
struct Distance
{
    int feet;
    float inch;
} dist1, dist2, sum;

int main()
{
    printf("1st distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);
```

# C Programming Language

```
printf("Enter inch: ");
scanf("%f", &dist1.inch);
printf("2nd distance\n");

printf("Enter feet: ");
scanf("%d", &dist2.feet);

printf("Enter inch: ");
scanf("%f", &dist2.inch);

// adding feet
sum.feet = dist1.feet + dist2.feet;
// adding inches
sum.inch = dist1.inch + dist2.inch;

// changing to feet if inch is greater than 12
while (sum.inch >= 12)
{
    ++sum.feet;
    sum.inch = sum.inch - 12;
}

printf("Sum of distances = %d'-%.1f\"", sum.feet, sum.inch);
return 0;
}
```

## Output

```
1st distance
Enter feet: 12
Enter inch: 7.9
2nd distance
Enter feet: 2
Enter inch: 9.8
Sum of distances = 15'-5.7"
```

## Keyword **typedef**

# C Programming Language

We use the **typedef** keyword to create an alias name for data types. It is commonly used with structures to simplify the syntax of declaring variables.

This code

```
struct Distance{  
    int feet;  
    float inch;  
};  
  
int main() {  
    structure Distance d1, d2;  
}
```

is equivalent to

```
typedef struct Distance{  
    int feet;  
    float inch;  
} distances;  
  
int main() {  
    distances d1, d2;  
}
```

## Nested Structures

You can create structures within a structure in C programming. For example,

```
struct complex  
{  
    int imag;  
    float real;  
};  
  
struct number
```

# C Programming Language

```
{  
    struct complex comp;  
    int integers;  
} num1, num2;
```

Suppose, you want to set imag of num2 variable to 11. Here's how you can do it:

```
num2.comp.imag = 11;
```

## Why structs in C?

Suppose, you want to store information about a person: his/her name, citizenship number, and salary. You can create different variables **name**, **citNo** and **salary** to store this information.

What if you need to store information of more than one person? Now, you need to create different variables for each information per person: **name1**, **citNo1**, **salary1**, **name2**, **citNo2**, **salary2**,etc.

A better approach would be to have a collection of all related information under a single name Person structure and use it for every person.

- **C structs and Pointers**

## C Pointers to struct

Here's how you can create pointers to structs.

```
struct name {  
    member1;  
    member2;  
    .  
    .
```

# C Programming Language

```
};

int main()
{
    struct name *ptr, Harry;
}
```

Here, **ptr** is a pointer to **struct**.

## Example: Access members using Pointer

To access members of a structure using pointers, we use the **->** operator.

```
#include <stdio.h>
struct person
{
    int age;
    float weight;
};

int main()
{
    struct person *personPtr, person1;
    personPtr = &person1;

    printf("Enter age: ");
    scanf("%d", &personPtr->age);

    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);

    printf("Displaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);

    return 0;
}
```

In this example, the address of **person1** is stored in the **personPtr** pointer using **personPtr = &person1;**.

# C Programming Language

Now, you can access the members of person1 using the personPtr pointer.

By the way,

- `personPtr->age` is equivalent to `(*personPtr).age`
- `personPtr->weight` is equivalent to `(*personPtr).weight`

## Dynamic memory allocation of structs

Before you proceed this section, we recommend you to check C dynamic memory allocation.

Sometimes, the number of struct variables you declared may be insufficient. You may need to allocate memory during run-time. Here's how you can achieve this in C programming.

### Example: Dynamic memory allocation of structs

```
#include <stdio.h>
#include <stdlib.h>
struct person {
    int age;
    float weight;
    char name[30];
};

int main()
{
    struct person *ptr;
    int i, n;

    printf("Enter the number of persons: ");
    scanf("%d", &n);

    // allocating memory for n numbers of struct person
    ptr = (struct person*) malloc(n * sizeof(struct person));

    for(i = 0; i < n; ++i)
    {
        printf("Enter first name and age respectively: ");
    }
}
```

# C Programming Language

```
// To access members of 1st struct person,  
// ptr->name and ptr->age is used  
  
// To access members of 2nd struct person,  
// (ptr+1)->name and (ptr+1)->age is used  
scanf("%s %d", (ptr+i)->name, &(ptr+i)->age);  
}  
  
printf("Displaying Information:\n");  
for(i = 0; i < n; ++i)  
    printf("Name: %s\tAge: %d\n", (ptr+i)->name, (ptr+i)->age);  
  
return 0;  
}
```

When you run the program, the output will be:

```
Enter the number of persons: 2  
Enter first name and age respectively: Harry 24  
Enter first name and age respectively: Gary 32  
Displaying Information:  
Name: Harry      Age: 24  
Name: Gary       Age: 32
```

In the above example, **n number** of struct variables are created where **n** is entered by the user.

To allocate the memory for n number of struct person, we used,

```
ptr = (struct person*) malloc(n * sizeof(struct person));
```

Then, we used the **ptr** pointer to access elements of **person**.

## • C Structure and Function

Similar to variables of built-in types, you can also pass structure variables to a function.

### Passing structs to functions

Here's how you can pass structures to a function

# C Programming Language

```
#include <stdio.h>
struct student
{
    char name[50];
    int age;
};

// function prototype
void display(struct student s);

int main()
{
    struct student s1;

    printf("Enter name: ");
    scanf("%[^\\n]*c", s1.name);

    printf("Enter age: ");
    scanf("%d", &s1.age);

    display(s1); // passing struct as an argument

    return 0;
}
void display(struct student s)
{
    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nAge: %d", s.age);
}
```

## Output

```
Enter name: Bond
Enter age: 13

Displaying information
Name: Bond
```

# C Programming Language

Age: 13

Here, a struct variable **s1** of type **struct student** is created. The variable is passed to the **display()** function using **display(s1);** statement.

## Return struct from a function

Here's how you can return structure from a function:

```
#include <stdio.h>
struct student
{
    char name[50];
    int age;
};

// function prototype
struct student getInformation();

int main()
{
    struct student s;

    s = getInformation();

    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nRoll: %d", s.age);

    return 0;
}
struct student getInformation()
{
    struct student s1;

    printf("Enter name: ");
    scanf ("%[^\\n]*c", s1.name);

    printf("Enter age: ");
    scanf("%d", &s1.age);
```

# C Programming Language

```
    return s1;
}
```

Here, the **getInformation()** function is called using **s = getInformation();** statement. The function returns a structure of type struct student. The returned structure is displayed from the **main()** function.

Notice that, the return type of **getInformation()** is also struct student.

## Passing struct by reference

You can also pass structs by reference (in a similar way like you pass variables of built-in type by reference). We suggest you to read pass by reference tutorial before you proceed.

During pass by reference, the memory addresses of struct variables are passed to the function.

```
#include <stdio.h>
typedef struct Complex
{
    float real;
    float imag;
} complex;

void addNumbers(complex c1, complex c2, complex *result);

int main()
{
    complex c1, c2, result;

    printf("For first number, \n");
    printf("Enter real part: ");
    scanf("%f", &c1.real);
    printf("Enter imaginary part: ");
    scanf("%f", &c1.imag);

    printf("For second number, \n");
    printf("Enter real part: ");
```

# C Programming Language

```
scanf("%f", &c2.real);
printf("Enter imaginary part: ");
scanf("%f", &c2.imag);

addNumbers(c1, c2, &result);
printf("\nresult.real = %.1f\n", result.real);
printf("result.imag = %.1f", result.imag);

return 0;
}

void addNumbers(complex c1, complex c2, complex *result)
{
    result->real = c1.real + c2.real;
    result->imag = c1.imag + c2.imag;
}
```

## Output

```
For first number,
Enter real part: 1.1
Enter imaginary part: -2.4
For second number,
Enter real part: 3.4
Enter imaginary part: -3.2

result.real = 4.5
result.imag = -5.6
```

In the above program, three structure variables **c1**, **c2** and the address of result is passed to the **addNumbers()** function. Here, result is passed by reference.

When the result variable inside the **addNumbers()** is altered, the result variable inside the **main()** function is also altered accordingly.

- **C Unions**

## How to define a union?

# C Programming Language

We use the **union** keyword to define unions. Here's an example:

```
union car
{
    char name[50];
    int price;
};
```

The above code defines a derived type **union car**.

## Create union variables

When a **union** is defined, it creates a **user-defined** type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables. Here's how we create **union** variables.

```
union car
{
    char name[50];
    int price;
};

int main()
{
    union car car1, car2, *car3;
    return 0;
}
```

Another way of creating union variables is:

```
union car
{
    char name[50];
    int price;
} car1, car2, *car3;
```

In both cases, union variables **car1**, **car2**, and a union pointer **car3** of union car type are created.

## Access members of a union

# C Programming Language

We use the `.` operator to access members of a union. To access pointer variables, we use also use the `->` operator.

In the above example,

- To access price for `car1`, `car1.price` is used.
- To access price using `car3`, either `(*car3).price` or `car3->price` can be used.

## Difference between unions and structures

Let's take an example to demonstrate the difference between unions and structures:

```
#include <stdio.h>
union unionJob
{
    //defining a union
    char name[32];
    float salary;
    int workerNo;
} uJob;

struct structJob
{
    char name[32];
    float salary;
    int workerNo;
} sJob;

int main()
{
    printf("size of union = %d bytes", sizeof(uJob));
    printf("\nsize of structure = %d bytes", sizeof(sJob));
    return 0;
}
```

## Output

```
size of union = 32
```

# C Programming Language

```
size of structure = 40
```

## Why this difference in the size of union and structure variables?

Here, the size of **sJob** is 40 bytes because

- the size of **name[32]** is 32 bytes
- the size of **salary** is 4 bytes
- the size of **workerNo** is 4 bytes

However, the size of **uJob** is 32 bytes. It's because the size of a union variable will always be the size of its largest element. In the above example, the size of its largest element, **(name[32])**, is 32 bytes.

## Only one union member can be accessed at a time

You can access all members of a structure at once as sufficient memory is allocated for all members. However, it's not the case in unions. You can only access a single member of a union at one time. Let's see an example.

```
#include <stdio.h>
union Job
{
    float salary;
    int workerNo;
} j;
int main()
{
    j.salary = 12.3;
    j.workerNo = 100;
    printf("Salary = %.1f\n", j.salary);
    printf("Number of workers = %d", j.workerNo);
    return 0;
}
```

## Output

```
Salary = 0.0
```

# C Programming Language

Number of workers = 100

Notice that **12.3** was not stored in **j.salary**.

## C FILE HANDLING

---

# C Programming Language

A **file** is a container in computer storage devices used for storing data.

Why files are needed?

When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.

If you have to enter a large number of data, it will take a lot of time to enter them all.

However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.

You can easily move your data from one computer to another without any changes.

Types of Files

When dealing with files, there are two types of files you should know about:

## 1. Text files

### 2. Binary files

#### 1. Text files

Text files are the normal **.txt** files. You can easily create text files using any simple text editors such as Notepad.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

#### 2. Binary files

Binary files are mostly the **.bin** files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold a higher amount of data, are not readable easily, and provides better security than text files.

## File Operations

In C, you can perform four major operations on files, either text or binary:

# C Programming Language

3. Creating a new file
4. Opening an existing file
5. Closing a file
6. Reading from and writing information to a file

## Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

```
FILE *fptr;
```

### Opening a file - for creation and edit

Opening a file is performed using the fopen() function defined in the stdio.h header file. The syntax for opening a file in standard I/O is:

```
ptr = fopen("fileopen", "mode");
```

For example,

```
fopen("E:\\cprogram\\newprogram.txt", "w");  
fopen("E:\\cprogram\\oldprogram.bin", "rb");
```

Let's suppose the file newprogram.txt doesn't exist in the location E:\\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode 'w'.

The writing mode allows you to create and edit (overwrite) the contents of the file.

Now let's suppose the second binary file oldprogram.bin exists in the location E:\\cprogram. The second function opens the existing file for reading in binary mode 'rb'. The reading mode only allows you to read the file, you cannot write into the file.

#### OPENING MODES IN STANDARD I/O

MODE | Meaning of Mode

During Inexistence of file

# C Programming Language

R	Open for reading.	If the file does not exist, <code>fopen()</code> returns NULL.
RB	Open for reading in binary mode.	If the file does not exist, <code>fopen()</code> returns NULL.
W	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
WB	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
A	Open for append. Data is added to the end of the file.	If the file does not exist, it will be created.
AB	Open for append in binary mode. Data is added to the end of the file.	If the file does not exist, it will be created.
R+	Open for both reading and writing.	If the file does not exist, <code>fopen()</code> returns NULL.
RB+	Open for both reading and writing in binary mode.	If the file does not exist, <code>fopen()</code> returns NULL.
W+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
WB+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
A+	Open for both reading and appending.	If the file does not exist, it will be created.
AB+	Open for both reading and appending in binary mode.	If the file does not exist, it will be created.

## Closing a File

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using the `fclose()` function.

# C Programming Language

```
fclose(fptr);
```

Here, **fptr** is a file pointer associated with the file to be closed.

## Reading and writing to a text file

For reading and writing to a text file, we use the functions **fprintf()** and **fscanf()**.

They are just the file versions of **printf()** and **scanf()**. The only difference is that **fprint()** and **fscanf()** expects a pointer to the structure FILE.

### Example 1: Write to a text file

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

int main()
{
    int num;
    FILE *fptr;

    // use appropriate location if you are using MacOS or Linux
    fptr = fopen("C:\\program.txt", "w");

    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }
    printf("Enter num: ");
    scanf("%d", &num);

    fprintf(fptr, "%d", num);
    fclose(fptr);

    return 0;
}
```

This program takes a number from the user and stores in the file program.txt.

# C Programming Language

After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

## Example 2: Read from a text file

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

int main()
{
    int num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.txt","r")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    fscanf(fptr,"%d", &num);

    printf("Value of n=%d", num);
    fclose(fptr);

    return 0;
}
```

This program reads the integer present in the program.txt file and prints it onto the screen. If you successfully created the file from Example 1, running this program will get you the integer you entered.

Other functions like **fgetchar()**, **fputc()** etc. can be used in a similar way.

## Reading and writing to a binary file

Functions **fread()** and **fwrite()** are used for reading from and writing to a file on the disk respectively in case of binary files.

## Writing to a binary file

# C Programming Language

To write into a binary file, you need to use the **fwrite()** function. The functions take four arguments:

- 1- address of data to be written in the disk
- 2- size of data to be written in the disk
- 3- number of such type of data
- 4- pointer to the file where you want to write.

```
5- fwrite(addressData, sizeData, numbersData, pointerToFile);
```

## Example 3: Write to a binary file using fwrite()

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

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    for(n = 1; n < 5; ++n)
    {
        num.n1 = n;
        num.n2 = 5*n;
        num.n3 = 5*n + 1;
        fwrite(&num, sizeof(struct threeNum), 1, fptr);
    }
}
```

# C Programming Language

```
}

fclose(fptr);

return 0;
}
```

In this program, we create a new file program.bin in the C drive.

We declare a structure threeNum with three numbers - n1, n2 and n3, and define it in the main function as num.

Now, inside the for loop, we store the value into the file using fwrite().

The first parameter takes the address of num and the second parameter takes the size of the structure threeNum.

Since we're only inserting one instance of num, the third parameter is 1. And, the last parameter \*fptr points to the file we're storing the data.

Finally, we close the file.

## Reading from a binary file

Function **fread()** also take 4 arguments similar to the **fwrite()** function as above.

```
fread(addressData, sizeData, numbersData, pointerToFile);
```

## Example 4: Read from a binary file using fread()

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

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
```

# C Programming Language

```
FILE *fptr;

if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
    printf("Error! opening file");

    // Program exits if the file pointer returns NULL.
    exit(1);
}

for(n = 1; n < 5; ++n)
{
    fread(&num, sizeof(struct threeNum), 1, fptr);
    printf("n1: %d\\tn2: %d\\tn3: %d", num.n1, num.n2, num.n3);
}
fclose(fptr);

return 0;
}
```

In this program, you read the same file **program.bin** and loop through the records one by one.

In simple terms, you read one **threeNum** record of **threeNum** size from the file pointed by **\*fptr** into the structure **num**.

## Getting data using **fseek()**

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using **fseek()**.

As the name suggests, **fseek()** seeks the cursor to the given record in the file.

## Syntax of **fseek()**

```
fseek(FILE * stream, long int offset, int whence);
```

# C Programming Language

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

## DIFFERENT WHENCE IN FSEEK()

WHENCE	Meaning
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

## Example 5: fseek()

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

struct threeNum
{
    int n1, n2, n3;
};

int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    // Moves the cursor to the end of the file
    fseek(fptr, -sizeof(struct threeNum), SEEK_END);

    for(n = 1; n < 5; ++n)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\nn2: %d\nn3: %d\n", num.n1, num.n2, num.n3);
        fseek(fptr, -2*sizeof(struct threeNum), SEEK_CUR);
    }
}
```

# C Programming Language

```
}

fclose(fptr);

return 0;
}
```

This program will start reading the records from the file **program.bin** in the reverse order (last to first) and prints it.

# C++ Programming Language

# C++ Programming Language

## C++ INTRODUCTION

### What is C++

- C++ is a cross-platform language that can be used to create high-performance applications.
- C++ was developed by Bjarne Stroustrup, as an extension to the C language.
- C++ gives programmers a high level of control over system resources and memory.
- The language was updated 3 major times in 2011, 2014, and 2017 to C++11, C++14, and C++17.

### Why Use C++

C++ is one of the world's most popular programming languages.

C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.

C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.

C++ is portable and can be used to develop applications that can be adapted to multiple platforms.

C++ is fun and easy to learn!

As C++ is close to C# and Java, it makes it easy for programmers to switch to C++ or vice versa.

# C++ Programming Language

## • C++ Get Started

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.

**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.

You can find the latest version of Codeblocks at <http://www.codeblocks.org/downloads/26>. 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.cpp (**File > Save File as**):

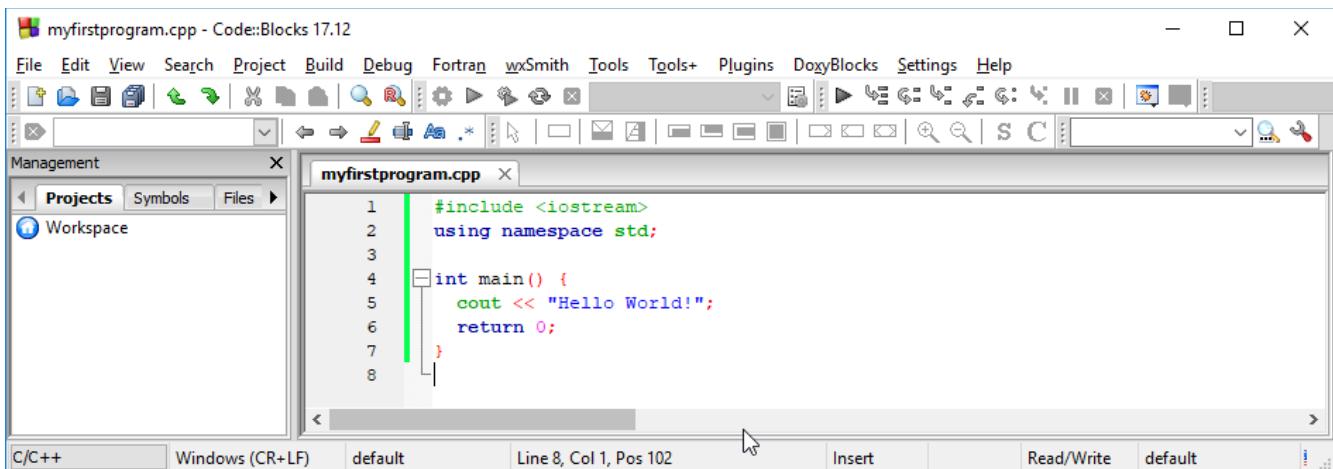
# C++ Programming Language

```
#include <iostream>
using namespace std;

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

Don't worry if you don't understand the code above - we will discuss it in detail. 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

Let's break up the following code to understand it better:

### Example

```
#include <iostream>
using namespace std;
```

# C++ Programming Language

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

```
Hello World!
```

## Example explained

**Line 1:** #include <iostream> is a header file library that lets us work with input and output objects, such as cout (used in line 5). Header files add functionality to C++ programs.

**Line 2:** using namespace std means that we can use names for objects and variables from the standard library.

**Line 3:** A blank line. C++ ignores white space.

**Line 4:** Another thing that always appear in a C++ program, is int main(). This is called a function. Any code inside its curly brackets {} will be executed.

**Line 5:** cout (pronounced "see-out") is an object used together with the insertion operator (<<) to output/print text. In our example it will output "Hello World".

Note: Every C++ statement ends with a semicolon ;.

Note: The body of int main() could also been written as:

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

Remember: The compiler ignores white spaces. However, multiple lines makes the code more readable.

**Line 6:** return 0 ends the main function.

**Line 7:** Do not forget to add the closing curly bracket } to actually end the main function.

## Omitting Namespace

# C++ Programming Language

You might see some C++ programs that runs without the standard namespace library. The **using namespace std** line can be omitted and replaced with the **std** keyword, followed by the **::** operator for some objects:

## Example

```
#include <iostream>

int main() {
    std::cout << "Hello World!";
    return 0;
}
```

```
Hello World!
```

## • C++ Output (Print Text)

The **cout** object, together with the **<<** operator, is used to output values/print text:

## Example

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!";
    return 0;
}
```

```
Hello World!
```

You can add as many **cout** objects as you want. However, note that it does not insert a new line at the end of the output:

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!";
```

# C++ Programming Language

```
cout << "I am learning C++";  
return 0;  
}  
  
Hello World! I am learning C++
```

- **C++ New Lines**

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

### Example

```
#include <iostream>  
using namespace std;  
  
int main() {  
    cout << "Hello World! \n";  
    cout << "I am learning C++";  
    return 0;  
}
```

```
Hello World!  
I am learning C++
```

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

### Example

```
#include <iostream>  
using namespace std;  
  
int main() {  
    cout << "Hello World! \n\n";  
    cout << "I am learning C++";  
    return 0;  
}
```

```
Hello World!  
  
I am learning C++
```

# C++ Programming Language

Another way to insert a new line, is with the `endl` manipulator:

## Example

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello World!" << endl;
    cout << "I am learning C++";
    return 0;
}
```

```
Hello World!
I am learning C++
```

Both `\n` and `endl` are used to break lines. However, `\n` is used more often and is the preferred way.

- ## C++ Comments

Comments can be used to explain C++ code, and to make it more readable. It can also be used to prevent execution when testing alternative code. Comments can be single-lined or multi-lined.

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
cout << "Hello World!";
```

```
Hello World!
```

This example uses a **single-line** comment at the end of a line of code:

# C++ Programming Language

## Example

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!" // This is a comment
    return 0;
}
```

## C++ Multi-line Comments

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

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

## Example

```
#include <iostream>
using namespace std;
int main() {
    /* The code below will print the words Hello World!
       to the screen, and it is amazing */
    cout << "Hello World!";
    return 0;
}
```

Hello World!

## Single or multi-line comments?

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

- C++ Variables

# C++ Programming Language

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
- **double** - 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
- **string** - stores text, such as "Hello World". String values are surrounded by double quotes
- **bool** - stores values with two states: true or false.

## Declaring (Creating) Variables

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

### Syntax

**type variable = value;**

Where type is one of C++ types (such as **int**), and variable is the name of the variable (such as **x** or **myName**). The equal sign is used to assign values to the variable.

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 it the value **15**:

```
#include <iostream>
using namespace std;
int main() {
    int myNum = 15;
    cout << myNum;
    return 0;
}
```

# C++ Programming Language

15

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

## Example

```
#include <iostream>
using namespace std;

int main() {
    int myNum;
    myNum = 15;
    cout << myNum;
    return 0;
}
```

15

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

## Example

```
#include <iostream>
using namespace std;

int main() {
    int myNum = 15; // Now myNum is 15
    myNum = 10;    // Now myNum is 10
    cout << myNum;
    return 0;
}
```

10

# C++ Programming Language

## Other Types

A demonstration of other data types:

### Example

```
int myNum = 5;           // Integer (whole number without decimals)
double myFloatNum = 5.99; // Floating point number (with decimals)
char myLetter = 'D';     // Character
string myText = "Hello"; // String (text)
bool myBoolean = true;   // Boolean (true or false)
```

## Display Variables

The **cout** object is used together with the `<<` operator to display variables.

To combine both text and a variable, separate them with the `<<` operator:

### Example

```
#include <iostream>
using namespace std;

int main() {
    int myAge = 35;
    cout << "I am " << myAge << " years old.";
    return 0;
}
```

```
I am 35 years old.
```

## Add Variables Together

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

- **C++ Declare Multiple Variables**

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

### Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main() {
    int x = 5, y = 6, z = 50;
    cout << x + y + z;
    return 0;
}
```

61

- ## C++ Identifiers

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:

### Example

```
#include <iostream>
using namespace std;

int main() {
    // Good name
    int minutesPerHour = 60;

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

    cout << minutesPerHour << "\n";
    cout << m;
    return 0;
}
```

60  
60

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits and underscores

# C++ Programming Language

- 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 (like C++ keywords, such as int) cannot be used as names
- C++ Constants**

When you do not 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

```
#include <iostream>
using namespace std;

int main() {
    const int myNum = 15;
    myNum = 10;
    cout << myNum;
    return 0;
}
```

```
In function 'int main()':
6.9: 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

```
#include <iostream>
using namespace std;

int main() {
    const int minutesPerHour = 60;
    const float PI = 3.14;
    cout << minutesPerHour << "\n";
    cout << PI;
    return 0;
}
```

# C++ Programming Language

60  
3.14

## • C++ User Input

You have already learned that **cout** is used to output (print) values. Now we will use **cin** to get user input.

**cin** is a predefined variable that reads data from the keyboard with the extraction operator (**>>**).

In the following example, the user **can** input a number, which is stored in the variable **x**.

Then we print the value of **x**:

### Example

```
#include <iostream>
using namespace std;

int main() {
    int x;
    cout << "Type a number: "; // Type a number and press enter
    cin >> x; // Get user input from the keyboard
    cout << "Your number is: " << x;
    return 0;
}
```

Type a number:

### Good To Know

**cout** is pronounced "see-out". Used for **output**, and uses the insertion operator (**<<**)

**cin** is pronounced "see-in". Used for **input**, and uses the extraction operator (**>>**)

## Creating a Simple Calculator

In this example, the user must input two numbers. Then we print the sum by calculating (adding) the two numbers:

### Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main() {
    int x, y;
    int sum;
    cout << "Type a number: ";
    cin >> x;
    cout << "Type another number: ";
    cin >> y;
    sum = x + y;
    cout << "Sum is: " << sum;
    return 0;
}
```

```
Type a number:  
Type another number:
```

## • C++ Data Types

A **variable** in C++ must be a specified data type:

### Example

```
#include <iostream>
#include <string>
using namespace std;

int main () {
    // Creating variables
    int myNum = 5;           // Integer (whole number)
    float myFloatNum = 5.99; // Floating point number
    double myDoubleNum = 9.98; // Floating point number
    char myLetter = 'D';     // Character
    bool myBoolean = true;   // Boolean
    string myString = "Hello"; // String

    // Print variable values
    cout << "int: " << myNum << "\n";
    cout << "float: " << myFloatNum << "\n";
    cout << "double: " << myDoubleNum << "\n";
    cout << "char: " << myLetter << "\n";
    cout << "bool: " << myBoolean << "\n";
    cout << "string: " << myString << "\n";

    return 0;
}
```

# C++ Programming Language

```
int: 5
float: 5.99
double: 9.98
char: D
bool: 1
string: Hello
```

## Basic Data Types

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

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

- C++ Numeric Data Types

### Numeric Types

Use **int** when you need to store a whole number without decimals, like **35** or **1000**, and **float** or **double** when you need a **floating-point** number (with decimals), like 9.99 or 3.14515.

#### int

```
#include <iostream>
using namespace std;

int main () {
    int myNum = 1000;
    cout << myNum;
    return 0;
}
```

```
1000
```

# C++ Programming Language

## float

```
#include <iostream>
using namespace std;

int main () {
    float myNum = 5.75;
    cout << myNum;
    return 0;
}
```

5.75

## Double

```
#include <iostream>
using namespace std;

int main () {
    double myNum = 19.99;
    cout << myNum;
    return 0;
}
```

19.99

## float vs. double

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of **float** is only six or seven decimal digits, while **double** variables have a precision of about 15 digits. Therefore it is safer to use **double** for most calculations.

## Scientific Numbers

A floating-point number can also be a scientific number with an "e" to indicate the power of 10:

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main () {
    float f1 = 35e3;
    double d1 = 12E4;
    cout << f1 << "\n";
    cout << d1;
    return 0;
}
```

```
35000
120000
```

- C++ Boolean Data Types

## Boolean Types

A **boolean** data type is declared with the `bool` keyword and can only take the values true or false. When the value is returned, true = 1 and false = 0.

### Example

```
#include <iostream>
using namespace std;

int main() {
    bool isCodingFun = true;
    bool isFishTasty = false;
    cout << isCodingFun << "\n";
    cout << isFishTasty;
    return 0;
}
```

```
1
0
```

- C++ Character Data Types

## Character Types

The `char` data type is used to store a single character. The character must be surrounded by single quotes, like '`'A'`' or '`'c'`:

### Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main () {
    char myGrade = 'B';
    cout << myGrade;
    return 0;
}
```

B

Alternatively, you can use ASCII values to display certain characters:

## Example

```
#include <iostream>
using namespace std;

int main () {
    char a = 65, b = 66, c = 67;
    cout << a;
    cout << b;
    cout << c;
    return 0;
}
```

ABC

## • C++ String Data Types

### String Types

The string type is used to store a sequence of characters (text). This is not a built-in type, but it behaves like one in its most basic usage. String values must be surrounded by double quotes:

## Example

```
string greeting = "Hello";
cout << greeting;
```

To use strings, you must include an additional header file in the source code, the `<string>` library:

# C++ Programming Language

## Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string greeting = "Hello";
    cout << greeting;
    return 0;
}
```

```
Hello
```

## • 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

```
#include <iostream>
using namespace std;

int main() {
    int x = 100 + 50;
    cout << x;
    return 0;
}
```

```
150
```

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

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main() {
    int sum1 = 100 + 50;           // 150 (100 + 50)
    int sum2 = sum1 + 250;         // 400 (150 + 250)
    int sum3 = sum2 + sum2;       // 800 (400 + 400)
    cout << sum1 << "\n";
    cout << sum2 << "\n";
    cout << sum3;
    return 0;
}
```

```
150
400
800
```

- **C++ 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

```
#include <iostream>
using namespace std;

int main() {
    int x = 10;
    cout << x;
    return 0;
}
```

```
10
```

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

### Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main() {
    int x = 10;
    x += 5;
    cout << x;
    return 0;
}
```

```
15
```

## • C++ 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

```
#include <iostream>
using namespace std;

int main() {
    int x = 5;
    int y = 3;
    cout << (x > y); // returns 1 (true) because 5 is greater than 3
    return 0;
}
```

```
1
```

A list of all comparison operators:

# C++ Programming Language

Operator	Name	Example
<code>==</code>	Equal to	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>&gt;</code>	Greater than	<code>x &gt; y</code>
<code>&lt;</code>	Less than	<code>x &lt; y</code>
<code>&gt;=</code>	Greater than or equal to	<code>x &gt;= y</code>
<code>&lt;=</code>	Less than or equal to	<code>x &lt;= y</code>

## • C++ Logical Operators

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

Operator	Name	Description	Example
<code>&amp;&amp;</code>	Logical and	Returns true if both statements are true	<code>x &lt; 5 &amp;&amp; x &lt; 10</code>
<code>  </code>	Logical or	Returns true if one of the statements is true	<code>x &lt; 5    x &lt; 4</code>
<code>!</code>	Logical not	Reverse the result, returns false if the result is true	<code>!(x &lt; 5 &amp;&amp; x &lt; 10)</code>

## • C++ Strings

**Strings** are used for storing text.

A **string variable** contains a collection of characters surrounded by double quotes:

### Example

Create a variable of type string and assign it a value:

```
string greeting = "Hello";
```

To use strings, you must include an additional header file in the source code, the `<string>` library:

### Example

# C++ Programming Language

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string greeting = "Hello";
    cout << greeting;
    return 0;
}
```

```
Hello
```

## • C++ String Concatenation

The `+` operator can be used between strings to add them together to make a new string.

This is called **concatenation**:

### Example

```
#include <iostream>
#include <string>
using namespace std;

int main () {
    string firstName = "John ";
    string lastName = "Doe";
    string fullName = firstName + lastName;
    cout << fullName;
    return 0;
}
```

```
John Doe
```

In the example above, we added a space after `firstName` to create a space between John and Doe on output. However, you could also add a space with quotes (" " or ' '):

### Example

# C++ Programming Language

```
#include <iostream>
#include <string>
using namespace std;

int main () {
    string firstName = "John";
    string lastName = "Doe";
    string fullName = firstName + " " + lastName;
    cout << fullName;
    return 0;
}
```

```
John Doe
```

## Append

A **string** in C++ is actually an object, which contain functions that can perform certain operations on strings. For example, you can also concatenate strings with the **append()** function:

### Example

```
#include <iostream>
#include <string>
using namespace std;

int main () {
    string firstName = "John ";
    string lastName = "Doe";
    string fullName = firstName.append(lastName);
    cout << fullName;
    return 0;
}
```

```
John Doe
```

It is up to you whether you want to use **+** or **append()**. The major difference between the two, is that the **append()** function is much faster. However, for testing and such, it might be easier to just use **+**.

- **C++ Numbers and Strings**

# C++ Programming Language

## Adding Numbers and Strings

### WARNING!

C++ uses the `+` operator for both **addition** and **concatenation**.

Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

### Example

```
#include <iostream>
using namespace std;

int main () {
    int x = 10;
    int y = 20;
    int z = x + y;
    cout << z;
    return 0;
}
```

```
30
```

If you add two strings, the result will be a string concatenation:

### Example

```
#include <iostream>
#include <string>
using namespace std;

int main () {
    string x = "10";
    string y = "20";
    string z = x + y;
    cout << z;
    return 0;
}
```

```
1020
```

If you try to add a number to a string, an error occurs:

### Example

# C++ Programming Language

```
string x = "10";
int y = 20;
string z = x + y;
```

- ## C++ String Length

### String Length

To get the length of a string, use the **length()** function:

#### Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string txt = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
    cout << "The length of the txt string is: " << txt.length();
    return 0;
}
```

```
The length of the txt string is: 26
```

**Tip:** You might see some C++ programs that use the **size()** function to get the length of a string. This is just an alias of **length()**. It is completely up to you if you want to use **length()** or **size()**:

#### Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string txt = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
    cout << "The length of the txt string is: " << txt.size();
    return 0;
}
```

```
The length of the txt string is: 26
```

- ## C++ Access Strings

# C++ Programming Language

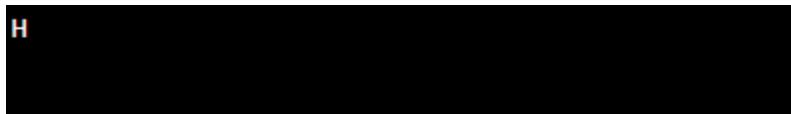
you can access the characters in a string by referring to its index number inside square brackets [ ].

This example prints the first character in **myString**:

## Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string myString = "Hello";
    cout << myString[0];
    return 0;
}
```



**Note:** String indexes start with 0: [0] is the first character. [1] is the second character, etc.

This example prints the second character in myString:

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string myString = "Hello";
    cout << myString[1];
    return 0;
}
```



## Change String Characters

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

## Example

# C++ Programming Language

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string myString = "Hello";
    myString[0] = 'J';
    cout << myString;
    return 0;
}
```

Jello

- **C++ User Input Strings**

It is possible to use the extraction operator `>>` on `cin` to display a string entered by a user:

### Example

```
string firstName;

cout << "Type your first name: ";
cin >> firstName; // get user input from the keyboard

cout << "Your name is: " << firstName;

// Type your first name: John
// Your name is: John
```

However, `cin` considers a space (whitespace, tabs, etc) as a terminating character, which means that it can only display a single word (even if you type many words):

### Example

```
string fullName;

cout << "Type your full name: ";
cin >> fullName;

cout << "Your name is: " << fullName;

// Type your full name: John Doe
// Your name is: John
```

# C++ Programming Language

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

That's why, when working with strings, we often use the **getline()** function to read a line of text. It takes **cin** as the first parameter, and the string variable as second:

## Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string fullName;
    cout << "Type your full name: ";
    getline (cin, fullName);
    cout << "Your name is: " << fullName;
    return 0;
}
```

```
Type your full name: elaf
Your name is: elaf
```

- ## C++ String Namespace

### Omitting Namespace

You might see some C++ programs that runs without the standard namespace library. The `using namespace std` line can be omitted and replaced with the `std` keyword, followed by the `::` operator for string (and **cout**) objects:

## Example

```
#include <iostream>
#include <string>

int main() {
    std::string greeting = "Hello";
    std::cout << greeting;
    return 0;
}
```

```
Hello
```

# C++ Programming Language

It is up to you if you want to include the standard namespace library or not.

- ## C++ Math

C++ has many functions that allows you to perform mathematical tasks on numbers.

### Max and min

The **max(x,y)** function can be used to find the highest value of x and y:

#### Example

```
#include <iostream>
using namespace std;

int main() {
    cout << max(5, 10);
    return 0;
}
```

10

And the **min(x,y)** function can be used to find the lowest value of x and y:

#### Example

```
#include <iostream>
using namespace std;

int main() {
    cout << min(5, 10);
    return 0;
}
```

5

### C++ <cmath> Header

Other functions, such as **sqrt** (square root), **round** (rounds a number) and **log** (natural logarithm), can be found in the **<cmath>** header file:

#### Example

# C++ Programming Language

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    cout << sqrt(64) << "\n";
    cout << round(2.6) << "\n";
    cout << log(2) << "\n";
    return 0;
}
```

```
8
3
0.693147
```

## Other Math Functions

A list of other popular Math functions (from the `<cmath>` library) can be found in the table below:

Function	Description
<code>abs(x)</code>	Returns the absolute value of x
<code>acos(x)</code>	Returns the arccosine of x, in radians
<code>asin(x)</code>	Returns the arcsine of x, in radians
<code>atan(x)</code>	Returns the arctangent of x, in radians
<code>cbrt(x)</code>	Returns the cube root of x
<code>ceil(x)</code>	Returns the value of x rounded up to its nearest integer
<code>cos(x)</code>	Returns the cosine of x, in radians
<code>cosh(x)</code>	Returns the hyperbolic cosine of x, in radians
<code>exp(x)</code>	Returns the value of E <sup>x</sup>
<code>expm1(x)</code>	Returns e <sup>x</sup> - 1
<code>fabs(x)</code>	Returns the absolute value of a floating x

# C++ Programming Language

fdim(x, y)	Returns the positive difference between x and y
floor(x)	Returns the value of x rounded down to its nearest integer
hypot(x, y)	Returns $\sqrt{x^2 + y^2}$ without intermediate overflow or underflow
fma(x, y, z)	Returns $x * y + z$ without losing precision
fmax(x, y)	Returns the highest value of a floating x and y
fmin(x, y)	Returns the lowest value of a floating x and y
fmod(x, y)	Returns the floating point remainder of x/y
pow(x, y)	Returns the value of x to the power of y
sin(x)	Returns the sine of x (x is in radians)
sinh(x)	Returns the hyperbolic sine of a double value
tan(x)	Returns the tangent of an angle
tanh(x)	Returns the hyperbolic tangent of a double value

## • C++ Booleans

Very often, in programming, you 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 can take the values **true (1)** or **false (0)**.

## Boolean Values

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

### Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main() {
    bool isCodingFun = true;
    bool isFishTasty = false;
    cout << isCodingFun << "\n";
    cout << isFishTasty;
    return 0;
}
```

```
1
0
```

From the example above, you can read that a **true** value returns **1**, and **false** returns **0**.

- ## C++ Boolean Expressions

A **Boolean expression** is a C++ expression that returns a boolean value: **1** (true) or **0** (false).

You can use a comparison operator, such as the greater than (**>**) operator to find out if an expression (or a variable) is **true**:

### Example

```
#include <iostream>
using namespace std;

int main() {
    int x = 10;
    int y = 9;
    cout << (x > y);
    return 0;
}
```

```
1
```

Or even easier:

### Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main() {
    cout << (10 > 9);
    return 0;
}
```

```
1
```

In the examples below, we use the equal to (`==`) operator to evaluate an expression:

## Example

```
#include <iostream>
using namespace std;

int main() {
    int x = 10;
    cout << (x == 10);
    return 0;
}
```

```
1
```

## Example

```
#include <iostream>
using namespace std;

int main() {
    cout << (10 == 15);
    return 0;
}
```

```
0
```

- C++ If ... Else

## C++ Conditions and If Statements

C++ supports the usual logical conditions from mathematics:

# C++ Programming Language

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

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main() {
    if (20 > 18) {
        cout << "20 is greater than 18";
    }
    return 0;
}
```

```
20 is greater than 18
```

We can also test variables:

## Example

```
#include <iostream>
using namespace std;

int main() {
    int x = 20;
    int y = 18;
    if (x > y) {
        cout << "x is greater than y";
    }
    return 0;
}
```

```
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".

- ### C++ Else

## The else Statement

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

### Syntax

# C++ Programming Language

```
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

```
#include <iostream>  
using namespace std;  
  
int main() {  
    int time = 20;  
    if (time < 18) {  
        cout << "Good day.";  
    } else {  
        cout << "Good evening.";  
    }  
    return 0;  
}
```

```
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".

- ### C++ Else If

## 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) {
```

# C++ Programming Language

```
// 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

```
#include <iostream>  
using namespace std;  
  
int main() {  
    int time = 22;  
    if (time < 10) {  
        cout << "Good morning.";  
    } else if (time < 20) {  
        cout << "Good day.";  
    } else {  
        cout << "Good evening.";  
    }  
    return 0;  
}
```

```
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."

- **C++ Short Hand If Else**

## 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:

# C++ Programming Language

## Syntax

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

Instead of writing:

## Example

```
#include <iostream>
using namespace std;

int main() {
    int time = 20;
    if (time < 18) {
        cout << "Good day.";
    } else {
        cout << "Good evening.";
    }
    return 0;
}
```

Good evening.

You can simply write:

## Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    int time = 20;
    string result = (time < 18) ? "Good day." : "Good evening.";
    cout << result;
    return 0;
}
```

Good evening.

- ## C++ Switch

## C++ Switch Statements

Use the switch statement to select one of many code blocks to be executed.

# C++ Programming Language

## 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 and default keywords are optional, and will be described later in this chapter

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

## Example

```
#include <iostream>  
using namespace std;  
  
int main() {  
    int day = 4;  
    switch (day) {  
    case 1:  
        cout << "Monday";  
        break;  
    case 2:  
        cout << "Tuesday";  
        break;  
    case 3:  
        cout << "Wednesday";  
        break;  
    case 4:  
        cout << "Thursday";  
        break;  
    case 5:  
        cout << "Friday";  
        break;  
    case 6:  
        cout << "Saturday";  
        break;  
    case 7:  
        cout << "Sunday";  
        break;  
    }  
    return 0;  
}
```

# C++ Programming Language

Thursday

## 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

```
#include <iostream>
using namespace std;

int main() {
    int day = 4;
    switch (day) {
        case 6:
            cout << "Today is Saturday";
            break;
        case 7:
            cout << "Today is Sunday";
            break;
        default:
            cout << "Looking forward to the Weekend";
    }
    return 0;
}
```

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.

- **C++ While Loop**

# C++ Programming Language

## C++ 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.

## C++ 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

```
#include <iostream>  
using namespace std;  
  
int main() {  
    int i = 0;  
    while (i < 5) {  
        cout << i << "\n";  
        i++;  
    }  
    return 0;  
}
```

```
0  
1  
2  
3  
4
```

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

# C++ Programming Language

- C++ Do/While Loop

## 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

```
#include <iostream>  
using namespace std;  
  
int main() {  
    int i = 0;  
    do {  
        cout << i << "\n";  
        i++;  
    }  
    while (i < 5);  
    return 0;  
}
```

```
0  
1  
2  
3  
4
```

# C++ Programming Language

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

- ## C++ 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

```
#include <iostream>  
using namespace std;  
  
int main() {  
    for (int i = 0; i < 5; i++) {  
        cout << i << "\n";  
    }  
    return 0;  
}
```

```
0  
1  
2  
3  
4
```

### Example explained

**Statement 1** sets a variable before the loop starts (int i = 0).

# C++ Programming Language

**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:

```
#include <iostream>
using namespace std;

int main() {
    for (int i = 0; i <= 10; i = i + 2) {
        cout << i << "\n";
    }
    return 0;
}
```

```
0
2
4
6
8
10
```

## • C++ Break and Continue

### C++ 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

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 4) {
            break;
        }
        cout << i << "\n";
    }
    return 0;
}
```

```
0
1
2
3
```

## C++ 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

```
#include <iostream>
using namespace std;

int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 4) {
            continue;
        }
        cout << i << "\n";
    }
    return 0;
}
```

```
0
1
2
3
5
6
7
8
9
```

# C++ Programming Language

## Break and Continue in While Loop

You can also use break and continue in while loops:

### Break Example

```
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    while (i < 10) {
        cout << i << "\n";
        i++;
        if (i == 4) {
            break;
        }
    }
    return 0;
}
```

```
0
1
2
3
```

### Continue Example

```
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    while (i < 10) {
        if (i == 4) {
            i++;
            continue;
        }
        cout << i << "\n";
        i++;
    }
    return 0;
}
```

# C++ Programming Language

```
0  
1  
2  
3  
5  
6  
7  
8  
9
```

- **C++ Arrays**

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

To declare an array, define the variable type, specify the name of the array followed by square brackets and specify the number of elements it should store:

**string cars[4];**

We have now declared a variable that holds an array of four strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

**string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};**

To create an array of three integers, you could write:

**int myNum[3] = {10, 20, 30};**

## Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

### Example

# C++ Programming Language

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
    cout << cars[0];
    return 0;
}
```

```
Volvo
```

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

## Change an Array Element

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

### Example

```
cars[0] = "Opel";
```

### Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
    cars[0] = "Opel";
    cout << cars[0];
    return 0;
}
```

```
Opel
```

## • C++ Arrays and Loops

### Loop Through an Array

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

The following example outputs all elements in the cars array:

# C++ Programming Language

## Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
    for(int i = 0; i < 4; i++) {
        cout << cars[i] << "\n";
    }
    return 0;
}
```

```
Volvo
BMW
Ford
Mazda
```

The following example outputs the index of each element together with its value:

## Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
    for(int i = 0; i < 4; i++) {
        cout << i << ":" << cars[i] << "\n";
    }
    return 0;
}
```

```
0: Volvo
1: BMW
2: Ford
3: Mazda
```

## • C++ Omit Array Size

### Omit Array Size

You don't have to specify the size of the array. But if you don't, it will only be as big as the elements that are inserted into it:

# C++ Programming Language

```
string cars[] = {"Volvo", "BMW", "Ford"}; // size of array is always 3
```

This is completely fine. However, the problem arise if you want extra space for future elements. Then you have to overwrite the existing values:

```
string cars[] = {"Volvo", "BMW", "Ford"};
string cars[] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
```

If you specify the size however, the array will reserve the extra space:

```
string cars[5] = {"Volvo", "BMW", "Ford"}; // size of array is 5, even though it's
only three elements inside it
```

Now you can add a fourth and fifth element without overwriting the others:

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string cars[5] = {"Volvo", "BMW", "Ford"};
    cars[3] = "Mazda";
    cars[4] = "Tesla";
    for(int i = 0; i < 5; i++) {
        cout << cars[i] << "\n";
    }
    return 0;
}
```

```
Volvo
BMW
Ford
Mazda
Tesla
```

## Omit Elements on Declaration

It is also possible to declare an array without specifying the elements on declaration, and add them later:

# C++ Programming Language

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string cars[5];
    cars[0] = "Volvo";
    cars[1] = "BMW";
    cars[2] = "Ford";
    cars[3] = "Mazda";
    cars[4] = "Tesla";
    for(int i = 0; i < 5; i++) {
        cout << cars[i] << "\n";
    }
    return 0;
}
```

```
Volvo
BMW
Ford
Mazda
Tesla
```

- **C++ References**

## Creating References

A reference variable is a "reference" to an existing variable, and it is created with the & operator:

```
string food = "Pizza"; // food variable
```

```
string &meal = food; // reference to food
```

Now, we can use either the variable name food or the reference name meal to refer to the food variable:

### Example

# C++ Programming Language

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string food = "Pizza";
    string &meal = food;

    cout << food << "\n";
    cout << meal << "\n";
    return 0;
}
```

```
Pizza
Pizza
```

## • C++ Memory Address

In the example from the previous page, the **&** operator was used to create a reference variable. But it can also be used to get the memory address of a variable; which is the location of where the variable is stored on the computer.

When a variable is created in C++, a memory address is assigned to the variable. And when we assign a value to the variable, it is stored in this memory address.

To access it, use the **&** operator, and the result will represent where the variable is stored:

### Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string food = "Pizza";

    cout << &food;
    return 0;
}
```

```
0x6dfed4
```

**Note:** The memory address is in hexadecimal form (0x..). Note that you may not get the same result in your program.

# C++ Programming Language

## And why is it useful to know the memory address?

References and Pointers (which you will learn about in the next chapter) are important in C++, because they give you the ability to manipulate the data in the computer's memory - which can reduce the code and improve the performance.

- ### C++ Pointers

#### Creating Pointers

You learned from the previous chapter, that we can get the memory address of a variable by using the & operator:

#### Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string food = "Pizza";

    cout << food << "\n";
    cout << &food << "\n";
    return 0;
}
```

```
Pizza
0x6dfed4
```

A **pointer** however, is a variable that stores the memory address as its value.

A **pointer** variable points to a data type (like int or string) 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

# C++ Programming Language

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string food = "Pizza"; // A string variable
    string* ptr = &food; // A pointer variable that stores the address of food

    // Output the value of food
    cout << food << "\n";

    // Output the memory address of food
    cout << &food << "\n";

    // Output the memory address of food with the pointer
    cout << ptr << "\n";
    return 0;
}
```

```
Pizza
0x6dfed4
0x6dfed4
```

## Example explained

Create a pointer variable with the name `ptr`, that points to a string variable, by using the asterisk sign `*` (`string* ptr`). 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 variable called `food`, and assign it to the pointer.

Now, `ptr` holds the value of `food`'s memory address.

**Tip:** There are three ways to declare pointer variables, but the first way is preferred:

```
string* mystring; // Preferred
string *mystring;
string * mystring;
```

- ## C++ Dereference

## Get Memory Address and Value

# C++ Programming Language

In the example from the previous page, we used the pointer variable to get the memory address of a variable (used together with the & reference operator). However, you can also use the pointer to get the value of the variable, by using the \* operator (the dereference operator):

## Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string food = "Pizza"; // Variable declaration
    string* ptr = &food; // Pointer declaration

    // Reference: Output the memory address of food with the pointer
    cout << ptr << "\n";

    // Dereference: Output the value of food with the pointer
    cout << *ptr << "\n";
    return 0;
}
```

```
0x6dfed4
Pizza
```

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

- When used in declaration (**string\*** **ptr**), it creates a pointer variable.
- When not used in declaration, it act as a dereference operator.
- **C++ Modify Pointers**

## Modify the Pointer Value

You can also change the pointer's value. But note that this will also change the value of the original variable:

## Example

# C++ Programming Language

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string food = "Pizza";
    string* ptr = &food;

    // Output the value of food
    cout << food << "\n";

    // Output the memory address of food
    cout << &food << "\n";

    // Access the memory address of food and output its value
    cout << *ptr << "\n";

    // Change the value of the pointer
    *ptr = "Hamburger";

    // Output the new value of the pointer
    cout << *ptr << "\n";

    // Output the new value of the food variable
    cout << food << "\n";
    return 0;
}
```

```
Pizza
0x6dfed4
Pizza
Hamburger
Hamburger
```

# C++ Programming Language

## 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.

### Create a Function

C++ provides some pre-defined functions, such as `main()`, which is used to execute code.

But you can also create your own functions to perform certain actions.

To create (often referred to as declare) a function, specify the name of the function, followed by parentheses ():

#### 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 later, when they are called.

To call a function, write the function's name followed by two parentheses () and a semicolon ;

# C++ Programming Language

In the following example, **myFunction()** is used to print a text (the action), when it is called:

## Example

```
#include <iostream>
using namespace std;

void myFunction() {
    cout << "I just got executed!";
}

int main() {
    myFunction();
    return 0;
}
```

```
I just got executed!
```

A function can be called multiple times:

## Example

```
#include <iostream>
using namespace std;

void myFunction() {
    cout << "I just got executed!\n";
}

int main() {
    myFunction();
    myFunction();
    myFunction();
    return 0;
}
```

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

## Function Declaration and Definition

A C++ function consist of two parts:

**Declaration:** the function's name, return type, and parameters (if any)

# C++ Programming Language

**Definition:** the body of the function (code to be executed).

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

**Note:** If a user-defined function, such as myFunction() is declared after the main() function, an error will occur. It is because C++ works from top to bottom; which means that if the function is not declared above main(), the program is unaware of it:

## Example

```
#include <iostream>  
using namespace std;  
  
int main() {  
    myFunction();  
    return 0;  
}  
  
void myFunction() {  
    cout << "I just got executed!";  
}
```

```
In function 'int main()':  
5:3: error: 'myFunction' was not declared in this scope
```

However, it is possible to separate the declaration and the definition of the function - for code optimization.

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

# C++ Programming Language

```
#include <iostream>
using namespace std;

// Function declaration
void myFunction();

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

// Function definition
void myFunction() {
    cout << "I just got executed!";
}
```

```
I just got executed!
```

- ## C++ Function Parameters

### 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

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

The following example has a function that takes a string called **fname** as parameter. When the function is called, we pass along a first name, which is used inside the function to print the full name:

#### Example

# C++ Programming Language

```
#include <iostream>
#include <string>
using namespace std;

void myFunction(string fname) {
    cout << fname << " Refsnes\n";
}

int main() {
    myFunction("Liam");
    myFunction("Jenny");
    myFunction("Anja");
    return 0;
}
```

```
Liam Refsnes
Jenny Refsnes
Anja Refsnes
```

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

- ## C++ Default Parameters

### Default Parameter Value

You can also use a default parameter value, by using the equals sign (=).

If we call the function without an argument, it uses the default value ("Norway"):

#### Example

```
#include <iostream>
#include <string>
using namespace std;

void myFunction(string country = "Norway") {
    cout << country << "\n";
}

int main() {
    myFunction("Sweden");
    myFunction("India");
    myFunction();
    myFunction("USA");
    return 0;
}
```

# C++ Programming Language

Sweden  
India  
Norway  
USA

A parameter with a default value, is often known as an "**optional parameter**". From the example above, **country** is an optional parameter and "Norway" is the default value.

- ## C++ Multiple Parameters

### Multiple Parameters

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

#### Example

```
#include <iostream>
#include <string>
using namespace std;

void myFunction(string fname, int age) {
    cout << fname << " Refsnes. " << age << " years old. \n";
}

int main() {
    myFunction("Liam", 3);
    myFunction("Jenny", 14);
    myFunction("Anja", 30);
    return 0;
}
```

```
Liam Refsnes. 3 years old.
Jenny Refsnes. 14 years old.
Anja Refsnes. 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.

- ## C++ The Return Keyword

### Return Values

# C++ Programming Language

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**, **string**, etc.) instead of **void**, and use the **return** keyword inside the function:

## Example

```
#include <iostream>
using namespace std;

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

int main() {
    cout << myFunction(3);
    return 0;
}
```

8

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

## Example

```
#include <iostream>
using namespace std;

int myFunction(int x, int y) {
    return x + y;
}

int main() {
    cout << myFunction(5, 3);
    return 0;
}
```

8

You can also store the result in a variable:

## Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

int myFunction(int x, int y) {
    return x + y;
}

int main() {
    int z = myFunction(5, 3);
    cout << z;
    return 0;
}
```

8

- C++ Functions - Pass By Reference

## Pass By Reference

In the examples from the previous page, we used normal variables when we passed parameters to a function. You can also pass a reference to the function. This can be useful when you need to change the value of the arguments:

### Example

```
#include <iostream>
using namespace std;

void swapNums(int &x, int &y) {
    int z = x;
    x = y;
    y = z;
}

int main() {
    int firstNum = 10;
    int secondNum = 20;

    cout << "Before swap: " << "\n";
    cout << firstNum << secondNum << "\n";

    swapNums(firstNum, secondNum);

    cout << "After swap: " << "\n";
    cout << firstNum << secondNum << "\n";

    return 0;
}
```

# C++ Programming Language

Before swap:

1020

After swap:

2010

## • C++ Function Overloading

### Function Overloading

With function overloading, multiple functions can have the same name with different parameters:

#### Example

```
int myFunction(int x)  
float myFunction(float x)  
double myFunction(double x, double y)
```

Consider the following example, which have two functions that add numbers of different type:

#### Example

```
#include <iostream>  
using namespace std;  
  
int plusFuncInt(int x, int y) {  
    return x + y;  
}  
  
double plusFuncDouble(double x, double y) {  
    return x + y;  
}  
  
int main() {  
    int myNum1 = plusFuncInt(8, 5);  
    double myNum2 = plusFuncDouble(4.3, 6.26);  
    cout << "Int: " << myNum1 << "\n";  
    cout << "Double: " << myNum2;  
    return 0;  
}
```

```
Int: 13  
Double: 10.56
```

# C++ Programming Language

Instead of defining two functions that should do the same thing, it is better to overload one.

In the example below, we overload the **plusFunc** function to work for both int and double:

## Example

```
#include <iostream>
using namespace std;

int plusFunc(int x, int y) {
    return x + y;
}

double plusFunc(double x, double y) {
    return x + y;
}

int main() {
    int myNum1 = plusFunc(8, 5);
    double myNum2 = plusFunc(4.3, 6.26);
    cout << "Int: " << myNum1 << "\n";
    cout << "Double: " << myNum2;
    return 0;
}
```

```
Int: 13
Double: 10.56
```

**Note:** Multiple functions can have the same name as long as the number and/or type of parameters are different.

# C++ Programming Language

## C++ CLASSES

- C++ OOP

### C++ What is OOP?

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

Object-oriented programming has several advantages over procedural programming:

- **OOP** is faster and easier to execute
- **OOP** provides a clear structure for the programs
- **OOP** helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- **OOP** makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

### C++ What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

# C++ Programming Language

class	objects
Fruit	Apple
	Banana
	Mango

Another example:

class	objects
Car	Volvo
	Audi
	Toyota

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and functions from the class.

- **C++ Classes and Objects**

## C++ Classes/Objects

C++ is an object-oriented programming language.

Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

Attributes and methods are basically variables and functions that belongs to the class.

These are often referred to as "class members".

A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

## Create a Class

# C++ Programming Language

To create a class, use the class keyword:

## Example

Create a class called "**MyClass**":

```
class MyClass {      // The class
public:            // Access specifier
    int myNum;     // Attribute (int variable)
    string myString; // Attribute (string variable)
};
```

## Example explained

The class keyword is used to create a class called MyClass.

The public keyword is an access specifier, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about access specifiers later.

Inside the class, there is an integer variable myNum and a string variable myString. When variables are declared within a class, they are called attributes.

At last, end the class definition with a **semicolon ;**.

## Create an Object

In C++, an object is created from a class. We have already created the class named **MyClass**, so now we can use this to create objects.

To create an object of **MyClass**, specify the class name, followed by the object name.

To access the class attributes (**myNum** and **myString**), use the dot syntax (.) on the object:

## Example

# C++ Programming Language

```
#include <iostream>
#include <string>
using namespace std;

class MyClass {      // The class
public:             // Access specifier
    int myNum;       // Attribute (int variable)
    string myString; // Attribute (string variable)
};

int main() {
    MyClass myObj; // Create an object of MyClass

    // Access attributes and set values
    myObj.myNum = 15;
    myObj.myString = "Some text";

    // Print values
    cout << myObj.myNum << "\n";
    cout << myObj.myString;
    return 0;
}
```

```
15
Some text
```

## Multiple Objects

You can create multiple objects of one class:

### Example

```
#include <iostream>
#include <string>
using namespace std;

class Car {
public:
    string brand;
    string model;
    int year;
};

int main() {
    Car carObj1;
    carObj1.brand = "BMW";
    carObj1.model = "X5";
    carObj1.year = 1999;

    Car carObj2;
    carObj2.brand = "Ford";
    carObj2.model = "Mustang";
    carObj2.year = 1969;

    cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
    cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
    return 0;
}
```

# C++ Programming Language

BMW X5 1999

Ford Mustang 1969

## • C++ Class Methods

### Class Methods

Methods are functions that belongs to the class.

There are two ways to define functions that belongs to a class:

- Inside class definition
- Outside class definition

In the following example, we define a function inside the class, and we name it "**myMethod**".

Note: You access methods just like you access attributes; by creating an object of the class and by using the dot syntax (.):

### Inside Example

```
#include <iostream>
using namespace std;

class MyClass {          // The class
public:                  // Access specifier
    void myMethod() {   // Method/function
        cout << "Hello World!";
    }
};

int main() {
    MyClass myObj;      // Create an object of MyClass
    myObj.myMethod();    // Call the method
    return 0;
}
```

Hello World!

# C++ Programming Language

To define a function outside the class definition, you have to declare it inside the class and then define it outside of the class. This is done by specifying the name of the class, followed the scope resolution `::` operator, followed by the name of the function:

## Outside Example

```
#include <iostream>
using namespace std;

class MyClass {          // The class
public:                  // Access specifier
    void myMethod();    // Method/function declaration
};

// Method/function definition outside the class
void MyClass::myMethod() {
    cout << "Hello World!";
}

int main() {
    MyClass myObj;      // Create an object of MyClass
    myObj.myMethod();   // Call the method
    return 0;
}
```

```
Hello World!
```

## Parameters

You can also add parameters:

## Example

```
#include <iostream>
using namespace std;

class Car {
public:
    int speed(int maxSpeed);
};

int Car::speed(int maxSpeed) {
    return maxSpeed;
}

int main() {
    Car myObj;
    cout << myObj.speed(200);
    return 0;
}
```

# C++ Programming Language

200

- ## C++ Constructors

### Constructors

A **constructor** in C++ is a special method that is automatically called when an object of a class is created.

To create a constructor, use the same name as the class, followed by parentheses () .

#### Example

```
#include <iostream>
using namespace std;

class MyClass {      // The class
public:             // Access specifier
    MyClass() {    // Constructor
        cout << "Hello World!";
    }
};

int main() {
    MyClass myObj;    // Create an object of MyClass (this will call the constructor)
    return 0;
}
```

Hello World!

**Note:** The constructor has the same name as the class, it is always **public**, and it does not have any return value.

### Constructor Parameters

**Constructors** can also take parameters (just like regular functions), which can be useful for setting initial values for attributes.

The following class have brand, model and year attributes, and a constructor with different parameters. Inside the constructor we set the attributes equal to the constructor parameters (brand=x, etc). When we call the constructor (by creating an object of the class), we pass

# C++ Programming Language

parameters to the constructor, which will set the value of the corresponding attributes to the same:

## Example

```
#include <iostream>
using namespace std;

class Car {          // The class
public:             // Access specifier
    string brand;   // Attribute
    string model;   // Attribute
    int year;       // Attribute
    Car(string x, string y, int z) { // Constructor with parameters
        brand = x;
        model = y;
        year = z;
    }
};

int main() {
    // Create Car objects and call the constructor with different values
    Car carObj1("BMW", "X5", 1999);
    Car carObj2("Ford", "Mustang", 1969);

    // Print values
    cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
    cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
    return 0;
}
```

```
BMW X5 1999
Ford Mustang 1969
```

Just like functions, constructors can also be defined outside the class. First, declare the constructor inside the class, and then define it outside of the class by specifying the name of the class, followed by the scope resolution `::` operator, followed by the name of the constructor (which is the same as the class):

## Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

class Car {          // The class
public:             // Access specifier
    string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute
    Car(string x, string y, int z); // Constructor declaration
};

// Constructor definition outside the class
Car::Car(string x, string y, int z) {
    brand = x;
    model = y;
    year = z;
}

int main() {
    // Create Car objects and call the constructor with different values
    Car carObj1("BMW", "X5", 1999);
    Car carObj2("Ford", "Mustang", 1969);

    // Print values
    cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
    cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
    return 0;
}
```

```
BMW X5 1999
Ford Mustang 1969
```

- ## C++ Access Specifiers

By now, you are quite familiar with the public keyword that appears in all of our class examples:

### Example

```
#include <iostream>
using namespace std;

class MyClass {    // The class
public:            // Public access specifier
    int x;        // Public attribute (int variable)
};

int main() {
    MyClass myObj; // Create an object of MyClass

    // Access attributes and set values
    myObj.x = 15;

    // Print values
    cout << myObj.x;
    return 0;
}
```

# C++ Programming Language

15

The **public** keyword is an access specifier. Access specifiers define how the members (attributes and methods) of a class can be accessed. In the example above, the members are public - which means that they can be accessed and modified from outside the code. However, what if we want members to be private and hidden from the outside world?

In C++, there are three access specifiers:

- **public** - members are accessible from outside the class
- **private** - members cannot be accessed (or viewed) from outside the class
- **protected** - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

In the following example, we demonstrate the differences between **public** and **private** members:

## Example

```
#include <iostream>
using namespace std;

class MyClass {
    public: // Public access specifier
        int x; // Public attribute
    private: // Private access specifier
        int y; // Private attribute
};

int main() {
    MyClass myObj;
    myObj.x = 25; // Allowed (x is public)
    myObj.y = 50; // Not allowed (y is private)
    return 0;
}
```

```
In function 'int main()':
Line 8: error: 'int MyClass::y' is private
Line 14: error: within this context
```

**Note:** By default, all members of a class are private if you don't specify an access specifier:

# C++ Programming Language

## Example

```
class MyClass {  
    int x; // Private attribute  
    int y; // Private attribute  
};
```

- **C++ Encapsulation**

## Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as **private** (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public get and set methods.

## Access Private Members

To access a private attribute, use public "**get**" and "**set**" methods:

## Example

```
#include <iostream>  
using namespace std;  
  
class Employee {  
private:  
    int salary;  
  
public:  
    void setSalary(int s) {  
        salary = s;  
    }  
    int getSalary() {  
        return salary;  
    }  
};  
  
int main() {  
    Employee myObj;  
    myObj.setSalary(50000);  
    cout << myObj.getSalary();  
    return 0;  
}
```

# C++ Programming Language

50000

## Example explained

The salary attribute is private, which have restricted access.

The public **setSalary()** method takes a parameter (s) and assigns it to the salary attribute (salary = s).

The public **getSalary()** method returns the value of the private salary attribute.

Inside **main()**, we create an object of the Employee class. Now we can use the **setSalary()** method to set the value of the private attribute to 50000. Then we call the **getSalary()** method on the object to return the value.

## Why Encapsulation?

- It is considered good practice to declare your class attributes as private (as often as you can). Encapsulation ensures better control of your data, because you (or others) can change one part of the code without affecting other parts
- Increased security of data.

## • C++ Inheritance

In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **derived class** (child) - the class that inherits from another class
- **base class** (parent) - the class being inherited from

To inherit from a class, use the **:** symbol.

In the example below, the Car class (child) inherits the attributes and methods from the Vehicle class (parent):

## Example

# C++ Programming Language

```
#include <iostream>
#include <string>
using namespace std;

// Base class
class Vehicle {
public:
    string brand = "Ford";
    void honk() {
        cout << "Tuut, tuut! \n" ;
    }
};

// Derived class
class Car: public Vehicle {
public:
    string model = "Mustang";
};

int main() {
    Car myCar;
    myCar.honk();
    cout << myCar.brand + " " + myCar.model;
    return 0;
}
```

```
Tuut, tuut!
Ford Mustang
```

## Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

### • C++ Multilevel Inheritance

A **class** can also be derived from one class, which is already derived from another class.

In the following example, **MyGrandChild** is derived from class **MyChild** (which is derived from **MyClass**).

### Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

// Parent class
class MyClass {
public:
    void myFunction() {
        cout << "Some content in parent class." ;
    }
};

// Child class
class MyChild: public MyClass {
};

// Grandchild class
class MyGrandChild: public MyChild {
};

int main() {
    MyGrandChild myObj;
    myObj.myFunction();
    return 0;
}
```

Some content in parent class.

- **C++ Multiple Inheritance**

A **class** can also be derived from more than one base class, using a comma-separated list:

# C++ Programming Language

```
#include <iostream>
using namespace std;

// Base class
class MyClass {
public:
    void myFunction() {
        cout << "Some content in parent class.\n" ;
    }
};

// Another base class
class MyOtherClass {
public:
    void myOtherFunction() {
        cout << "Some content in another class.\n" ;
    }
};

// Derived class
class MyChildClass: public MyClass, public MyOtherClass {

int main() {
    MyChildClass myObj;
    myObj.myFunction();
    myObj.myOtherFunction();
    return 0;
}
}
```

```
Some content in parent class.
Some content in another class.
```

- ## C++ Inheritance Access

### Access Specifiers

You learned from the Access Specifiers chapter that there are three specifiers available in C++. Until now, we have only used public (members of a class are accessible from outside the class) and private (members can only be accessed within the class). The third specifier, protected, is similar to private, but it can also be accessed in the inherited class:

### Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

// Base class
class Employee {
protected: // Protected access specifier
    int salary;
};

// Derived class
class Programmer: public Employee {
public:
    int bonus;
    void setSalary(int s) {
        salary = s;
    }
    int getSalary() {
        return salary;
    }
};

int main() {
    Programmer myObj;
    myObj.setSalary(50000);
    myObj.bonus = 15000;
    cout << "Salary: " << myObj.getSalary() << "\n";
    cout << "Bonus: " << myObj.bonus << "\n";
    return 0;
}
```

```
Salary: 50000
Bonus: 15000
```

- ## C++ Polymorphism

### Polymorphism

**Polymorphism** means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways. For example, think of a base class called Animal that has a method called **animalSound()**. Derived classes of Animals

# C++ Programming Language

could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

## Example

```
// Base class
class Animal {
public:
    void animalSound() {
        cout << "The animal makes a sound \n" ;
    }
};

// Derived class
class Pig : public Animal {
public:
    void animalSound() {
        cout << "The pig says: wee wee \n" ;
    }
};

// Derived class
class Dog : public Animal {
public:
    void animalSound() {
        cout << "The dog says: bow wow \n" ;
    }
};
```

# C++ Programming Language

Now we can create **Pig** and **Dog** objects and override the **animalSound()** method:

## Example

```
#include <iostream>
#include <string>
using namespace std;

// Base class
class Animal {
public:
    void animalSound() {
        cout << "The animal makes a sound \n" ;
    }
};

// Derived class
class Pig : public Animal {
public:
    void animalSound() {
        cout << "The pig says: wee wee \n" ;
    }
};

// Derived class
class Dog : public Animal {
public:
    void animalSound() {
        cout << "The dog says: bow wow \n" ;
    }
};
```

# C++ Programming Language

```
int main() {
    Animal myAnimal;
    Pig myPig;
    Dog myDog;
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
    return 0;
}
```

```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
```

- ## C++ Files

The **fstream** library allows us to work with files.

To use the **fstream** library, include both the standard **<iostream>** AND the **<fstream>** header file:

### Example

```
#include <iostream>
#include <fstream>
```

There are three objects included in the **fstream** library, which are used to create, write or read files:

Object/Data Type	Description
ofstream	Creates and writes to files

# C++ Programming Language

**ifstream**      Reads from files

**fstream**      A combination of ofstream and ifstream: creates, reads, and writes to files

## Create and Write To a File

To create a file, use either the **ofstream** or **fstream** object, and specify the name of the file.

To write to the file, use the insertion operator (**<<**).

### Example

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    // Create and open a text file
    ofstream MyFile("filename.txt");

    // Write to the file
    MyFile << "Files can be tricky, but it is fun enough!";

    // Close the file
    MyFile.close();
}
```

### Why do we close the file?

It is considered good practice, and it can clean up unnecessary memory space.

# C++ Programming Language

## Read a File

To read from a file, use either the **ifstream** or **fstream** object, and the name of the file.

Note that we also use a while loop together with the **getline()** function (which belongs to the **ifstream** object) to read the file line by line, and to print the content of the file:

### Example

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    // Create a text file
    ofstream MyWriteFile("filename.txt");

    // Write to the file
    MyWriteFile << "Files can be tricky, but it is fun enough!";

    // Close the file
    MyWriteFile.close();

    // Create a text string, which is used to output the text file
    string myText;

    // Read from the text file
    ifstream MyReadFile("filename.txt");

    // Use a while loop together with the getline() function to read the file line by line
    while (getline (MyReadFile, myText)) {
        // Output the text from the file
        cout << myText;
    }

    // Close the file
    MyReadFile.close();
}
```

Files can be tricky, but it is fun enough!

## • C++ Exceptions

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

# C++ Programming Language

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (throw an error).

## C++ try and catch

Exception handling in C++ consist of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

### Example

```
try {  
    // Block of code to try  
    throw exception; // Throw an exception when a problem arise  
}  
  
catch () {  
    // Block of code to handle errors  
}
```

Consider the following example:

### Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main() {
    try {
        int age = 15;
        if (age > 18) {
            cout << "Access granted - you are old enough.";
        } else {
            throw (age);
        }
    }
    catch (int myNum) {
        cout << "Access denied - You must be at least 18 years old.\n";
        cout << "Age is: " << myNum;
    }
    return 0;
}
```

```
Access denied - You must be at least 18 years old.
```

```
Age is: 15
```

## Example explained

We use the try block to test some code: If the age variable is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error and do something about it. The catch statement takes a parameter: in our example we use an int variable (**myNum**) (because we are throwing an exception of int type in the try block (**age**)), to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped:

## Example

# C++ Programming Language

```
#include <iostream>
using namespace std;

int main() {
    try {
        int age = 20;
        if (age > 18) {
            cout << "Access granted - you are old enough.";
        } else {
            throw (age);
        }
    }
    catch (int myNum) {
        cout << "Access denied - You must be at least 18 years old.\n";
        cout << "Age is: " << myNum;
    }
    return 0;
}
```

```
Access granted - you are old enough.
```

You can also use the **throw** keyword to output a reference number, like a custom error number/code for organizing purposes:

## Example

```
#include <iostream>
using namespace std;

int main() {
    try {
        int age = 15;
        if (age > 18) {
            cout << "Access granted - you are old enough.";
        } else {
            throw 505;
        }
    }
    catch (int myNum) {
        cout << "Access denied - You must be at least 18 years old.\n";
        cout << "Error number: " << myNum;
    }
    return 0;
}
```

```
Access denied - You must be at least 18 years old.
Error number: 505
```

## Handle Any Type of Exceptions (...)

# C++ Programming Language

If you do not know the throw type used in the try block, you can use the "three dots" syntax (...) inside the catch block, which will handle any type of exception:

## Example

```
#include <iostream>
using namespace std;

int main() {
    try {
        int age = 15;
        if (age > 18) {
            cout << "Access granted - you are old enough.";
        } else {
            throw 505;
        }
    }
    catch (...) {
        cout << "Access denied - You must be at least 18 years old.\n";
    }
    return 0;
}
```

```
Access denied - You must be at least 18 years old.
```

# C# Programming Language

# C# Programming Language

## C# INTRODUCTION

### What is C#

C# is pronounced "C-Sharp".

It is an object-oriented programming language created by Microsoft that runs on the .NET Framework.

C# has roots from the C family, and the language is close to other popular languages like C++ and Java.

The first version was released in year 2002. The latest version, C# 8, was released in September 2019.

C# is used for:

- Mobile applications
- Desktop applications
- Web applications
- Web services
- Web sites
- Games
- VR
- Database applications
- And much, much more!

### Why Use C#?

- It is one of the most popular programming languages in the world
- It is easy to learn and simple to use
- It has a huge community support

# C# Programming Language

- C# is an object-oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs.
- As C# is close to C, C++ and Java, it makes it easy for programmers to switch to C# or vice versa

## • C# Get Started

### C# IDE

The easiest way to get started with C#, is to use an IDE.

An **IDE** (Integrated Development Environment) is used to edit and compile code.

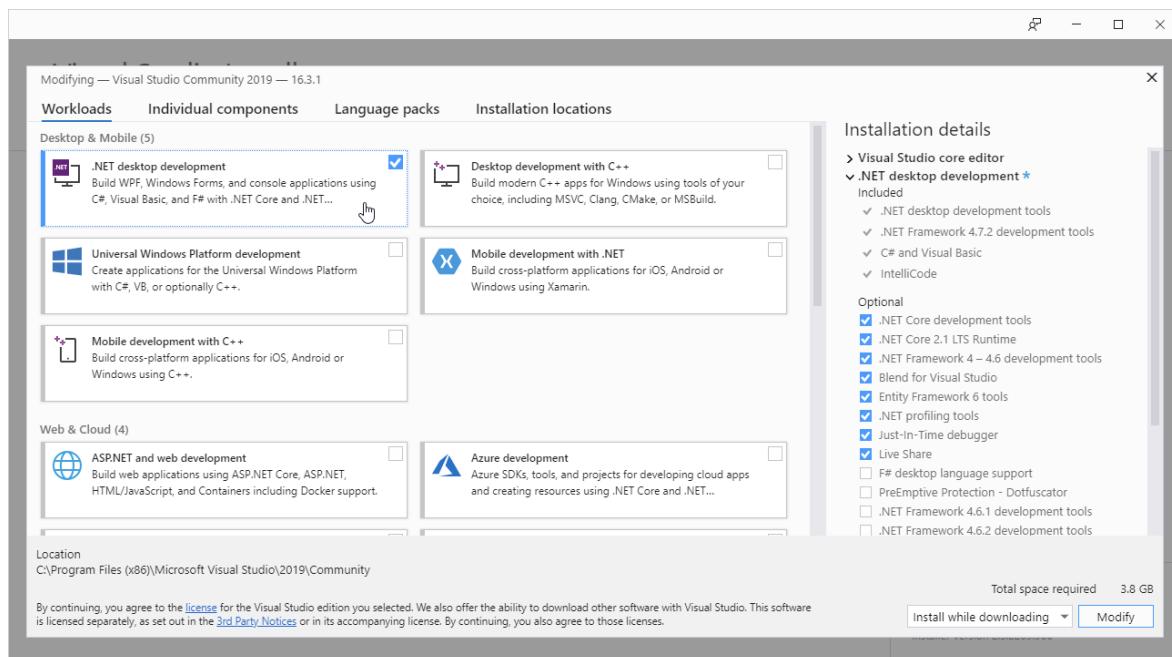
In our tutorial, we will use Visual Studio Community, which is free to download from

<https://visualstudio.microsoft.com/vs/community/>.

Applications written in **C#** use the **.NET Framework**, so it makes sense to use **Visual Studio**, as the program, the framework, and the language, are all created by Microsoft.

### C# Install

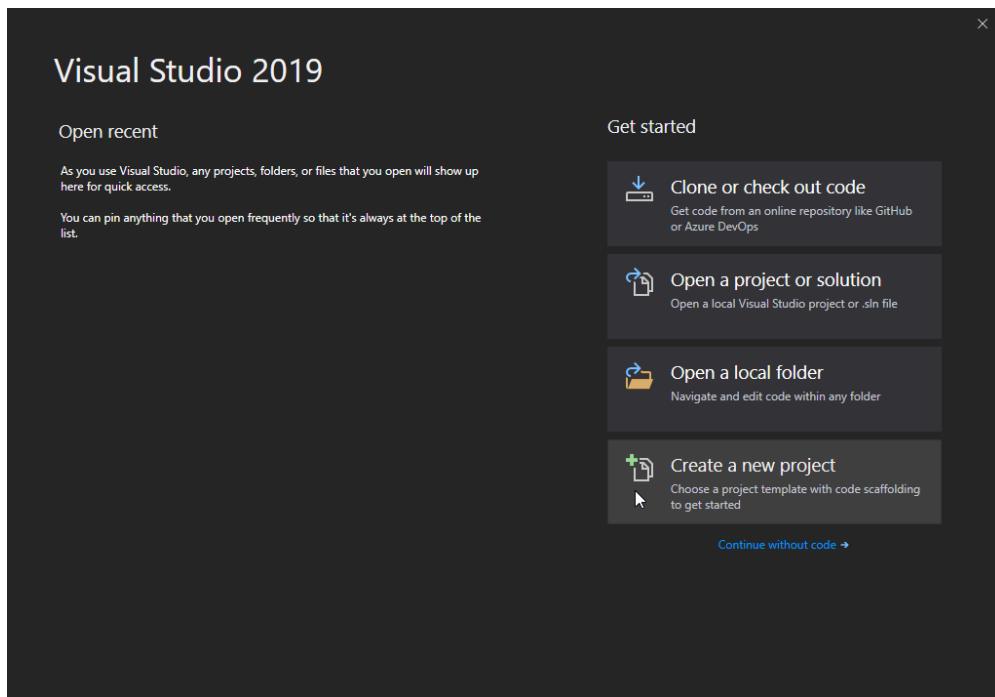
Once the Visual Studio Installer is downloaded and installed, choose the .NET workload and click on the **Modify/Install button**:



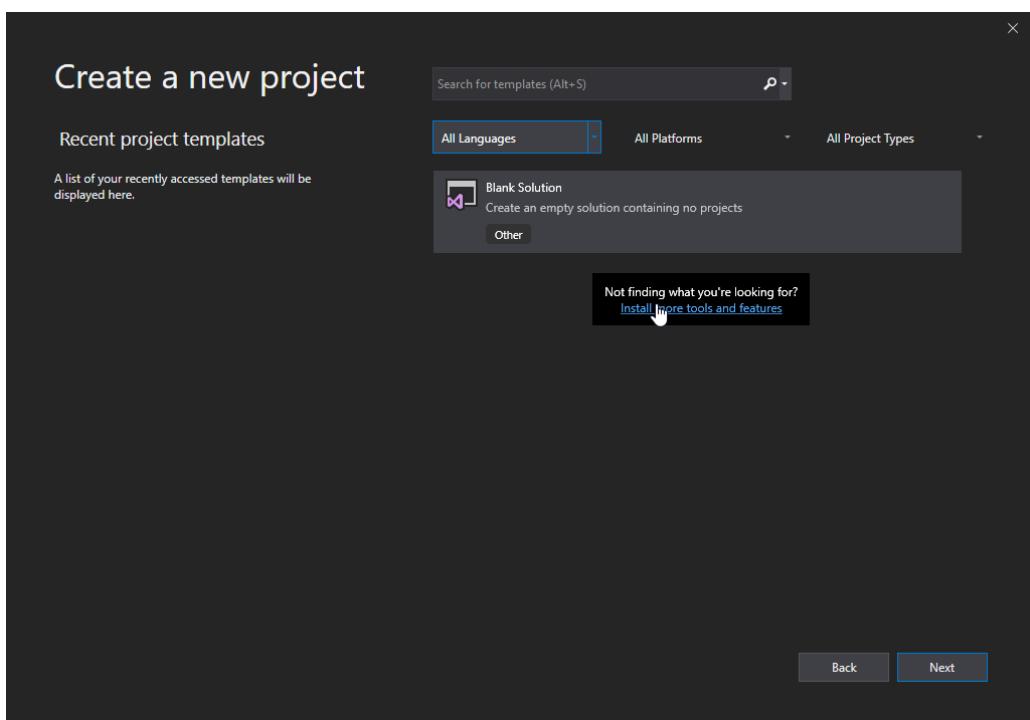
# C# Programming Language

After the installation is complete, click on the Launch button to get started with **Visual Studio**.

On the start window, choose Create a new project:

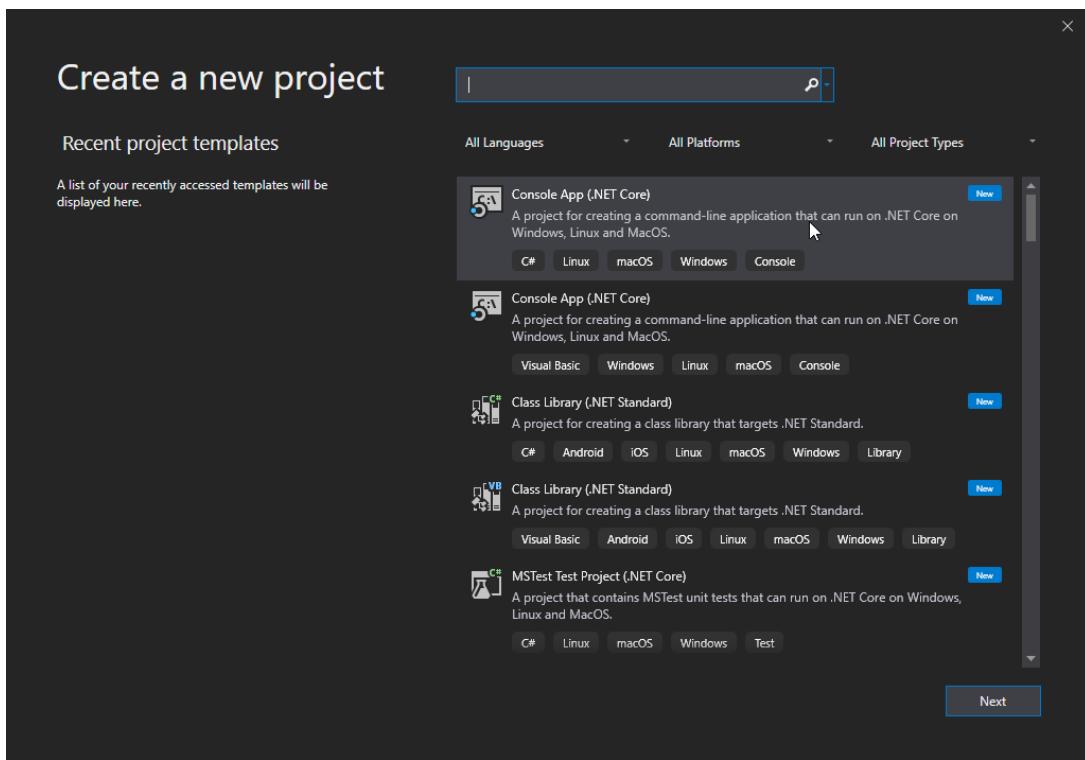


Then click on the "Install more tools and features" button:

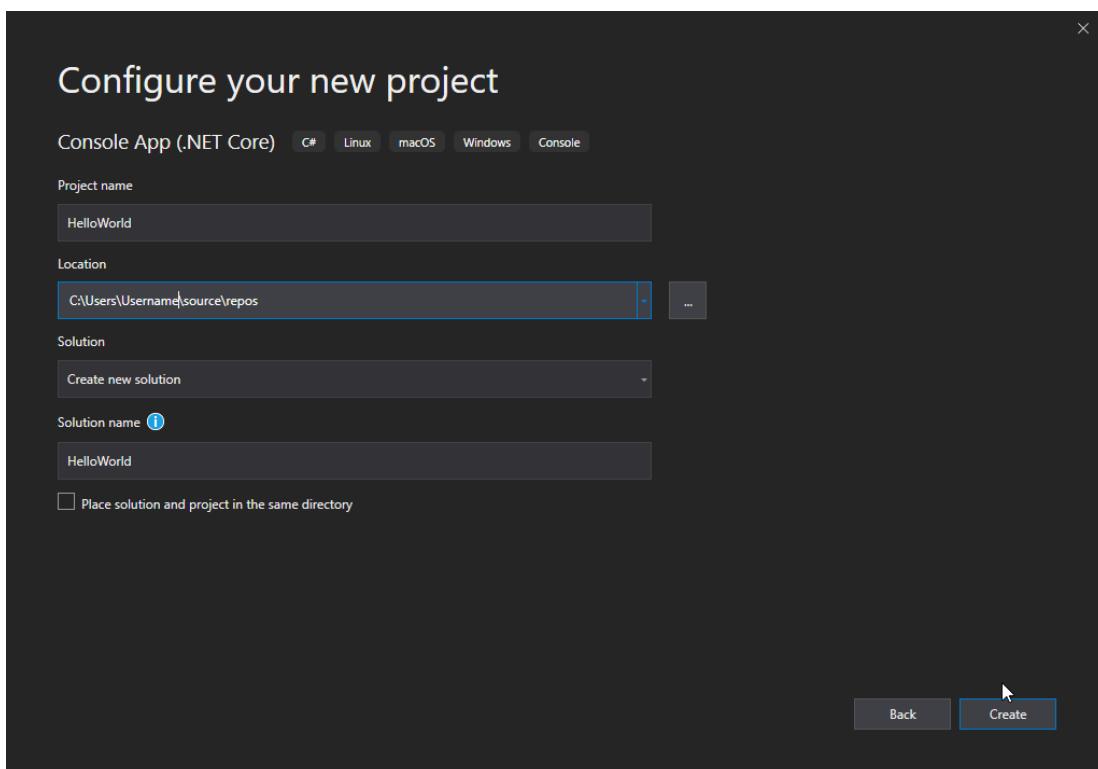


# C# Programming Language

Choose "Console App (.NET Core)" from the list and click on the Next button:

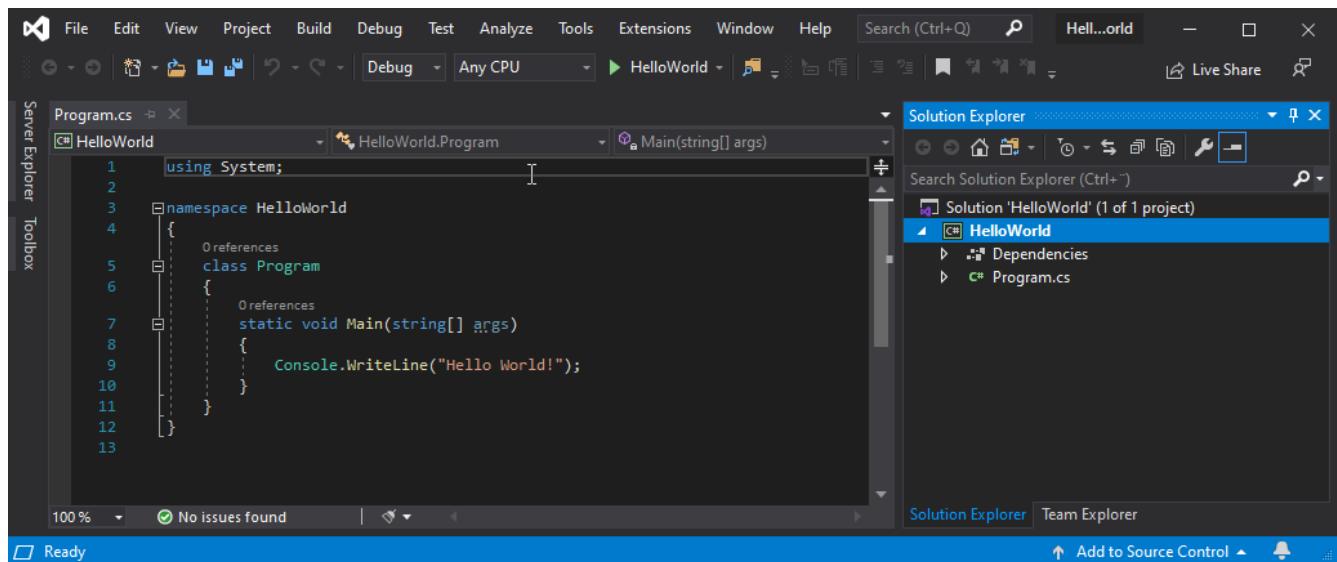


Enter a name for your project, and click on the Create button:



# C# Programming Language

Visual Studio will automatically generate some code for your project:



## C# Syntax

the following code to print "Hello World" to the screen:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Hello World!

## Example explained

**Line 1:** using System means that we can use classes from the System namespace.

**Line 2:** A blank line. C# ignores white space. However, multiple lines makes the code more readable.

# C# Programming Language

**Line 3:** namespace is used to organize your code, and it is a container for classes and other namespaces.

**Line 4:** The curly braces {} marks the beginning and the end of a block of code.

**Line 5:** class is a container for data and methods, which brings functionality to your program. Every line of code that runs in C# must be inside a class. In our example, we named the class Program.

**Line 7:** Another thing that always appear in a C# program, is the Main method. Any code inside its curly brackets {} will be executed. You don't have to understand the keywords before and after Main. You will get to know them bit by bit while reading this tutorial.

**Line 9:** Console is a class of the System namespace, which has a **WriteLine()** method that is used to output/print text. In our example it will output "Hello World!".

If you omit the using System line, you would have to write

**System.Console.WriteLine()** to print/output text.

**Note:** Every C# statement ends with a semicolon ;

**Note:** C# is case-sensitive: "MyClass" and "myclass" has different meaning.

```
Hello World!
```

## WriteLine or Write

The most common method to output something in C# is **WriteLine()**, but you can also use **Write()**.

The difference is that **WriteLine()** prints the output on a new line each time, while **Write()** prints on the same line (note that you should remember to add spaces when needed, for better readability):

## Example

# C# Programming Language

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.WriteLine("I will print on a new line.");

            Console.Write("Hello World! ");
            Console.Write("I will print on the same line.");
        }
    }
}
```

```
Hello World!
I will print on a new line.
Hello World! I will print on the same line.
```

## • C# Comments

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

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

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

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

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            // This is a comment
            Console.WriteLine("Hello World!");
        }
    }
}
```

# C# Programming Language

Hello World!

This **example** uses a single-line comment at the end of a line of code:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!"); // This is a comment
        }
    }
}
```

Hello World!

## C# Multi-line Comments

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

Any text between `/*` and `*/` will be ignored by C#.

This example uses a **multi-line comment** (a comment block) to explain the code:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            /* The code below will print the words Hello World
               to the screen, and it is amazing */
            Console.WriteLine("Hello World!");
        }
    }
}
```

# C# Programming Language

Hello World!

## *Single or multi-line comments?*

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

## • 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
- **double** - 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
- **string** - stores text, such as "Hello World". String values are surrounded by double quotes
- **bool** - stores values with two states: true or false

## Declaring (Creating) Variables

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

### Syntax

**type variableName = value;**

Where **type** is a **C#** type (such as **int** or **string**), and **variableName** is the name of the variable (such as **x** or **name**). The equal sign is used to assign values to the variable.

To create a variable that should store text, look at the following example:

### Example

Create a variable called **name** of type **string** and assign it the value "John":

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string name = "John";
            Console.WriteLine(name);
        }
    }
}
```

John

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 it the value 15:

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myNum = 15;
            Console.WriteLine(myNum);
        }
    }
}
```

15

## Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myNum = 15;
            Console.WriteLine(myNum);
        }
    }
}
```

15

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

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myNum;
            myNum = 15;
            Console.WriteLine(myNum);
        }
    }
}
```

15

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

# C# Programming Language

## Example

Change the value of **myNum** to 20:

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myNum = 15;
            myNum = 20;
            Console.WriteLine(myNum);
        }
    }
}
```

```
20
```

## Constants

However, you can add the **const** keyword if you don't want others (or yourself) to overwrite existing values (this will declare the variable as "constant", which means unchangeable and read-only):

## Example

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

```
myNum = 20; // error
```

The **const** keyword is useful when you want a variable to always store the same value, so that others (or yourself) won't mess up your code. An example that is often referred to as a constant, is PI (3.14159...).

**Note:** You cannot declare a constant variable without assigning the value. If you do, an error will occur: A const field requires a value to be provided.

## Other Types

---

# C# Programming Language

A demonstration of how to declare variables of other types:

## Example

```
int myNum = 5;
double myDoubleNum = 5.99D;
char myLetter = 'D';
bool myBool = true;
string myText = "Hello";
```

## Display Variables

The **WriteLine()** method is often used to display variable values to the console window.

To combine both **text** and a **variable**, use the **+** character:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string name = "John";
            Console.WriteLine("Hello " + name);
        }
    }
}
```

```
Hello John
```

You can also use the **+** character to add a variable to another variable:

## Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string firstName = "John ";
            string lastName = "Doe";
            string fullName = firstName + lastName;
            Console.WriteLine(fullName);
        }
    }
}
```

John Doe

For numeric values, the **+** character works as a mathematical operator (notice that we use int (integer) variables here):

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            int y = 6;
            Console.WriteLine(x + y);
        }
    }
}
```

11

From the example above, you can expect:

# C# Programming Language

- x stores the value 5
- y stores the value 6
- Then we use the **WriteLine()** method to display the value of x + y, which is 11

## Declare Many Variables

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

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5, y = 6, z = 50;
            Console.WriteLine(x + y + z);
        }
    }
}
```

61

## C# Identifiers

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:

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // Good
            int minutesPerHour = 60;

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

            Console.WriteLine(minutesPerHour);
            Console.WriteLine(m);
        }
    }
}
```

```
60
60
```

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits and the underscore character (`_`)
- Names must begin with a letter
- Names should start with a lowercase letter and it cannot contain whitespace
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like C# keywords, such as `int` or `double`) cannot be used as names
- **C# Data Types**

A **data type** specifies the **size** and **type** of variable values. It is important to use the correct data type for the corresponding variable; to avoid errors, to **save time** and **memory**, but it will also make your code more maintainable and readable. The most common data types are:

# C# Programming Language

Data Type	Size	Description
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
bool	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter, surrounded by single quotes
string	2 bytes per character	Stores a sequence of characters, surrounded by double quotes

## Numbers

**Number types** are divided into two groups:

# C# Programming Language

**Integer types** stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are **int** and **long**. Which type you should use, depends on the numeric value.

**Floating point** types represents numbers with a fractional part, containing one or more decimals. Valid types are **float** and **double**.

## Integer Types

### Int

The **int** data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the int data type is the preferred data type when we create variables with a numeric value.

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myNum = 100000;
            Console.WriteLine(myNum);
        }
    }
}
```

```
100000
```

### Long

The **long data** type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

# C# Programming Language

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            long myNum = 15000000000L;
            Console.WriteLine(myNum);
        }
    }
}
```

```
15000000000
```

## Floating Point Types

You should use a **floating point** type whenever you need a number with a decimal, such as 9.99 or 3.14515.

### Float

The float data type can store fractional numbers from 3.4e-038 to 3.4e+038. Note that you should end the value with an "F":

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            float myNum = 5.75F;
            Console.WriteLine(myNum);
        }
    }
}
```

# C# Programming Language

5.75

## Double

The double data type can store fractional numbers from 1.7e-308 to 1.7e+308. Note that you can end the value with a "D" (although not required):

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            double myNum = 19.99D;
            Console.WriteLine(myNum);
        }
    }
}
```

19.99

## Use float or double?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of **float** is only six or seven decimal digits, while **double** variables have a precision of about 15 digits. Therefore it is safer to use **double** for most calculations.

## Scientific Numbers

A **floating point** number can also be a scientific number with an "e" to indicate the power of 10:

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            float f1 = 35e3F;
            double d1 = 12E4D;
            Console.WriteLine(f1);
            Console.WriteLine(d1);
        }
    }
}
```

```
35000
120000
```

## Booleans

A **boolean** data type is declared with the `bool` keyword and can only take the values true or false:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            bool isCSharpFun = true;
            bool isFishTasty = false;
            Console.WriteLine(isCSharpFun); // Outputs True
            Console.WriteLine(isFishTasty); // Outputs False
        }
    }
}
```

# C# Programming Language

True

False

## Characters

The **char** data type is used to store a single character. The character must be surrounded by single quotes, like 'A' or 'c':

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            char myGrade = 'B';
            Console.WriteLine(myGrade);
        }
    }
}
```

B

## Strings

The **string** data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string greeting = "Hello World";
            Console.WriteLine(greeting);
        }
    }
}
```

# C# Programming Language

Hello World

## • C# Type Casting

Type casting is when you assign a value of one data type to another type.

In C#, there are two types of casting:

- **Implicit Casting** (automatically) - converting a smaller type to a larger type size  
**char -> int -> long -> float -> double**
- **Explicit Casting** (manually) - converting a larger type to a smaller size type  
**double -> float -> long -> int -> char**

## Implicit Casting

Implicit casting is done automatically when passing a smaller size type to a larger size type:

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myInt = 9;
            double myDouble = myInt; // Automatic casting: int to double

            Console.WriteLine(myInt);
            Console.WriteLine(myDouble);
        }
    }
}
```

9  
9

## Explicit Casting

# C# Programming Language

Explicit casting must be done manually by placing the type in parentheses in front of the value:

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            double myDouble = 9.78;
            int myInt = (int) myDouble; // Manual casting: double to int

            Console.WriteLine(myDouble);
            Console.WriteLine(myInt);
        }
    }
}
```

```
9.78
9
```

## Type Conversion Methods

It is also possible to convert data types explicitly by using built-in methods, such as **Convert.ToBoolean**, **Convert.ToDouble**, **Convert.ToString**, **Convert.ToInt32** (int) and **Convert.ToInt64** (long):

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int myInt = 10;
            double myDouble = 5.25;
            bool myBool = true;

            Console.WriteLine(Convert.ToString(myInt));      // Convert int to string
            Console.WriteLine(Convert.ToDouble(myInt));       // Convert int to double
            Console.WriteLine(Convert.ToInt32(myDouble));     // Convert double to int
            Console.WriteLine(Convert.ToString(myBool));      // Convert bool to string
        }
    }
}
```

```
10
10
5
True
```

- ## C# User Input

### Get User Input

You have already learned that **Console.WriteLine()** is used to output (print) values. Now we will use **Console.ReadLine()** to get user input.

In the following example, the user can input his or hers username, which is stored in the variable **userName**. Then we print the value of **userName**:

#### Example

```
using System;

namespace MyApplication
{
    class Program
    {
```

# C# Programming Language

```
static void Main(string[] args)
{
    // Type your username and press enter
    Console.WriteLine("Enter username:");

    // Create a string variable and get user input from the
    // keyboard and store it in the variable
    string userName = Console.ReadLine();

    // Print the value of the variable (userName), which will
    // display the input value
    Console.WriteLine("Username is: " + userName);
}
}
```

Enter username:  
elaf  
Username is: elaf

## User Input and Numbers

The **Console.ReadLine()** method returns a string. Therefore, you cannot get information from another data type, such as int. The following program will cause an error:

### Example

```
Console.WriteLine("Enter your age:");
int age = Console.ReadLine();
Console.WriteLine("Your age is: " + age);
```

The error message will be something like this:

```
Cannot implicitly convert type 'string' to 'int'
```

Like the error message says, you cannot implicitly convert type '**string**' to '**int**'.  
you can convert any type explicitly, by using one of the **Convert.To** methods:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter your age:");
            int age = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Your age is: " + age);
        }
    }
}
```

```
Enter your age:
```

```
20
```

```
Your age is: 20
```

- ## 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

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 100 + 50;
            Console.WriteLine(x);
        }
    }
}
```

```
150
```

# C# Programming Language

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

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int sum1 = 100 + 50;      // 150 (100 + 50)
            int sum2 = sum1 + 250;    // 400 (150 + 250)
            int sum3 = sum2 + sum1;  // 800 (400 + 400)
            Console.WriteLine(sum1);
            Console.WriteLine(sum2);
            Console.WriteLine(sum3);
        }
    }
}
```

```
150
400
800
```

## Arithmetic Operators

**Arithmetic operators** are used to perform common mathematical operations:

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

# C# Programming Language

## C# 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

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10;
            Console.WriteLine(x);
        }
    }
}
```

10

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

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10;
            x += 5;
            Console.WriteLine(x);
        }
    }
}
```

# C# Programming Language

15

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

## C# Comparison Operators

**Comparison operators** are used to compare two values:

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

# C# Programming Language

## C# 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)

## • C# Math

The **C#** Math class has many methods that allows you to perform mathematical tasks on numbers.

### Math.Max(x,y)

The **Math.Max(x,y)** method can be used to find the highest value of x and y:

#### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Math.Max(5, 10));
        }
    }
}
```

10

### Math.Min(x,y)

The **Math.Min(x,y)** method can be used to find the lowest value of of x and y:

#### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Math.Min(5, 10));
        }
    }
}
```

5

## Math.Sqrt(x)

The **Math.Sqrt(x)** method returns the square root of x:

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Math.Sqrt(64));
        }
    }
}
```

8

## Math.Abs(x)

The **Math.Abs(x)** method returns the absolute (positive) value of x:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Math.Abs(-4.7));
        }
    }
}
```

4.7

## Math.Round()

**Math.Round()** rounds a number to the nearest whole number:

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Math.Round(9.99));
        }
    }
}
```

10

- **C# Strings**

**Strings** are used for storing text.

A **string variable** contains a collection of characters surrounded by **double quotes**:

# C# Programming Language

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string greeting = "Hello";
            Console.WriteLine(greeting);
        }
    }
}
```

```
Hello
```

## String Length

A **string** in C# is actually an object, which contain properties and methods that can perform certain operations on strings. For example, the length of a string can be found with the Length property:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string txt = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
            Console.WriteLine("The length of the txt string is: " + txt.Length);
        }
    }
}
```

# C# Programming Language

The length of the txt string is: 26

## Other Methods

There are many string methods available, for example **ToUpper()** and **ToLower()**, which returns a copy of the string converted to uppercase or lowercase:

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string txt = "Hello World";
            Console.WriteLine(txt.ToUpper()); // Outputs "HELLO WORLD"
            Console.WriteLine(txt.ToLower()); // Outputs "hello world"
        }
    }
}
```

HELLO WORLD  
hello world

## String Concatenation

The **+** operator can be used between strings to combine them. This is called **concatenation**:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string firstName = "John ";
            string lastName = "Doe";
            string name = firstName + lastName;
            Console.WriteLine(name);
        }
    }
}
```

John Doe

You can also use the **string.Concat()** method to concatenate two strings:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string firstName = "John ";
            string lastName = "Doe";
            string name = string.Concat(firstName, lastName);
            Console.WriteLine(name);
        }
    }
}
```

John Doe

## String Interpolation

# C# Programming Language

Another option of string concatenation, is string interpolation, which substitutes values of variables into placeholders in a string. Note that you do not have to worry about spaces, like with concatenation:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string firstName = "John";
            string lastName = "Doe";
            string name = $"My full name is: {firstName} {lastName}";
            Console.WriteLine(name);
        }
    }
}
```

```
My full name is: John Doe
```

Also note that you have to use the dollar sign (\$) when using the string interpolation method.

## Access Strings

You can access the characters in a string by referring to its index number inside square brackets [ ].

This **example** prints the first character in **myString**:

## Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string myString = "Hello";
            Console.WriteLine(myString[0]);
        }
    }
}
```

H

**Note:** String indexes start with 0: [0] is the first character. [1] is the second character, etc.

This **example** prints the second character (1) in **myString**:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string myString = "Hello";
            Console.WriteLine(myString[1]);
        }
    }
}
```

e

You can also find the index position of a specific character in a string, by using the **IndexOf()** method:

# C# Programming Language

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string myString = "Hello";
            Console.WriteLine(myString.IndexOf("e"));
        }
    }
}
```

1

Another useful method is **Substring()**, which extracts the characters from a string, starting from the specified character position/index, and returns a new string. This method is often used together with **IndexOf()** to get the specific character position:

## Example

```
using System;

namespace GetLastName
{
    class Program
    {
        static void Main()
        {
            // Full name
            string name = "John Doe";

            // Location of the letter D
            int charPos = name.IndexOf("D");

            // Get last name
            string lastName = name.Substring(charPos);

            // Print the result
            Console.WriteLine(lastName);
        }
    }
}
```

# C# Programming Language

Doe

## Special Characters

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

```
string 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

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string txt = "We are the so-called \"Vikings\\\" from the north.";
            Console.WriteLine(txt);
        }
    }
}
```

We are the so-called "Vikings" from the north.

# C# Programming Language

The sequence \ inserts a single quote in a string:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string txt = "It\'s alright.";
            Console.WriteLine(txt);
        }
    }
}
```

It's alright.

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

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string txt = "The character \\ is called backslash.";
            Console.WriteLine(txt);
        }
    }
}
```

The character \ is called backslash.

Other useful escape characters in C# are:

# C# Programming Language

Code	Result
\n	New Line
\t	Tab
\b	Backspace

## Adding Numbers and Strings

### WARNING!

C# uses the + operator for both addition and concatenation.

**Remember:** Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10;
            int y = 20;
            int z = x + y;
            Console.WriteLine(z);
        }
    }
}
```

30

If you add two strings, the result will be a string concatenation:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string x = "10";
            string y = "20";
            string z = x + y;
            Console.WriteLine(z);
        }
    }
}
```

1020

## • C# Booleans

Very often, in programming, you 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 can take the values true or false.

## Boolean Values

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

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            bool isCSharpFun = true;
            bool isFishTasty = false;
            Console.WriteLine(isCSharpFun); // Outputs True
            Console.WriteLine(isFishTasty); // Outputs False
        }
    }
}
```

```
True  
False
```

## Boolean Expression

A **Boolean expression** is a C# expression that returns a Boolean value: True or False.

You can use a comparison operator, such as the greater than (>) operator to find out if an expression (or a variable) is true:

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10;
            int y = 9;
            Console.WriteLine(x > y); // returns True, because 10 is higher than 9
        }
    }
}
```

```
True
```

Or even easier:

# C# Programming Language

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(10 > 9); // returns True, because 10 is higher than 9
        }
    }
}
```

True

In the examples below, we use the equal to (==) operator to evaluate an expression:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 10;
            Console.WriteLine(x == 10); // returns True, because the value of x is equal to 10
        }
    }
}
```

True

## Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(10 == 15); // returns False, because 10 is not equal to 15
        }
    }
}
```

False

## • C# If ... Else

### C# Conditions and If Statements

C# supports the usual logical conditions from mathematics:

- Less than:  $a < b$
- Less than or equal to:  $a \leq b$
- Greater than:  $a > b$
- Greater than or equal to:  $a \geq b$
- 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

# C# Programming Language

## 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

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            if (20 > 18)
            {
                Console.WriteLine("20 is greater than 18");
            }
        }
    }
}
```

```
20 is greater than 18
```

We can also test variables:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 20;
            int y = 18;
            if (x > y)
            {
                Console.WriteLine("x is greater than y");
            }
        }
    }
}
```

```
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
}
```

# C# Programming Language

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int time = 20;
            if (time < 18)
            {
                Console.WriteLine("Good day.");
            }
            else
            {
                Console.WriteLine("Good evening.");
            }
        }
    }
}
```

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

# C# Programming Language

```
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

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int time = 22;
            if (time < 10)
            {
                Console.WriteLine("Good morning.");
            }
            else if (time < 20)
            {
                Console.WriteLine("Good day.");
            }
            else
            {
                Console.WriteLine("Good evening.");
            }
        }
    }
}
```

Good evening.

# C# Programming Language

## 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."

## 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

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int time = 20;
            if (time < 18)
            {
                Console.WriteLine("Good day.");
            }
            else
            {
                Console.WriteLine("Good evening.");
            }
        }
    }
}
```

# C# Programming Language

Good evening.

You can simply write:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int time = 20;
            string result = (time < 18) ? "Good day." : "Good evening.";
            Console.WriteLine(result);
        }
    }
}
```

Good evening.

- ## C# Switch

### C# Switch Statements

Use the switch statement to select one of many code blocks to be executed.

#### Syntax

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

# C# Programming Language

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** and **default** keywords will be described later in this chapter

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

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int day = 4;
            switch (day)
            {
                case 1:
                    Console.WriteLine("Monday");
                    break;
                case 2:
                    Console.WriteLine("Tuesday");
                    break;
                case 3:
                    Console.WriteLine("Wednesday");
                    break;
                case 4:
                    Console.WriteLine("Thursday");
                    break;
            }
        }
    }
}
```

# C# Programming Language

```
case 5:  
    Console.WriteLine("Friday");  
    break;  
case 6:  
    Console.WriteLine("Saturday");  
    break;  
case 7:  
    Console.WriteLine("Sunday");  
    break;  
}  
}  
}  
}
```

Thursday

## 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 is optional and specifies some code to run if there is no case match:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int day = 4;
            switch (day)
            {
                case 6:
                    Console.WriteLine("Today is Saturday.");
                    break;
                case 7:
                    Console.WriteLine("Today is Sunday.");
                    break;
                default:
                    Console.WriteLine("Looking forward to the Weekend.");
                    break;
            }
        }
    }
}
```

Looking forward to the Weekend.

- ## C# While Loop

### 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.

### C# While Loop

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

#### Syntax

# C# Programming Language

```
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

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            while (i < 5)
            {
                Console.WriteLine(i);
                i++;
            }
        }
    }
}
```

```
0
1
2
3
4
```

## 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

---

# C# Programming Language

```
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

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            do
            {
                Console.WriteLine(i);
                i++;
            }
            while (i < 5);
        }
    }
}
```

```
0
1
2
3
4
```

- **C# 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:

# C# Programming Language

## 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

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

```
0  
1  
2  
3  
4
```

Example explained

**Statement 1** sets a variable before the loop starts (int i = 0).

# C# Programming Language

**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:

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i <= 10; i = i + 2)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

```
0
2
4
6
8
10
```

## • The foreach Loop

There is also a foreach loop, which is used exclusively to loop through elements in an array:

### Syntax

# C# Programming Language

```
foreach (type variableName in arrayName)
{
    // code block to be executed
}
```

The following example outputs all elements in the cars array, using a **foreach** loop:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
            foreach (string i in cars)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

```
Volvo
BMW
Ford
Mazda
```

- **C# Break and Continue**

## C# 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:

# C# Programming Language

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 10; i++)
            {
                if (i == 4)
                {
                    break;
                }
                Console.WriteLine(i);
            }
        }
    }
}
```

```
0
1
2
3
```

## C# 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

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 10; i++)
            {
                if (i == 4)
                {
                    continue;
                }
                Console.WriteLine(i);
            }
        }
    }
}
```

```
0  
1  
2  
3  
5  
6  
7  
8  
9
```

## Break and Continue in While Loop

You can also use break and continue in while loops:

### Break Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            while (i < 10)
            {
                Console.WriteLine(i);
                i++;
                if (i == 4)
                {
                    break;
                }
            }
        }
    }
}
```

```
0  
1  
2  
3
```

## Continue Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            while (i < 10)
            {
                if (i == 4)
                {
                    i++;
                    continue;
                }
                Console.WriteLine(i);
                i++;
            }
        }
    }
}
```

# C# Programming Language

```
0  
1  
2  
3  
5  
6  
7  
8  
9
```

- ## C# Arrays

### Create an Array

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

To declare an array, define the variable type with square brackets:

```
string[] cars;
```

We have now declared a variable that holds an array of strings.

To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

### Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

#### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
            Console.WriteLine(cars[0]);
        }
    }
}
```

```
Volvo
```

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

## Change an Array Element

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

### Example

```
cars[0] = "Opel";
```

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
            cars[0] = "Opel";
            Console.WriteLine(cars[0]);
        }
    }
}
```

# C# Programming Language

Opel

## Array Length

To find out how many elements an array has, use the Length property:

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
            Console.WriteLine(cars.Length);
        }
    }
}
```

4

## Loop Through an Array

You can loop through the array elements with the for loop, and use the Length property to specify how many times the loop should run.

The following example outputs all elements in the cars array:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
            for (int i = 0; i < cars.Length; i++)
            {
                Console.WriteLine(cars[i]);
            }
        }
    }
}
```

```
Volvo
BMW
Ford
Mazda
```

## The foreach Loop

There is also a foreach loop, which is used exclusively to loop through elements in an array:

### Syntax

```
foreach (type variableName in arrayName)
{
    // code block to be executed
}
```

The following example outputs all elements in the cars array, using a foreach loop:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
            foreach (string i in cars)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

```
Volvo
BMW
Ford
Mazda
```

The example above can be read like this: for each string element (called **i** - as in index) in cars, print out the value of **i**.

If you compare the for loop and foreach loop, you will see that the foreach method is easier to write, it does not require a counter (using the Length property), and it is more readable.

## Sort Arrays

There are many array methods available, for example **Sort()**, which sorts an array alphabetically or in an ascending order:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // Sort a string
            string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
            Array.Sort(cars);
            foreach (string i in cars)
            {
                Console.WriteLine(i);
            }

            // Sort an int
            int[] myNumbers = {5, 1, 8, 9};
            Array.Sort(myNumbers);
            foreach (int i in myNumbers)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

```
BMW
Ford
Mazda
Volvo
1
5
8
9
```

## System.Linq Namespace

Other useful array methods, such as Min, Max, and Sum, can be found in the **System.Linq namespace**:

# C# Programming Language

## Example

```
using System;
using System.Linq;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] myNumbers = {5, 1, 8, 9};
            Console.WriteLine(myNumbers.Max()); // largest value
            Console.WriteLine(myNumbers.Min()); // smallest value
            Console.WriteLine(myNumbers.Sum()); // sum of myNumbers
        }
    }
}
```

```
9
1
23
```

## Other Ways to Create an Array

If you are familiar with C#, you might have seen arrays created with the new keyword, and perhaps you have seen arrays with a specified size as well. In C#, there are different ways to create an array:

```
// Create an array of four elements, and add values later
string[] cars = new string[4];

// Create an array of four elements and add values right away
string[] cars = new string[4] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements without specifying the size
string[] cars = new string[] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements, omitting the new keyword, and without specifying the size
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

# C# Programming Language

It is up to you which option you choose. In our tutorial, we will often use the last option, as it is faster and easier to read.

However, you should note that if you declare an array and initialize it later, you have to use the new keyword:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare an array
            string[] cars;

            // Add values, using new
            cars = new string[] {"Volvo", "BMW", "Ford"};

            // This would cause an error: cars = {"Volvo", "BMW", "Ford"};

            Console.WriteLine(cars[0]);
        }
    }
}
```

Volvo

# C# Programming Language

## C# METHODS

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

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

Methods are used to perform certain actions, and they are also known as functions.

Why use methods? To reuse code: define the code once, and use it many times.

### Create a Method

A method is defined with the name of the method, followed by **parentheses ()**. C# provides some pre-defined methods, which you already are familiar with, such as **Main()**, but you can also create your own methods to perform certain actions:

### Example

Create a method inside the Program class:

```
class Program
{
    static void MyMethod()
    {
        // code to be executed
    }
}
```

### Example Explained

- **MyMethod()** is the name of the method
- **static** means that the method belongs to the Program class and not an object of the Program class.
- **void** means that this method does not have a return value.

**Note:** In C#, it is good practice to start with an uppercase letter when naming methods, as it makes the code easier to read.

# C# Programming Language

## Call a Method

To call (execute) a method, write the method's name followed by two parentheses () and a semicolon;

In the following example, **MyMethod()** is used to print a text (the action), when it is called:

### Example

Inside **Main()**, call the **myMethod()** method:

```
using System;

namespace MyApplication
{
    class Program
    {
        static void MyMethod()
        {
            Console.WriteLine("I just got executed!");
        }

        static void Main(string[] args)
        {
            MyMethod();
        }
    }
}
```

I just got executed!

A **method** can be called multiple times:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void MyMethod()
        {
            Console.WriteLine("I just got executed!");
        }

        static void Main(string[] args)
        {
            MyMethod();
            MyMethod();
            MyMethod();
        }
    }
}
```

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

## • C# Method Parameters

### Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

They are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a string called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void MyMethod(string fname)
        {
            Console.WriteLine(fname + " Refsnes");
        }

        static void Main(string[] args)
        {
            MyMethod("Liam");
            MyMethod("Jenny");
            MyMethod("Anja");
        }
    }
}
```

```
Liam Refsnes
Jenny Refsnes
Anja Refsnes
```

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

## Default Parameter Value

You can also use a default parameter value, by using the equals sign (=). If we call the method without an argument, it uses the default value ("Norway"):

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void MyMethod(string country = "Norway")
        {
            Console.WriteLine(country);
        }

        static void Main(string[] args)
        {
            MyMethod("Sweden");
            MyMethod("India");
            MyMethod();
            MyMethod("USA");
        }
    }
}
```

```
Sweden
India
Norway
USA
```

A parameter with a default value, is often known as an "**optional parameter**". From the example above, **country** is an optional parameter and "**Norway**" is the default value.

## Multiple Parameters

You can have as many parameters as you like:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void MyMethod(string fname, int age)
        {
            Console.WriteLine(fname + " is " + age);
        }

        static void Main(string[] args)
        {
            MyMethod("Liam", 5);
            MyMethod("Jenny", 8);
            MyMethod("Anja", 31);
        }
    }
}
```

```
Liam is 5
Jenny is 8
Anja is 31
```

Note that when you are working with multiple parameters, the method 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 examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as int or double) instead of void, and use the return keyword inside the method:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static int MyMethod(int x)
        {
            return 5 + x;
        }

        static void Main(string[] args)
        {
            Console.WriteLine(MyMethod(3));
        }
    }
}
```

8

This example returns the sum of a method's two parameters:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static int MyMethod(int x)
        {
            return 5 + x;
        }

        static void Main(string[] args)
        {
            Console.WriteLine(MyMethod(3));
        }
    }
}
```

# C# Programming Language

8

You can also store the result in a variable (recommended, as it is easier to read and maintain):

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static int MyMethod(int x, int y)
        {
            return x + y;
        }

        static void Main(string[] args)
        {
            int z = MyMethod(5, 3);
            Console.WriteLine(z);
        }
    }
}
```

8

## Named Arguments

It is also possible to send arguments with the key: value syntax.

That way, the order of the arguments does not matter:

## Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void MyMethod(string child1, string child2, string child3)
        {
            Console.WriteLine("The youngest child is: " + child3);
        }

        static void Main(string[] args)
        {
            MyMethod(child3: "John", child1: "Liam", child2: "Liam");
        }
    }
}
```

The youngest child is: John

Named arguments are especially useful when you have multiple parameters with default values, and you only want to specify one of them when you call it:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void MyMethod(string child1 = "Liam", string child2 = "Jenny", string child3 = "John")
        {
            Console.WriteLine(child3);
        }

        static void Main(string[] args)
        {
            MyMethod("child3");
        }
    }
}
```

# C# Programming Language

```
}
```

```
John
```

- **C# Method Overloading**

Method Overloading

With method overloading, multiple methods can have the same name with different parameters:

### Example

```
int MyMethod(int x)
float MyMethod(float x)
double MyMethod(double x, double y)
```

Consider the following example, which have two methods that add numbers of different type:

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static int PlusMethodInt(int x, int y)
        {
            return x + y;
        }

        static double PlusMethodDouble(double x, double y)
        {
            return x + y;
        }

        static void Main(string[] args)
        {
            int myNum1 = PlusMethodInt(8, 5);
            double myNum2 = PlusMethodDouble(4.3, 6.26);
            Console.WriteLine("Int: " + myNum1);
            Console.WriteLine("Double: " + myNum2);
```

# C# Programming Language

```
    }  
}  
}
```

```
Int: 13  
Double: 10.559999999999999
```

Instead of defining two methods that should do the same thing, it is better to overload one. In the example below, we overload the PlusMethod method to work for both int and double:

## Example

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static int PlusMethod(int x, int y)  
        {  
            return x + y;  
        }  
  
        static double PlusMethod(double x, double y)  
        {  
            return x + y;  
        }  
  
        static void Main(string[] args)  
        {  
            int myNum1 = PlusMethod(8, 5);  
            double myNum2 = PlusMethod(4.3, 6.26);  
            Console.WriteLine("Int: " + myNum1);  
            Console.WriteLine("Double: " + myNum2);  
  
        }  
    }  
}
```

# C# Programming Language

```
Int: 13  
Double: 10.559999999999999
```

**Note:** Multiple methods can have the same name as long as the number and/or type of parameters are different.

## C# CLASSES

- **C# OOP**

### C# - What is OOP?

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

**Tip:** The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

### C# - What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

# C# Programming Language

class	objects
Fruit	Apple Banana Mango

Another example:

class	objects
Car	Volvo Audi Toyota

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

- ## C# Classes and Objects

### Classes and Objects

You learned from the previous chapter that C# is an object-oriented programming language.

Everything in C# is associated with classes and objects, along with its attributes and **methods**. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and methods, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

### Create a Class

To create a class, use the **class** keyword:

Create a class named "Car" with a variable color:

# C# Programming Language

```
class Car
{
    string color = "red";
}
```

## Create an Object

An object is created from a class. We have already created the class named Car, so now we can use this to create objects.

To create an object of Car, specify the class name, followed by the object name, and use the keyword new:

### Example

```
using System;

namespace MyApplication
{
    class Car
    {
        string color = "red";

        static void Main(string[] args)
        {
            Car myObj = new Car();
            Console.WriteLine(myObj.color);
        }
    }
}
```

```
red
```

Note that we use the dot syntax (.) to access variables/fields inside a class (myObj.color).

## Multiple Objects

You can create multiple objects of one class:

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Car
    {
        string color = "red";

        static void Main(string[] args)
        {
            Car myObj1 = new Car();
            Car myObj2 = new Car();
            Console.WriteLine(myObj1.color);
            Console.WriteLine(myObj2.color);
        }
    }
}
```

```
red  
red
```

## Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the fields and methods, while the other class holds the **Main()** method (code to be executed)).

- Car.cs
- Program.cs

```
class Car
{
    public string color = "red";
}
```

## Program.cs

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Car
    {
        public string color = "red";
    }
}
```

```
red
```

- ## C# Class Members

### Class Members

Fields and methods inside classes are often referred to as "Class Members":

#### Example

```
// The class
class MyClass
{
    // Class members
    string color = "red";           // field
    int maxSpeed = 200;             // field
    public void fullThrottle()      // method
    {
        Console.WriteLine("The car is going as fast as it can!");
    }
}
```

### Fields

In the previous chapter, you learned that variables inside a class are called fields, and that you can access them by creating an object of the class, and by using the dot syntax (.).

The following example will create an object of the Car class, with the name **myObj**. Then we print the value of the fields **color** and **maxSpeed**:

# C# Programming Language

## Example

```
using System;

namespace MyApplication
{
    class Car
    {
        string color = "red";
        int maxSpeed = 200;

        static void Main(string[] args)
        {
            Car myObj = new Car();
            Console.WriteLine(myObj.color);
            Console.WriteLine(myObj.maxSpeed);
        }
    }
}
```

```
red
200
```

You can also leave the fields blank, and modify them when creating the object:

## Example

```
using System;

namespace MyApplication
{
    class Car
    {
        string color;
        int maxSpeed;

        static void Main(string[] args)
        {
            Car myObj = new Car();
            myObj.color = "red";
            myObj.maxSpeed = 200;
            Console.WriteLine(myObj.color);
            Console.WriteLine(myObj.maxSpeed);
        }
    }
}
```

# C# Programming Language

red  
200

This is especially useful when creating multiple objects of one class:

## Example

```
using System;

namespace MyApplication
{
    class Car
    {
        string model;
        string color;
        int year;

        static void Main(string[] args)
        {
            Car Ford = new Car();
            Ford.model = "Mustang";
            Ford.color = "red";
            Ford.year = 1969;

            Car Opel = new Car();
            Opel.model = "Astra";
            Opel.color = "white";
            Opel.year = 2005;

            Console.WriteLine(Ford.model);
            Console.WriteLine(Opel.model);
        }
    }
}
```

Mustang  
Astra

## Object Methods

You learned from the **C#** Methods chapter that methods are used to perform certain actions.

# C# Programming Language

Methods normally belongs to a class, and they define how an object of a class behaves.

Just like with fields, you can access methods with the dot syntax. However, note that the method must be public. And remember that we use the name of the method followed by two **parantheses ()** and a **semicolon ;** to call (execute) the method:

## Example

```
using System;

namespace MyApplication
{
    class Car
    {
        string color;                // field
        int maxSpeed;               // field
        public void fullThrottle()   // method
        {
            Console.WriteLine("The car is going as fast as it can!");
        }

        static void Main(string[] args)
        {
            Car myObj = new Car();
            myObj.fullThrottle(); // Call the method
        }
    }
}
```

The car is going as fast as it can!

### Why did we declare the method as **public**, and not **static**?

The reason is simple: a **static** method can be accessed without creating an object of the class, while **public** methods can only be accessed by objects.

## Use Multiple Classes

Remember from the last chapter, that we can use multiple classes for better organization (one for fields and methods, and another one for execution). This is recommended:

# C# Programming Language

## Car.cs

```
class Car
{
    public string model;
    public string color;
    public int year;
    public void fullThrottle()
    {
        Console.WriteLine("The car is going as fast as it can!");
    }
}
```

## Program.cs

```
using System;

namespace MyApplication
{
    class Car
    {
        public string model;
        public string color;
        public int year;
        public void fullThrottle()
        {
            Console.WriteLine("The car is going as fast as it can!");
        }
    }
}
```

Mustang

Astra

The **public** keyword is called an **access modifier**, which specifies that the fields of **Car** are accessible for other classes as well, such as **Program**.

**Tip:** As you continue to read, you will also learn more about other class members, such as constructors and properties.

# C# Programming Language

- C# Constructors

## Constructors

A constructor is a special method that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

```
using System;

namespace MyApplication
{
    // Create a Car class
    class Car
    {
        public string model; // Create a field

        // Create a class constructor for the Car class
        public Car()
        {
            model = "Mustang"; // Set the initial value for model
        }

        static void Main(string[] args)
        {
            Car Ford = new Car(); // Create an object of the Car Class (this will call the
constructor)
            Console.WriteLine(Ford.model); // Print the value of model
        }
    }
}
```

Mustang

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like **void** or **int**).

Also note that the constructor is called when the object is created.

# C# Programming Language

All classes have constructors by default: if you do not create a class constructor yourself, C# creates one for you. However, then you are not able to set initial values for fields.

## Constructor Parameters

Constructors can also take parameters, which is used to initialize fields.

The following example adds a string **modelName** parameter to the constructor. Inside the constructor we set model to **modelName (model=modelName)**. When we call the constructor, we pass a parameter to the constructor ("Mustang"), which will set the value of model to "Mustang":

### Example

```
using System;

namespace MyApplication
{
    class Car
    {
        public string model;

        // Create a class constructor with a parameter
        public Car(string modelName)
        {
            model = modelName;
        }

        static void Main(string[] args)
        {
            Car Ford = new Car("Mustang");
            Console.WriteLine(Ford.model);
        }
    }
}
```

Mustang

You can have as many parameters as you want:

# C# Programming Language

## Example

```
using System;

namespace MyApplication
{
    class Car
    {
        public string model;
        public string color;
        public int year;

        // Create a class constructor with multiple parameters
        public Car(string modelName, string modelColor, int modelYear)
        {
            model = modelName;
            color = modelColor;
            year = modelYear;
        }

        static void Main(string[] args)
        {
            Car Ford = new Car("Mustang", "Red", 1969);
            Console.WriteLine(Ford.color + " " + Ford.year + " " + Ford.model);
        }
    }
}
```

Red 1969 Mustang

**Tip:** Just like other methods, constructors can be **overloaded** by using different numbers of parameters.

- ## C# Access Modifiers

### Access Modifiers

By now, you are quite familiar with the `public` keyword that appears in many of our examples:

# C# Programming Language

```
public string color;
```

The public keyword is an access modifier, which is used to set the access level/visibility for classes, fields, methods and properties.

C# has the following access modifiers:

Modifier	Description
public	The code is accessible for all classes
private	The code is only accessible within the same class
protected	The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about <a href="#">inheritance</a> in a later chapter
internal	The code is only accessible within its own assembly, but not from another assembly.

There's also two combinations: protected internal and **private protected**.

For now, lets focus on **public** and **private** modifiers.

## Private Modifier

If you declare a field with a private access modifier, it can only be accessed within the same class:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Car
    {
        private string model = "Mustang";

        static void Main(string[] args)
        {
            Car myObj = new Car();
            Console.WriteLine(myObj.model);
        }
    }
}
```

Mustang

If you try to access it outside the class, an error will occur:

## Example

```
using System;

namespace MyApplication
{
    class Car
    {
        private string model = "Mustang";
    }
}
```

```
'Car.model' is inaccessible due to its protection level
The field 'Car.model' is assigned but its value is never used
```

## Public Modifier

If you declare a field with a public access modifier, it is accessible for all classes:

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Car
    {
        public string model = "Mustang";
    }
}
```

Mustang

**Note:** By default, all members of a class are private if you don't specify an access modifier:

```
class Car
{
    string model; // private
    string year; // private
}
```

- **C# Properties (Get and Set)**

## Properties and Encapsulation

Before we start to explain properties, you should have a basic understanding of "Encapsulation".

The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users.

To achieve this, you must:

- declare fields/variables as private
- provide public get and set methods, through properties, to access and update the value of a private field.

## Properties

You learned from the previous chapter that private variables can only be accessed within the same class (an outside class has no access to it). However, sometimes we need to access them - and it can be done with properties.

# C# Programming Language

A property is like a combination of a variable and a method, and it has two methods: a get and a set method:

## Example

```
class Person
{
    private string name; // field

    public string Name // property
    {
        get { return name; } // get method
        set { name = value; } // set method
    }
}
```

## Example explained

The Name property is associated with the name field. It is a good practice to use the same name for both the property and the private field, but with an uppercase first letter.

The get method returns the value of the variable name.

The set method assigns a value to the name variable. The value keyword represents the value we assign to the property.

Now we can use the Name property to access and update the private field of the Person class:

## Example

```
using System;

namespace MyApplication
{
    class Person
    {
        private string name; // field
        public string Name // property
        {
            get { return name; }
            set { name = value; }
        }
    }
}
```

# C# Programming Language

Liam

## Automatic Properties (Short Hand)

C# also provides a way to use short-hand / automatic properties, where you do not have to define the field for the property, and you only have to write get; and set; inside the property.

The following example will produce the same result as the example above. The only difference is that there is less code:

### Example

```
using System;

namespace MyApplication
{
    class Person
    {
        public string Name    // property
        { get; set; }
    }
}
```

Liam

## Why Encapsulation?

- Better control of class members (reduce the possibility of yourself (or others) to mess up the code)
- Fields can be made read-only (if you only use the get method), or write-only (if you only use the set method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data.

# C# Programming Language

## • C# Inheritance

Inheritance (Derived and Base Class)

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

- Derived Class (child) - the class that inherits from another class
- Base Class (parent) - the class being inherited from

To inherit from a class, use the `:` symbol.

In the example below, the Car class (child) inherits the fields and methods from the Vehicle class (parent):

## Example

```
class Vehicle // base class (parent)
{
    public string brand = "Ford"; // Vehicle field
    public void honk()           // Vehicle method
    {
        Console.WriteLine("Tuut, tuut!");
    }
}

class Car : Vehicle // derived class (child)
{
    public string modelName = "Mustang"; // Car field
}

class Program
{
    static void Main(string[] args)
    {
        // Create a myCar object
        Car myCar = new Car();

        // Call the honk() method (From the Vehicle class) on the myCar object
        myCar.honk();

        // Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class
        Console.WriteLine(myCar.brand + " " + myCar.modelName);
    }
}
```

# C# Programming Language

Tuut, tuut!  
Ford Mustang

## Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

**Tip:** Also take a look at the next chapter, [Polymorphism](#), which uses inherited methods to perform different tasks.

## The sealed Keyword

If you don't want other classes to inherit from a class, use the sealed keyword:

If you try to access a sealed class, C# will generate an error:

```
sealed class Vehicle
{
    ...
}

class Car : Vehicle
{
    ...
}
```

The error message will be something like this:

```
'Car': cannot derive from sealed type 'Vehicle'
```

## • C# Polymorphism

### Polymorphism and Overriding Methods

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; Inheritance lets us inherit fields and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

# C# Programming Language

For example, think of a base class called Animal that has a method called **animalSound()**. Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

## Example

```
class Animal // Base class (parent)
{
    public void animalSound()
    {
        Console.WriteLine("The animal makes a sound");
    }
}

class Pig : Animal // Derived class (child)
{
    public void animalSound()
    {
        Console.WriteLine("The pig says: wee wee");
    }
}

class Dog : Animal // Derived class (child)
{
    public void animalSound()
    {
        Console.WriteLine("The dog says: bow wow");
    }
}
```

Now we can create Pig and Dog objects and call the **animalSound()** method on both of them:

## Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Animal // Base class (parent)
    {
        public void animalSound()
        {
            Console.WriteLine("The animal makes a sound");
        }
    }

    class Pig : Animal // Derived class (child)
    {
        public void animalSound()
        {
            Console.WriteLine("The pig says: wee wee");
        }
    }

    class Dog : Animal // Derived class (child)
    {
        public void animalSound()
        {
            Console.WriteLine("The dog says: bow wow");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Animal myAnimal = new Animal(); // Create a Animal object
            Animal myPig = new Pig(); // Create a Pig object
            Animal myDog = new Dog(); // Create a Dog object

            myAnimal.animalSound();
            myPig.animalSound();
            myDog.animalSound();
        }
    }
}
```

# C# Programming Language

```
The animal makes a sound  
The animal makes a sound  
The animal makes a sound
```

## Not The Output I Was Looking For

The output from the example above was probably not what you expected. That is because the base class method overrides the derived class method, when they share the same name. However, C# provides an option to override the base class method, by adding the `virtual` keyword to the method inside the base class, and by using the `override` keyword for each derived class methods:

## Example

```
using System;  
  
namespace MyApplication  
{  
    class Animal // Base class (parent)  
    {  
        public virtual void animalSound()  
        {  
            Console.WriteLine("The animal makes a sound");  
        }  
    }  
  
    class Pig : Animal // Derived class (child)  
    {  
        public override void animalSound()  
        {  
            Console.WriteLine("The pig says: wee wee");  
        }  
    }  
  
    class Dog : Animal // Derived class (child)  
    {
```

# C# Programming Language

```
public override void animalSound()
{
    Console.WriteLine("The dog says: bow wow");
}

class Program
{
    static void Main(string[] args)
    {
        Animal myAnimal = new Animal(); // Create a Animal object
        Animal myPig = new Pig(); // Create a Pig object
        Animal myDog = new Dog(); // Create a Dog object

        myAnimal.animalSound();
        myPig.animalSound();
        myDog.animalSound();
    }
}
```

```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
```

## Why and When to Use "Inheritance" and "Polymorphism"?

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

## • C# Abstraction

### Abstract Classes and Methods

Data abstraction is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either abstract classes or interfaces (which you will learn more about in the next chapter).

The abstract keyword is used for classes and methods:

# C# Programming Language

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal
{
    public abstract void animalSound();
    public void sleep()
    {
        Console.WriteLine("Zzz");
    }
}
```

From the example above, it is not possible to create an object of the Animal class:

**Animal myObj = new Animal(); // Will generate an error (Cannot create an instance of the abstract class or interface 'Animal')**

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the Polymorphism chapter to an abstract class.

## Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    // Abstract class
    abstract class Animal
    {
        // Abstract method (does not have a body)
        public abstract void animalSound();

        // Regular method
        public void sleep()
        {
            Console.WriteLine("Zzz");
        }
    }

    // Derived class (inherit from Animal)
    class Pig : Animal
    {
        public override void animalSound()
        {
            // The body of animalSound() is provided here

            Console.WriteLine("The pig says: wee wee");
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```

```
The pig says: wee wee
Zzz
```

# C# Programming Language

- C# Interface

## Interfaces

Another way to achieve abstraction in C#, is with interfaces.

An interface is a completely "abstract class", which can only contain abstract methods and properties (with empty bodies):

### Example

```
// interface
interface Animal
{
    void animalSound(); // interface method (does not have a body)
    void run(); // interface method (does not have a body)
}
```

It is considered good practice to start with the letter "I" at the beginning of an interface, as it makes it easier for yourself and others to remember that it is an interface and not a class.

By default, members of an interface are abstract and public.

**Note:** Interfaces can contain properties and methods, but not fields.

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class. To implement an interface, use the **:** symbol (just like with inheritance). The body of the interface method is provided by the "implement" class. Note that you do not have to use the **override** keyword when implementing an interface:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    // Interface
    interface IAnimal
    {
        void animalSound(); // interface method (does not have a body)
    }

    // Pig "implements" the IAnimal interface
    class Pig : IAnimal
    {
        public void animalSound()
        {
            // The body of animalSound() is provided here
            Console.WriteLine("The pig says: wee wee");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Pig myPig = new Pig(); // Create a Pig object
            myPig.animalSound();
        }
    }
}
```

The pig says: wee wee

## Notes on Interfaces:

- Like abstract classes, interfaces cannot be used to create objects (in the example above, it is not possible to create an "IAnimal" object in the Program class)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interfaces can contain properties and methods, but not fields/variables

# C# Programming Language

- Interface members are by default abstract and public
- An interface cannot contain a constructor (as it cannot be used to create objects)

## Why And When To Use Interfaces?

- 1) To achieve security - hide certain details and only show the important details of an object (interface).
- 2) C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can implement multiple interfaces. Note: To implement multiple interfaces, separate them with a comma.

## Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

### Example

```
using System;

namespace MyApplication
{
    interface IFirstInterface
    {
        void myMethod(); // interface method
    }

    interface ISecondInterface
    {
        void myOtherMethod(); // interface method
    }

    // Implement multiple interfaces
    class DemoClass : IFirstInterface, ISecondInterface
    {
        public void myMethod()
        {
            Console.WriteLine("Some text..");
        }
        public void myOtherMethod()
```

# C# Programming Language

```
{  
    Console.WriteLine("Some other text...");  
}  
}  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        DemoClass myObj = new DemoClass();  
        myObj.myMethod();  
        myObj.myOtherMethod();  
    }  
}
```

```
Some text..  
Some other text...
```

- ## C# Enum

An **enum** is a special "class" that represents a group of constants (unchangeable/read-only variables).

To create an **enum**, use the **enum** keyword (instead of class or interface), and separate the **enum** items with a comma:

### Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    enum Level
    {
        Low,
        Medium,
        High
    }
    class Program
    {
        static void Main(string[] args)
        {
            Level myVar = Level.Medium;
            Console.WriteLine(myVar);
        }
    }
}
```

Medium

**Enum** is short for "enumerations", which means "specifically listed".

## Enum inside a Class

You can also have an **enum** inside a class:

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        enum Level
        {
            Low,
            Medium,
            High
        }
        static void Main(string[] args)
        {
            Level myVar = Level.Medium;
            Console.WriteLine(myVar);
        }
    }
}
```

# C# Programming Language

Medium

## Enum Values

By default, the first item of an **enum** has the value 0. The second has the value 1, and so on.

To get the integer value from an item, you must explicitly convert the item to an int:

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        enum Months
        {
            January,      // 0
            February,     // 1
            March,        // 2
            April,        // 3
            May,          // 4
            June,         // 5
            July          // 6
        }
        static void Main(string[] args)
        {
            int myNum = (int) Months.April;
            Console.WriteLine(myNum);
        }
    }
}
```

3

You can also assign your own **enum** values, and the next items will update the number accordingly:

# C# Programming Language

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        enum Months
        {
            January,      // 0
            February,     // 1
            March=6,      // 6
            April,        // 7
            May,          // 8
            June,         // 9
            July          // 10
        }
        static void Main(string[] args)
        {
            int myNum = (int) Months.April;
            Console.WriteLine(myNum);
        }
    }
}
```

7

## Enum in a Switch Statement

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

## Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        enum Level
        {
            Low,
            Medium,
            High
        }
        static void Main(string[] args)
        {
            Level myVar = Level.Medium;
            switch(myVar)
            {
                case Level.Low:
                    Console.WriteLine("Low level");
                    break;
                case Level.Medium:
                    Console.WriteLine("Medium level");
                    break;
                case Level.High:
                    Console.WriteLine("High level");
                    break;
            }
        }
    }
}
```

Medium level

## Why And When To Use Enums?

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

# C# Programming Language

- C# Files

## Working with Files

The **File** class from the **System.IO** namespace, allows us to work with files:

### Example

```
using System.IO; // include the System.IO namespace  
  
File.SomeFileMethod(); // use the file class with methods
```

The File class has many useful methods for creating and getting information about files.

For example:

Method	Description
AppendText()	Appends text at the end of an existing file
Copy()	Copies a file
Create()	Creates or overwrites a file
Delete()	Deletes a file
Exists()	Tests whether the file exists
ReadAllText()	Reads the contents of a file
Replace()	Replaces the contents of a file with the contents of another file
WriteAllText()	Creates a new file and writes the contents to it. If the file already exists, it will be overwritten.

## Write To a File and Read It

# C# Programming Language

In the following example, we use the WriteAllText() method to create a file named "filename.txt" and write some content to it. Then we use the ReadAllText() method to read the contents of the file:

## Example

```
using System;
using System.IO; // include the System.IO namespace

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            string writeText = "Hello World!"; // Create a text string
            File.WriteAllText("filename.txt", writeText); // Create a file and write the contents
            of writeText to it

            string readText = File.ReadAllText("filename.txt"); // Read the contents of the file
            Console.WriteLine(readText); // Output the content
        }
    }
}
```

Hello World!

- **C# Exceptions - Try..Catch**

## C# Exceptions

When executing C# code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C# will normally stop and generate an error message. The technical term for this is: C# will throw an exception (throw an error).

## C# try and catch

# C# Programming Language

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

## Syntax

```
try
{
    // Block of code to try
}
catch (Exception e)
{
    // Block of code to handle errors
}
```

Consider the following example, where we create an array of three integers:

This will generate an error, because **myNumbers[10]** does not exist.

```
int[] myNumbers = {1, 2, 3};
Console.WriteLine(myNumbers[10]); // error!
```

The error message will be something like this:

```
System.IndexOutOfRangeException: 'Index was outside the bounds of the array.'
```

If an error occurs, we can use try...catch to catch the error and execute some code to handle it.

In the following example, we use the variable inside the catch block (e) together with the built-in Message property, which outputs a message that describes the exception:

## Example

# C# Programming Language

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int[] myNumbers = {1, 2, 3};
                Console.WriteLine(myNumbers[10]);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Index was outside the bounds of the array.

You can also output your own error message:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int[] myNumbers = {1, 2, 3};
                Console.WriteLine(myNumbers[10]);
            }
            catch (Exception e)
            {
                Console.WriteLine("Something went wrong.");
            }
        }
    }
}
```

# C# Programming Language

Something went wrong.

## Finally

The **finally** statement lets you execute code, after try...catch, regardless of the result:

### Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int[] myNumbers = {1, 2, 3};
                Console.WriteLine(myNumbers[10]);
            }
            catch (Exception e)
            {
                Console.WriteLine("Something went wrong.");
            }
            finally
            {
                Console.WriteLine("The 'try catch' is finished.");
            }
        }
    }
}
```

Something went wrong.  
The 'try catch' is finished.

## The throw keyword

The throw statement allows you to create a custom error.

# C# Programming Language

The throw statement is used together with an exception class. There are many exception classes available in C#: ArithmeticException, FileNotFoundException, IndexOutOfRangeException, TimeOutException, etc:

## Example

```
static void checkAge(int age)
{
    if (age < 18)
    {
        throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    }
    else
    {
        Console.WriteLine("Access granted - You are old enough!");
    }
}

static void Main(string[] args)
{
    checkAge(15);
}
```

The error message displayed in the program will be:

```
System.ArithmeticException: 'Access denied - You must be at least 18 years old.'
```

If age was 20, you would not get an exception:

## Example

```
using System;

namespace MyApplication
{
    class Program
    {
        static void checkAge(int age)
        {
            if (age < 18)
            {
                throw new ArithmeticException("Access denied - You must be at least 18 years old.");
            }
        }
    }
}
```

# C# Programming Language

```
    }
} else
{
    Console.WriteLine("Access granted - You are old enough!");
}
}

static void Main(string[] args)
{
    checkAge(20);
}
}
}
```

Access granted - You are old enough!

## Reference

- 1- Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. 2006.
- 2- Ritchie, Dennis M., Brian W. Kernighan, and Michael E. Lesk. The C programming language. Englewood Cliffs: Prentice Hall, 1988.
- 3- Bauer, Christian, Alexander Frink, and Richard Kreckel. "Introduction to the GiNaC framework for symbolic computation within the C++ programming language." Journal of Symbolic Computation 33.1 (2002): 1-12.
- 4- Stroustrup, Bjarne. "An overview of the C++ programming language." Handbook of object technology (1999).
- 5- Hejlsberg, Anders, et al. C# Programming Language. Addison-Wesley Professional, 2010.
- 6- Hejlsberg, Anders, Scott Wiltamuth, and Peter Golde. C# language specification. Addison-Wesley Longman Publishing Co., Inc., 2003.
- 7- <https://www.w3schools.com/>