# C programming Tutorial
# Introduction to C Programming

## 1. Getting Started - Write your First Hello-world C Program

Let's begin by writing our first C program that prints the message "Hello, world!" on the display console:

```
Hello, world!
```

**Step 1: Write the Source Code:** Enter the following source codes using a programming text editor (such as NotePad++ for Windows or gEdit for UNIX/Linux/Mac) or an Interactive Development Environment (IDE) (such as CodeBlocks, Eclipse, NetBeans or MS Visual Studio - Read the respective "How-To" article on how to install and get started with these IDEs).

Do not enter the line numbers (on the left panel), which were added to help in the explanation. Save the source file as "`Hello.c`". A C source file should be saved with a file extension of "`.c`". You should choose a *filename* which reflects the purpose of the program.

```
1 /*
2  * First C program that says Hello (Hello.c)
3  */
4 #include <stdio.h>  // Needed to perform IO operations
5
6 int main() {                    // Program entry point
7    printf("Hello, world!\n");  // Says Hello
8    return 0;                   // Terminate main()
9 }                               // End of main()
```

**Step 2: Build the Executable Code:** Compile and Link (aka Build) the source code "`Hello.c`" into executable code ("`Hello.exe`" in Windows or "`Hello`" in UNIX/Linux/Mac).

- On IDE (such as CodeBlocks), push the "Build" button.
- On Text editor with the GNU GCC compiler, start a CMD Shell (Windows) or Terminal (Mac, Linux) and issue these commands:
- ```
  // Windows (CMD shell) - Build "Hello.c" into "Hello.exe"
  > gcc -o Hello.exe Hello.c

  // UNIX/Linux/Mac (Bash shell) - Build "Hello.c" into "Hello"
  $ gcc -o Hello Hello.c
  ```
- where `gcc` is the name of GCC C compiler; `-o` option specifies the output filename ("`Hello.exe`" for Windows or "`Hello`" for UNIX/Linux/Mac); "`Hello.c`" is the input source file.

**Step 3: Run the Executable Code:** Execute (Run) the program.

- On IDE (such as CodeBlocks), push the "Run" button.
- On Text Editor with GNU GCC compiler, issue these command from CMD Shell (Windows) or Terminal (UNIX/Linux/Mac):
  ```

- ```
  // Windows (CMD shell) - Run "Hello.exe" (.exe is optional)
  > Hello
  Hello, world!

  // UNIX/Linux/Mac (Bash shell) - Run "Hello" (./ denotes the current directory)
  $ ./Hello
  Hello, world!
  ```

## Brief Explanation of the Program

```
/* ...... */
// ... until the end of the line
```

These are called *comments*. Comments are NOT executable and are ignored by the compiler. But they provide useful explanation and documentation to your readers (and to yourself three days later). There are two kinds of comments:

1. *Multi-line Comment*: begins with `/*` and ends with `*/`. It may span more than one lines (as in Lines 1-3).
2. *End-of-line Comment*: begins with `//` and lasts until the end of the current line (as in Lines 4, 6, 7, 8, and 9).

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

The "`#include`" is called a *preprocessor directive*. A preprocessor directive begins with a `#` sign, and is processed before compilation. The directive "`#include <stdio.h>`" tells the preprocessor to include the "`stdio.h`" header file to support input/output operations. This line shall be present in all our programs. I will explain its meaning later.

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

defines the so-called `main()` *function*. The `main()` function is the *entry point* of program execution. `main()` is required to return an `int` (integer).

```
printf("Hello, world!\n");
```

We invoke the function `printf()` to print the string "Hello, world!" followed by a newline (`\n`) to the console. The newline (`\n`) brings the cursor to the beginning of the next line.

```
return 0;
```

terminates the `main()` function and returns a value of 0 to the operating system. Typically, return value of 0 signals normal termination; whereas value of non-zero (usually 1) signals abnormal termination. This line is optional. C compiler will implicitly insert a "`return 0;`" to the end of the `main()`function.
```

## 2. C Terminology and Syntax

**Statement**: A programming *statement* performs a piece of programming action. It must be terminated by a semi-colon (`;`) (just like an English sentence is ended with a period) as in Lines 7 and 8.

**Preprocessor Directive**: The `#include` (Line 4) is a *preprocessor directive* and NOT a programming statement. A preprocessor directive begins with hash sign (#). It is processed before compiling the program. A preprocessor directive is NOT terminated by a semicolon - Take note of this rule.

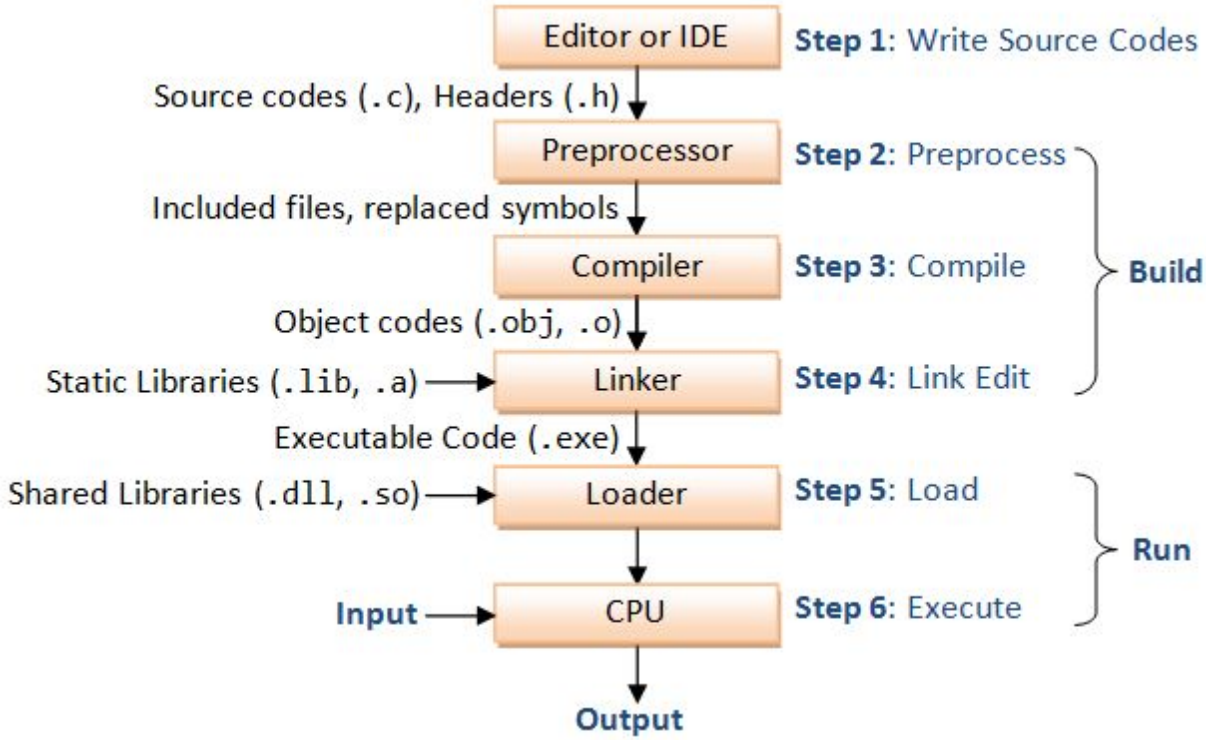
**Block**: A *block* is a group of programming statements enclosed by braces `{ }`. This group of statements is treated as one single unit. There is one block in this program, which contains the *body* of the `main()` function. There is no need to put a semi-colon after the closing brace.

**Comments**: A multi-line comment begins with `/*` and ends with `*/`. An end-of-line comment begins with `//` and lasts till the end of the line. Comments are NOT executable statements and are ignored by the compiler. But they provide useful explanation and documentation. *Use comments liberally*.

**Whitespaces**: Blank, tab, and newline are collectively called whitespaces. Extra whitespaces are ignored, i.e., only one whitespace is needed to separate the tokens. But they could help you and your readers better understand your program. *Use extra whitespaces liberally*.

**Case Sensitivity**: C is *case sensitive* - a *ROSE* is NOT a *Rose*, and is NOT a *rose*.

## 3. The Process of Writing a C Program




**Step 1:** Write the source codes (`.c`) and header files (`.h`).

**Step 2:** Pre-process the source codes according to the *preprocessor directives*. The preprocessor directives begin with a hash sign (#), such as `#include` and `#define`. They indicate that certain manipulations (such as including another file or replacement of symbols) are to be performed BEFORE compilation.

**Step 3:** Compile the pre-processed source codes into object codes (`.obj`, `.o`).

**Step 4:** Link the compiled object codes with other object codes and the library object codes (`.lib`, `.a`)to produce the executable code (`.exe`).

**Step 5:** Load the executable code into computer memory.

**Step 6:** Run the executable code.

## 4. C Program Template

You can use the following *template* to write your C programs. Choose a *meaningful* filename for you source file that reflects the purpose of your program with file extension of "`.c`". Write your programming statements inside the body of the `main()` function. Don't worry about the other terms for the time being. I will explain them later.

```
1 /*
2  * Comment to state the purpose of this program (filename.c)
3  */
4#include <stdio.h>
5
6 int main() {
7   // Your Programming statements HERE!
8
9   return 0;
10 }
```

# 5. Let's Write a Program to Add a Few Numbers

## 5.1 Example: Adding Two Integers

Let's write a C program called "`Add2Integers.c`" to add two integers as follows:

**Add2Integers.c**

```
1 /*
2  * Add two integers and print their sum (Add2Integers.c)
3  */
4 #include <stdio.h>
5
6 int main() {
7   int integer1; // Declare a variable named integer1 of the type integer
8   int integer2; // Declare a variable named integer2 of the type integer
9   int sum;      // Declare a variable named sum of the type integer
10
11   integer1 = 55;  // Assign value to variable integer1
12   integer2 = 66;  // Assign value to variable integer1
13   sum = integer1 + integer2;  // Compute the sum
14
15  // Print the result
16   printf("The sum of %d and %d is %d.\n", integer1, integer2, sum);
17
18   return 0;
19 }
```

```
The sum of 55 and 66 is 121.
```

### Dissecting the Program

```
int integer1;
```

```
int integer2;
```

```
int sum;
```

We first declare three `int` (integer) variables: `integer1`, `integer2`, and `sum`. A *variable* is a named storage location that can store a value of a particular *data type*, in this case, `int` (integer). You can declare one variable in one statement. You could also declare many variables in one statement, separating with commas, e.g.,

```
int integer1, integer2, sum;
```

```
integer1 = 55;
```

```
integer2 = 66;
```

```
sum = integer1 + integer2;
```

We assign values to variables `integer1` and `integer2`; compute their sum and store in variable `sum`.

```
printf("The sum of %d and %d is %d.\n", integer1, integer2, sum);
```

We use the `printf()` function to print the result. The first argument in `printf()` is known as the *formatting string*, which consists of normal texts and so-called *conversion specifiers*. Normal texts will be printed as they

are. A *conversion specifier* begins with a percent sign (%), followed by a code to indicate the data type, such as d for decimal integer. You can treat the %d as *placeholders*, which will be replaced by the value of variables given after the formatting string in sequential order. That is, the first %d will be replaced by the value of `integer1`, second %d by `integer2`, and third %d by `sum`. The \ndenotes a newline character. Printing a \n bring the cursor to the beginning of the next line.

## 5.2 Example: Prompting User for Inputs

In the previous example, we assigned fixed values into variables `integer1` and `integer2`. Instead of using fixed values, we shall prompt the user to enter two integers.

**PromptAdd2Integers.c**

```
/*
 * Prompt user for two integers and print their sum (PromptAdd2Integers.c)
 */
#include <stdio.h>

int main() {
   int integer1, integer2, sum;  // Declare 3 integer variables

   printf("Enter first integer: ");  // Display a prompting message
   scanf("%d", &integer1);           // Read input from keyboard into integer1
   printf("Enter second integer: "); // Display a prompting message
   scanf("%d", &integer2);           // Read input into integer2

   sum = integer1 + integer2;        // Compute the sum

  // Print the result
   printf("The sum of %d and %d is %d.\n", integer1, integer2, sum);

   return 0;
}
```

```
Enter first integer: 55
Enter second integer: 66
The sum of 55 and 66 is 121.
```

### Disecting the Program

```
int integer1, integer2, sum;
```

We first declare three `int` (integer) variables: `integer1`, `integer2`, and `sum` in one statement.

```
printf("Enter first integer: ");
```

We use the `printf()` function to put out a prompting message.

```
scanf("%d", &integer1);
```

We then use the `scanf()` function to read the user input from the keyboard and store the value into variable `integer1`. The first argument of `scanf()`is the *formatting string* (similar to `printf()`). The %d *conversion specifier* provides a *placeholder* for an integer, which will be substituted by variable `integer1`. Take note that

we have to place an ampersand sign (&), which stands for address-of operator, before the variable, I shall explain its significance later. It is important to stress that **missing ampersand (&) in `scanf()` is a common error**, which leads to abnormal termination of the program.

## Reading Multiple Integers

You can read multiple items in one `scanf()` statement as follows:

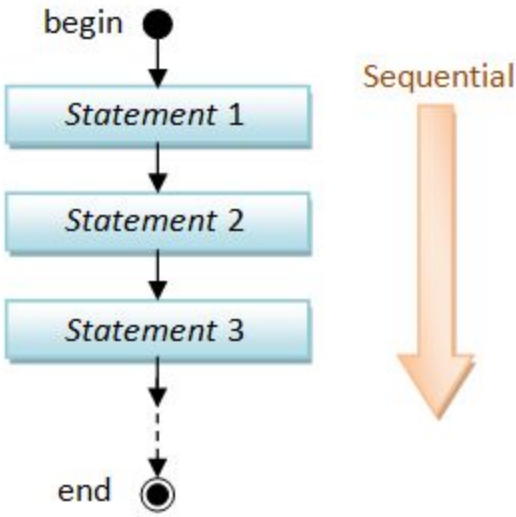
```
printf("Enter two integers: ");      // Display a prompting message
scanf("%d%d", &integer1, &integer2); // Read two integers
```

In the `scanf()`, the first `%d` puts the first integer entered into variable `integer1`, and the second `%d` puts into `integer2`. Again, remember to place an ampersand (&) before the variables in `scanf()`.

## Exercises

1. Print each of the following patterns. Use one `printf()` statement for each line of outputs. End each line by printing a newline (`\n`).
2. 
```
* * * * *      * * * * *    * * * * *
 * * * * *     *       *     *     *
* * * * *      *       *      *   *
 * * * * *     *       *       * *
* * * * *      * * * * *        *
   (a)             (b)           (c)
```
3. Print the above patterns using ONE `printf()` statement.
4. Write a program to prompt user for 5 integers and print their sum. Use five `int` variables `integer1` to `integer5` to store the five integers.
5. Write a program to prompt user for 5 integers and print their product. Use an `int` variable `product` to store the product and operator * for multiplication.

# 6. What is a Program?




A *program* is *a sequence of instructions* (called *programming statements*), executing one after another - usually in a *sequential* manner, as illustrated in the previous example and the following flow chart.

**Example (Sequential):** The following program (`CircleComputation.c`) prompts user for the radius of a circle, and prints its area and circumference. Take note that the programming statements are executed sequentially - one after another in the order that they are written.

```
1/*
2  * Prompt user for the radius of a circle and compute its area and
3circumference
4 * (CircleComputation.c)
5 */
6#include <stdio.h>
7
8int main() {
9   double radius, circumference, area; // Declare 3 floating-point variables
10   double pi = 3.14159265;             // Declare and define PI
11
12   printf("Enter the radius: "); // Prompting message
13   scanf("%lf", &radius);        // Read input into variable radius
14
15  // Compute area and circumference
16   area = radius * radius * pi;
17   circumference = 2.0 * radius * pi;
18
19  // Print the results
20   printf("The radius is %lf.\n", radius);
21   printf("The area is %lf.\n", area);
22   printf("The circumference is %lf.\n", circumference);
```

```
23
24    return 0;
   }
```

```
Enter the radius: 1.2
The radius is 1.200000.
The area is 4.523893.
The circumference is 7.539822.
```

## Dissecting the Program

```
double radius, circumference, area;
double pi = 3.14159265;
```

We declare three `double` variables called `radius`, `circumference` and `area`. A `double` variable, unlike `int`, can hold real number (or floating-point number) such as 1.23 or 4.5e6. We also declare a `double` variable called `pi` and initialize its value to 3.1416.

```
printf("Enter the radius: ");
scanf("%lf", &radius);
```

We use `print()` to put up a prompt message, and `scanf()` to read the user input into variable `radius`. Take note that the `%lf` conversion specifier for `double` (`lf` stands for long float). Also remember to place an ampersand (`&`) before `radius`.

```
area = radius * radius * pi;
circumference = 2.0 * radius * pi;
```

perform the computation.

```
printf("The radius is %lf.\n", radius);
printf("The area is %lf.\n", area);
printf("The circumference is %lf.\n", circumference);
```

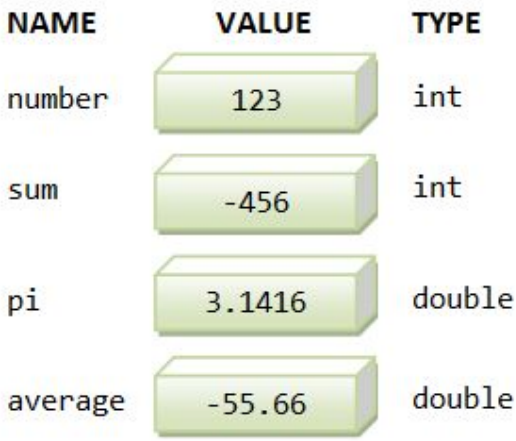
Again, we use `%lf` conversion specifier to print a `double`.

Take note that the programming statements inside the `main()` are executed one after another, sequentially.

## Exercises

1. Follow the above example, write a program to print the area and perimeter of a rectangle. Your program shall prompt the user for the length and width of the rectangle, in `double`s.
2. Follow the above example, write a program to print the surface area and volume of a cylinder. Your program shall prompt the user for the radius and height of the cylinder, in `double`s.

# 7. What is a Variable?




**A variable has a *name*, stores a *value* of the declared *type***

Computer programs *manipulate* (or *process*) data. A *variable* is used to store a piece of data for processing. It is called *variable* because you can change the value stored.

More precisely, a *variable* is a *named* storage location, that stores a *value* of a particular data *type*. In other words, a *variable* has a *name*, a *type* and stores a *value* of that type.

- A variable has a *name* (or *identifier*), e.g., `radius`, `area`, `age`, `height`. The name is needed to uniquely identify and reference a variable, so as to assign a value to the variable (e.g., `radius=1.2`), and retrieve the value stored (e.g., `area = radius*radius*pi`).
- A variable has a *type*. Examples of *type* are:
    - `int`: for integers (whole numbers) such as `123` and `-456`;
    - `double`: for floating-point or real numbers, such as `3.1416`, `-55.66`, `7.8e9`, `1.2e3`, `-4.5e-6` having a decimal point and fractional part, in fixed or scientific notations.
- A variable can store a *value* of the declared *type*. It is important to take note that a variable is associated with a type, and can only store value of that particular type. For example, a `int` variable can store an integer value such as `123`, but NOT real number such as `12.34`, nor texts such as `"Hello"`. The concept of *type* was introduced into the early programming languages to simplify interpretation of data.

The above diagram illustrates 2 types of variables: `int` and `double`. An `int` variable stores an integer (whole number). A `double` variable stores a real number.

To use a variable, you need to first *declare* its *name* and *type*, in one of the following syntaxes:

```
var-type var-name;                    // Declare a variable of a type
var-type var-name-1, var-name-2,...;  // Declare multiple variables of the same type
var-type var-name = initial-value;    // Declare a variable of a type, and assign an initial value
var-type var-name-1 = initial-value-1, var-name-2 = initial-value-2,... ;  // Declare variables with initial values
```

Take note that:

- Each *declaration statement* is terminated with a semi-colon (`;`).
- In multiple-variable declaration, the names are separated by commas (`,`).
- The symbol =, known as the *assignment operator*, can be used to assign an initial value (of the declared type) to the variable.

For example,

```
int sum;                 // Declare a variable named "sum" of the type "int" for storing an integer.
                         // Terminate the statement with a semi-colon.
int number1, number2; // Declare two "int" variables named "number1" and "number2",
                         // separated by a comma.
double average;          // Declare a variable named "average" of the type "double" for storing a real number.
int height = 20;       // Declare an int variable, and assign an initial value.
```

Once a variable is declared, you can *assign* and *re-assign* a value to a variable, via the *assignment operator* "=". For example,

```
int number;           // Declare a variable named "number" of the type "int" (integer)
number = 99;          // Assign an integer value of 99 to the variable "number"
number = 88;          // Re-assign a value of 88 to "number"
number = number + 1;  // Evaluate "number + 1", and assign the result back to "number"
int sum = 0;           // Declare an int variable named sum and assign an initial value of 0
sum = sum + number;   // Evaluate "sum + number", and assign the result back to "sum", i.e. add number into sum
int num1 = 5, num2 = 6;   // Declare and initialize two int variables in one statement, separated by a comma
double radius = 1.5;  // Declare a variable name radius, and initialize to 1.5
int number;           // ERROR: A variable named "number" has already been declared
sum = 55.66;            // WARNING: The variable "sum" is an int. It shall not be assigned a floating-point number
sum = "Hello";          // ERROR: The variable "sum" is an int. It cannot be assigned a text string
```

Take note that:

- Each variable can only be declared once.
- You can declare a variable anywhere inside the program, as long as it is declared before it is being used.
- Once the *type* of a variable is declared, it can only store a value belonging to this particular type. For example, an `int` variable can hold only integer such as `123`, and NOT floating-point number such as `-2.17` or text string such as `"Hello"`.
- The type of a variable cannot be changed inside the program.

### x=x+1?

Assignment (=) in programming is different from *equality* in Mathematics. e.g., "`x=x+1`" is invalid in Mathematics. However, in programming, it means compute the value of `x` plus 1, and assign the result back to variable `x`.

"x+y=1" is valid in Mathematics, but is invalid in programming. In programming, the RHS of "=" has to be evaluated to a value; while the LHS shall be a variable. That is, evaluate the RHS first, then assign to LHS.

Some languages uses `:=` as the assignment operator to avoid confusion with equality.

# 8. Basic Arithmetic Operations

The basic *arithmetic operators* are:

| Operator | Meaning | Example |
|---|---|---|
| + | Addition | x + y |
| - | Subtraction | x - y |
| * | Multiplication | x * y |
| / | Division | x / y |
| % | Modulus (Remainder) | x % y |
| ++ | Increment by 1 (Unary) | ++x or x++ |
| -- | Decrement by 1 (Unary) | --x or x-- |

Addition, subtraction, multiplication, division and remainder are *binary operators* that take two operands (e.g., x + y); while negation (e.g., -x), increment and decrement (e.g., x++, --x) are *unary operators* that take only one operand.

### Example

The following program (`TestArithmetics.c`) illustrates these arithmetic operations.

```
1/*
2 * Test arithmetic operations (TestArithmetics.c)
3 */
4#include <stdio.h>
5
6int main() {
7
8   int number1, number2; // Declare 2 integer variable number1 and number2
9      int  sum,  difference,  product,  quotient,  remainder;  //  declare  5  int
10variables
11
12  // Prompt user for the two numbers
13   printf("Enter two integers (separated by space): ");
14   scanf("%d%d", &number1, &number2);  // Use 2 %d to read 2 integers
15
16  // Do arithmetic Operations
17   sum = number1 + number2;
18   difference = number1 - number2;
19   product = number1 * number2;
20   quotient = number1 / number2;
21   remainder = number1 % number2;
22
23    printf("The  sum,  difference,  product,  quotient and  remainder  of  %d  and  %d
24are %d, %d, %d, %d, %d.\n",
```

```
25          number1, number2, sum, difference, product, quotient, remainder);
26
27   // Increment and Decrement
28    ++number1;    // Increment the value stored in variable number1 by 1
29                  // same as "number1 = number1 + 1"
30    --number2;    // Decrement the value stored in variable number2 by 1
31                  // same as "number2 = number2 - 1"
32    printf("number1 after increment is %d.\n", number1);
33    printf("number2 after decrement is %d.\n", number2);
34
35    quotient = number1 / number2;
36       printf("The  new  quotient  of  %d  and  %d  is  %d.\n",  number1,  number2,
37quotient);

      return 0;
    }
```

```
Enter two integers (separated by space): 98 5
The sum, difference, product, quotient and remainder of 98 and 5 are 103, 93, 49
0, 19, 3.
number1 after increment is 99.
number2 after decrement is 4.
The new quotient of 99 and 4 is 24.
```

## Dissecting the Program

```
int number1, number2;
```

```
int sum, difference, product, quotient, remainder;
```

declare all the `int` (integer) variables `number1`, `number2`, `sum`, `difference`, `product`, `quotient`, and `remainder`, needed in this program.

```
printf("Enter two integers (separated by space): ");
```

```
scanf("%d%d", &number1, &number2);
```

prompt user for two integers and store into `number1` and `number2`, respectively.

```
sum = number1 + number2;
```

```
difference = number1 - number2;
```

```
product = number1 * number2;
```

```
quotient = number1 / number2;
```

```
remainder = number1 % number2;
```

carry out the arithmetic operations on `number1` and `number2`. Take note that division of two integers produces a *truncated* integer, e.g., `98/5` → 19, `99/4` → 24, and `1/2` → `0`.

```
printf("The sum, difference, product, quotient and remainder of %d and %d are
%d, %d, %d, %d, %d.\n",
```

```
   number1, number2, sum, difference, product, quotient, remainder);
```

prints the results of the arithmetic operations, with the appropriate string descriptions in between.

```
++number1;
```

```
--number2;
```

illustrate the increment and decrement operations. Unlike `'+'`, `'-'`, `'*'`, `'/'` and `'%'`, which work on two operands (*binary operators*), `'++'` and `'--'`operate on only one operand (*unary operators*). `++x` is equivalent to `x = x + 1`, i.e., increment x by 1. You may place the increment operator before or after the operand, i.e., `++x` (pre-increment) or `x++` (post-increment). In this example, the effects of pre-increment and post-increment are the same. I shall point out the differences in later section.

## Exercises

1. Introduce one more `int` variable called `number3`, and prompt user for its value. Print the *sum* and *product* of all the three integers.
2. In Mathematics, we could omit the multiplication sign in an arithmetic expression, e.g., `x = 5a + 4b`. In programming, you need to explicitly provide all the operators, i.e., `x = 5*a + 4*b`. Try printing the sum of 31 times of `number1` and 17 times of `number2` and 87 time of `number3`.