Programming Fundamentals  
Tutorial 02 - Input and Output, Fundamental Datatypes and Arithmetic Operations

## Introduction

This tutorial will introduce you to the basics of input and output in C/C++ programs, as well as fundamental datatypes and arithmetic operations.

## Input and Output (I/O)

In **Tutorial 01 Exercise 02** you looked at a simple program that declared three variables, assigned them values and performed an addition operation. As you had access to the project and the source code you were able to step through the code using the debugger to see what the values of the variables were. However, if you were to run the program on its own by double-clicking on the **.exe** file then you wouldn't get to see much as the program doesn't **output** the variables to the screen. The way to do this is to use the function printf().

### Output Using **printf()**

You've come across printf() before, in the **Tutorial 01 Hello World** exercise:

#include <stdio.h>

int main()

{

printf("Hello World!\n");

getchar();

return 0;

}

By default the place where the output of a program goes is the screen, and printf() is the mechanism used in a program for output to the **default output**, so the above code displays "Hello World!" on screen.

printf(); essentially directs data to the output.

The \n at the end of the text tells the printf() function to print a new line , this makes any subsequent output go to the next line on screen. Just like pressing the enter key in Notepad.

The above example uses one block of text and one \n, but you can chain together as many as you want. So:

print(“Hello World\n Mark is here\n”)

outputs the same as:

printf(“Hello World\n”);

printf(“Mark is here\n”);

and:

printf(“Hello World”);

printf(“\n”);

printf(“Mark is here”);

printf(“\n”);

The text that is sent to the output in the above examples, such as "Hello World!" is called a **string literal**, which is a constant string value defined in your program. However, you are not restricted to outputting text to the screen, you can output other things such as numbers and variables, e.g.

int my\_age;

my\_age = 25;

printf("I am %d years old", **my\_age);**

**printf(**"I am %d years old", 25**);**

In order to use printf() you need to tell the compiler that you want to use it. This is done using the following:

#include <stdio.h>

#include <sdtio.h> tells the compiler to include the **stdio** library which allows you to use printf(), along with many other I/O operations. Notice that you **don't** need a semi-colon after the #include directive.

### Input using scanf()

You are now able to display the output of the program on the screen, but it's often useful to get **input** from whoever is using the program. This is where scanf() comes in. scanf() receives input from the **default input,** which is usually the keyboard. It is used in conjunction with a *specifier* character and a **variable** like so:

int my\_age;

my\_age = 25;

printf(“How old are you?\n”);

scanf(“%d”, &my\_age);

printf("You are %d years old", my\_age);

When the program gets to the scanf() statement it waits for some input, usually from the keyboard. To input a value for this, type in an integer number and press **Enter**. The value that was input is then assigned to the variable, in this case my\_age. Note that any value in the variable being used in the **scanf()** statement is overwritten in the same way as a normal assignment statement, so my\_age will no longer be 25, unless 25 is the value entered of course.

Don’t forget to put the “&” at the beginning of the variable, e.g. “&my\_age”. The “&“will be explained to you in the coming weeks.

### Using getchar()

The final I/O operation in the program to discuss is getchar(). This function can be used to get the **value of a single character** from the keyboard without waiting for the Enter key to be pressed. However, all it is being used for in this case is to **pause** the program so that you can see what is being displayed before the program exits, the actual key pressed is ignored. An example of its normal use is below –

char c;

printf("Enter character: ");

c = getchar();

printf("\n Character entered: ");

putchar(c);

**Fundamental Data Types**

So far you have used character and string literals such as "Hello World!", integer literals such as 25, and basic integer variables. There are several other fundamental data types in C\C++ -

### Floating Point Numbers

**Floating point numbers**, declared as floatordouble, have both an integer and a fractional part. The difference between the two types is that double uses **8 bytes** (64 bits) per variable instead of **4 bytes** (32 bits) for floats, and this means doubles can store bigger and more accurate fractional numbers. You will normally use floats as they take up less space and are faster than using doubles, you only need doubles for very accurate calculations. Here's some code using float:

float my\_float\_pi;

my\_float\_pi = 3.14159265358979f;

printf("The value of pi is %f\n", my\_float\_pi);

Here my\_float\_pi is of type float, and 3. 14159265358979f is a **floating point literal**. The 'f' tells the compiler that the literal is a float as opposed to a double.

The printf() function automatically detects that it is outputting a floating point number and displays it accordingly. If you try this code you might notice that it doesn't output the full fractional value, but instead just 3.14159. This is because the default precision that printf() uses is 6, so only the first 6 significant numbers are displayed. This can be increased to 15 decimal places by passing “**%0.15f**” to **printf()**:

float my\_float\_pi;

my\_float\_pi = 3.14159265358979f;

printf("The value of pi is %0.15f\n",my\_float\_pi);

This sets the precision to 15 significant figures. If you ran this code you might expect to see 3. 14159265358979, but in fact you would see a different value, 3.141592741012573. This is because a float is not accurate enough to that many significant digits, so this is the closest value it can be. If you need this level of precision, then you can use a double variable, which will contain and display the correct value:

double my\_double\_pi;

my\_double\_pi = 3.14159265358979; // NOTE - no 'f'

printf("The value of pi is %0.15f\n", my\_double\_pi); outputs 3.14159265358979

Be aware that when using printf() to display floating point values, very small and very large numbers might be displayed in a form of **scientific notation**, e.g. 0.000000123 might be shown as 1.23e-007 (this means 1.23 x 10-7). The format of the number output is dependent on its size and the precision printf() has been told to print.

### Character Variables

Character variables are declared as char. They are 1 byte in size and can represent either a single character (e.g. 'A', 'z', '3', '#', etc.) or a small number (1 byte can store 256 different values). The number that is stored corresponds to the **ASCII** (pronounced 'Asskee', means American Standard Code for Information Interchange) value of the character. Use the following code to declare and define a character variable:

char my\_char;

my\_char = 'A'; // NOTE - single quotes

printf("The character is %c\n", my\_char);

As a char can be treated as a number it can be assigned to an integer variable. The ASCII code for capital A happens to be 65, so the code on the following page:

char my\_char;

int my\_int;

my\_char = 'A';

my\_int = my\_char; //The ASCII value of A is 65 so it is converted and stored as an integer;

printf("The character is %c\n", my\_char); // outputs A

printf("The character ASCII code is %d", my\_int); // outputs 65

will output

The character is A

The character ASCII code is 65

### Boolean Variables

A **boolean** **variable** is declared using the keyword bool. Boolean variables require the **stdbool.h** library to be included in your program. A bool simply stores a value of 1 or 0, representing true or false respectively. You can set a boolean using 0 and 1 literals, or the true and false keywords, e.g.

#include <stdbool.h>

bool my\_bool;

my\_bool = true; // equivalent to **my\_bool = 1**;

printf("The value of my\_bool is %d\n", my\_bool);

my\_bool = false; // equivalent to **my\_bool = 0;**

printf("The value of my\_bool is now %d\n", my\_bool);

will output:

The value of my\_bool is 1

The value of my\_bool is now 0

Notice that the output from printf() is 1 or 0, instead of true or false as you might expect. This is because a boolean is generally used in a numerical or logical way, not as text, and true and false are just keywords that represent 1 and 0. You will see later in the unit how to output the value of a boolean in text format.

### Signed and Unsigned Data Types

int and char are **signed data types** by default, which means that they can store negative values. If chars are 1 byte then the range of values they can contain are -128 to +127. For ints, which are 4 bytes in length, the range will be -2147483648 to +2147483647. You can explicitly declare a signed data type, but it isn't required:

signed int my\_signed\_int; // same as int **my\_signed\_int;**

Sometimes you might know that you don't need negative numbers and therefore you can declare ints and chars as unsigned. This means that the range of values that can be represented for chars becomes 0 to 255, and for ints it becomes 0 to 4294967295. For instance:

unsigned int my\_unsigned\_int;

my\_unsigned\_int = 4294967295;

printf("The value of my\_unsigned\_int is %u\n",my\_unsigned\_int);

Be careful to use a data type that has sufficient range for the values you need. For example, if an **integer** (signed or unsigned) is set to its maximum value and you try to add one then **there is no error**. What happens is the number **wraps around back to zero**, much like a car mileometer:

unsigned int my\_unsigned\_int;

my\_unsigned\_int = 4294967295;

printf("The value of my\_unsigned\_int is %u\n", my\_unsigned\_int); //Outputs 4294967295

my\_unsigned\_int = my\_unsigned\_int + 1; //Wraps around to 0

printf("The value of my\_unsigned\_int plus 1 is %u\n", my\_unsigned\_int); //Outputs 0

### Constants

If you use a number that will never change in several different places it is a good idea to use a **constant**. Constants are **user defined labels** for values that will never change that can be used to replace literal values. As an example, if you had a situation in a four player co-op game where 50 bonus points were added to each player’s score:

int player\_1\_score;

int player\_2\_score;

int player\_3\_score;

int player\_4\_score;

player\_1\_score = 100;

player\_2\_score = 200;

player\_3\_score = 250;

player\_4\_score = 80;

// Award bonus points to all players

player\_1\_score = player\_1\_score + 50;

player\_2\_score = player\_2\_score + 50;

player\_3\_score = player\_3\_score + 50;

player\_4\_score = player\_4\_score + 50;

As you can see in the code each player needs to add the same number. If you ever decided to change the bonus points value then you would have to update all four values. This might seem trivial, but in more complex programs these updates could be scattered all over the code. A better way might be to use a variable:

int bonus;

bonus = 50;

// Award bonus points to all players

player\_1\_score = player\_1\_score + bonus;

player\_2\_score = player\_2\_score + bonus;

player\_3\_score = player\_3\_score + bonus;

player\_4\_score = player\_4\_score + bonus;

So to change the bonus requires just a single change to the code, and this is a perfectly valid way of doing it if you would be changing the bonus value at run time. However, if the bonus value will only be adjusted during the game's development, e.g. for gameplay balance, and really shouldn't change at run time, then a better way would be to use a constant so the compiler will tell you of any inadvertent changes:

const int CONSTANT\_BONUS = 50;

// Award bonus points to all players

player\_1\_score = player\_1\_score + CONSTANT\_BONUS;

player\_2\_score = player\_2\_score + CONSTANT\_BONUS;

player\_3\_score = player\_3\_score + CONSTANT\_BONUS;

player\_4\_score = player\_4\_score + CONSTANT\_BONUS;

Using all capitals for constants is a convention; you don't have to name them this way but it is a good indicator that it is a constant to yourself and others. Also notice the way the constant has been declared **and** defined **at the same time**. This is necessary, as obviously you **can't change a constant**. In fact, the compiler won't let you create a constant without assigning it a value if you try the following in your own code:

const int CONSTANT\_BONUS; //Won't compile, not been assigned a value

CONSTANT\_BONUS = 50; //Can't do, is a constant!

Another way of defining constants that you might see is like this:

#define CONSTANT\_BONUS 50

This works as well, but the const version is better as it lets the compiler know the correct type is being used.

### Declaring and Defining Variables in a Single Statement

You can declare and define variables in the same way as constants if you wish, e.g.

int my\_int;

my\_int = 0;

can be written as

int my\_int = 0;

## Arithmetic Operators

In **Tutorial 02 Exercise 02** you used the most basic arithmetic operator, addition (+), and in **Tutorial 02 Exercise 03** the multiplication operator (\*). There are three other arithmetic operators, **subtraction** (-), **division** (/) and **modulus** (%). Here is some code that uses all of these operators on integers:

int my\_int\_1 = 13;

int my\_int\_2 = 4;

printf("The value of my\_int\_1 + my\_int\_2 is %d\n", my\_int\_1 + my\_int\_2);

printf("The value of my\_int\_1 - my\_int\_2 is %d\n", my\_int\_1 - my\_int\_2);

printf("The value of my\_int\_1 \* my\_int\_2 is %d\n", my\_int\_1 \* my\_int\_2);

printf("The value of my\_int\_1 / my\_int\_2 is %d\n", my\_int\_1 / my\_int\_2);

printf("The value of my\_int\_1 % my\_int\_2 is %d\n", my\_int\_1 % my\_int\_2);

which would output:

The value of my\_int\_1 + my\_int\_2 is 17

The value of my\_int\_1 - my\_int\_2 is 9

The value of my\_int\_1 \* my\_int\_2 is 52

The value of my\_int\_1 / my\_int\_2 is 3

The value of my\_int\_1 % my\_int\_2 is 1

The first 3 are pretty obvious, but the last two may not be so much. When using division on integers there can be no fractional part, so it gets discarded and just the whole number portion remains. Therefore, although 13 divided by 4 equals 3.25, only the 3 portion is kept in the operation. It is also worth noting that the result never rounds up – any fractional decimal is simply discarded. Modulus may be new to you, but all it means is the **remainder of integer division**. You've seen that integer 13 divided by 4 equals 3, so if you calculate 4 multiplied by 3 (12) you can find the remainder of the division (13 minus 12) which equals 1.

If you were to use floating point numbers the only differences would be that the division would return 3.25, and that modulus isn't a valid operation on floating point values:

float my\_float\_1 = 13.0f;

float my\_float\_2 = 4.0f;

printf("The value of my\_float\_1 + my\_float\_2 is %f\n", my\_float\_1 + my\_float\_2); // 17

printf("The value of my\_float\_1 - my\_float\_2 is %f\n", my\_float\_1 - my\_float\_2); // 9

printf("The value of my\_float\_1 \* my\_float\_2 is %f\n", my\_float\_1 \* my\_float\_2); // 52

printf("The value of my\_float\_1 / my\_float\_2 is %f\n", my\_float\_1 / my\_float\_2); // 3.25

Note that in the above examples the arithmetic operations aren't assigned to a variable, but are **evaluated in place** with the printf() operations. What this means is the result is calculated and displayed on the screen, but it isn't stored anywhere. This can be used if you don't need the result later on, otherwise you would need to store the result in another variable:

float result = my\_float\_1 + my\_float\_2

printf("The value of my\_float\_1 + my\_float\_2 is %f\n", result);

### Compound Assignment

All of the operators have a shortcut notation called the **compound assignment**. All this means is that instead of using:

my\_int = my\_int + 25;

my\_float\_1 = my\_float\_1 / my\_float\_2;

You can use the shortcuts

my\_int += 25; // same as my\_int = my\_int + 25;

my\_float\_1 /= my\_float\_2; // same as my\_float\_1 = my\_float\_1 / my\_float\_2;

### Increment and Decrement Operators

Another shortcut that is often used are the **increment** (++) and **decrement** (--) operators. The increment operator adds one to a variable and the decrement operator subtracts 1. An additional feature of these operators is that they can be both **prefix** and **postfix** operators, that is they can go before or after the variable they are changing. Here are examples to show how they work:

my\_int = my\_int + 1; // normal addition

my\_int += 1; // compound addition

++my\_int; // increment with prefix operator

my\_int++; // increment with postfix operator

All of these statements do exactly the same thing, which is add 1 to a variable. The difference between the prefix and postfix versions only become apparent in more complex arithmetic expression - the prefix version updates the variable **before** the rest of the expression and the postfix **after**. The following code uses both types to illustrate this:

int my\_int1 = 5;

int my\_int2 = 5;

int my\_int\_result;

my\_int\_result = ++my\_int1;

printf("The value of my\_int\_result = ++my\_int1 is %d\n", my\_int\_result); // output 6

my\_int\_result = my\_int2++;

printf("The value of my\_int\_result = my\_int2++ is %d\n", my\_int\_result); // output 5

As you can see, using ++my\_int1 causes my\_int1 to be incremented from 5 to 6 **before** the assignment to my\_int\_result so it is assigned the value 6, whereas using my\_int2 ++ causes my\_int2 to be incremented **after** the assignment, so my\_int\_result is assigned the value 5. Be careful when you use them, make sure you understand when these operators change the variable if used in a complex arithmetic operation.

### Precedence and Brackets

**Precedence** is the order of operations that happen in an arithmetic expression. The most basic rule is that operators are evaluated from **left to right**, but certain operators have a **higher precedence** than others, **so get evaluated first**. Any C/C++ text book will list the precedence for all operators, but for now all you need to know is that multiplication, division and modulus have the same precedence, and they have a higher precedence than addition or subtraction. The upshot of this is that multiplication, division and modulus operations will be evaluated before addition and subtraction. So, for instance:

int my\_int = 3 + 4 \* 5;

will evaluate to 23, not 35 as you might expect by evaluating from left to right, as precedence means that 4 \* 5 is done first and not 3 + 4. In order to get around this you need to use **parentheses** (or **brackets**). Brackets allow you to determine exactly how operations should be ordered as operations inside brackets are done first. So if you changed the above line to:

int my\_int = (3 + 4) \* 5;

then 3 + 4 would be done first. You are probably best off **always** using brackets to show the order you want, even if it would work correctly without. It just means you can see what is intended without having to look up the precedence rules.

### Enumerations

Often you will find that you are using several numerical values in order to represent a set of different distinct states or items. For instance, you might be using an integer to store the current general wind direction, with 0 meaning no wind, 1 meaning a north wind, 2 meaning east wind, etc. So, you could set the wind direction variable like this:

int wind\_direction = 2;

but it's not exactly clear from this what the wind direction is. You could put a comment in, but a better way might be to use constants:

const int NO\_WIND = 0;

const int NORTH\_WIND = 1;

const int EAST\_WIND = 2;

const int SOUTH\_WIND = 3;

const int WEST\_WIND = 4;

int current\_wind\_direction = NO\_WIND; // will set variable to 0

// code to check wind

current\_wind\_direction = NORTH\_WIND; // will set variable to 1

which makes it much clearer. However, a better solution is to use an **enumeration**. Enumerations are a new data type that contain specific sets of named integer values. So, for the wind direction example a new enumeration could be:

enum enum\_wind\_directions{NO\_WIND, NORTH\_WIND, EAST\_WIND, SOUTH\_WIND, WEST\_WIND};

This creates a new data type called enum\_wind\_directions that has five distinct states. By default, the first element of an enumeration is equivalent to 0, and subsequent elements are incremented by 1, so the value of NO\_WIND is 0 through to WEST\_WIND with value 4. It's good practice to prefix the new enumeration name with something to identify it as an enumeration, in this case ‘enum’. Also good practice is to use capitals for the elements as they are essentially constant values.

You create and use enumerations just like other data types:

enum enum\_wind\_directions current\_wind\_direction = NO\_WIND; //Creates a variable of type enum\_wind\_direction

//and sets its value to NO\_WIND

current\_wind\_direction = NORTH\_WIND; //Alters its value to NORTH\_WIND

As enumerations are integer values you **can** assign them to integers:

current\_wind\_direction = NORTH\_WIND;

int x = current\_wind\_direction; // as NORTH\_WIND is equivalent to 1, sets x to 1

As mentioned above, the default values for enumerations start at 0 and increment by 1 per element. However, you are not limited to this. You can set elements to specific numbers as required:

enum enum\_car\_makes{BMW = 4, AUDI = 23, FIAT, FORD, SAAB = 7, KIA};

So BMW is 4, AUDI is 23 and SAAB is 7. Any elements not given a specific value increment by 1 from the previous element, so FIAT is 24, FORD is 25 and KIA is 8.

## Exercises -

#### Exercise 01

1. Create a new project with a new **main.c** as in previous exercises.
2. Add **comments** at the start to identify the program.
3. Add #include <stdio.h> in the correct place
4. Add a main() function, don't forget the **braces** that form the statement block that will contain all of the statements.
5. Declare three integer variables named as you wish.
6. For each variable, display "Input a number" and use scanf() to get a number from the keyboard.
7. Once all three numbers have been entered, **display** them in the reverse order that they were entered, e.g. if 1, 3 and 5 are entered the program should display 5, 3, 1.
8. Don't forget to use getchar() to pause.
9. Use the debugger to step through the code as usual to check how the program works.

#### Exercise 02

1. Create a new project and add **main.c** file, includes and main() function as usual.
2. Declare three floating point variables named as you wish.
3. Ask the user for 3 numbers.
4. Once the numbers have been entered, **display** all three numbers and their sum on the screen in the format:

Number 1 + Number 2 + Number 3 = Sum

e.g. if 1, 3 and 5 are entered, the program should display 1 + 3 + 5 = 9.

1. Calculate the **average** of the 3 numbers and display it on the next line (Evaluate the average **in place** if you can).
2. Step through the code in the debugger.

#### Exercise 03

1. Create a new project as usual.
2. Ask the user for a floating point number that is the radius of a circle.
3. Use that number to calculate the circumference and area of a circle, and display both of these. Consider where you might use a constant in the calculations. Examine your code and explain in comments and your notebook why using a constant is a good idea here.
4. Run the program again and enter 1000 for the radius. Write in your notebook what is output to the screen.
5. Experiment changing the precision of printf() to attempt to ensure neither number is displayed with scientific notation when 1000 is entered for the radius.
6. Step through the code in the debugger.

#### Exercise 04

1. Create a new project as usual.
2. Ask the user for two integer and two floating point numbers.
3. Use **all** of the appropriate arithmetic operators with the two integers, (e.g. a+b, a-b, etc.) and display the results.
4. Use **all** of the appropriate arithmetic operators with the two floats, and display the results.
5. Use all possible arithmetic operators with one of the floats and one of the integers, and display the results. Write in your notebook and in comments what happens. Research and try to explain any unexpected results.
6. Step through the code in the debugger.

#### Exercise 05

1. Create a new project as usual.
2. Ask the user for one integer and one floating point number.
3. Use both the compound assignment operator, prefix increment operator and postfix increment operator to change the value of both numbers input, displaying the new value after each statement.
4. Step through the code in the debugger.

#### Exercise 06

1. Create a new project as usual.
2. Ask the user for three integers.
3. In code, evaluate the expression: number 1 + number 2 \* number 3, so that the **addition is done first**, and output the result. Explain what you did.
4. Step through the code in the debugger.

#### Exercise 07

1. Create a new project as usual.
2. Declare an integer constant, and a single precision floating point constant.
3. Create an enumeration that contains several different elements, choose a subject that interests you for the states or objects in the enumeration.
4. Create several variables with your newly created enumeration, and assign them values.
5. Output the constants to the screen.
6. Output the enum variables to the screen. Are the values output to the screen what you expect? Explain why the variables are output in this way.
7. (Extra) If you know how, add code to display more meaningful values to the screen for the enum variables
8. Step through the code in the debugger.