TABLE OF CONTENTS (HIDE)

1. Getting Started - Write your Firs
2. C Terminology and Syntax
3. The Process of Writing a C Prog
4. C Program Template
5. Let's Write a Program to Add a
   - 5.1 Example: Adding Two Integer
   - 5.2 Example: Prompting User for
6. What is a Program?
7. What is a Variable?
8. Basic Arithmetic Operations
9. What If Your Need To Add a Th
10. Conditional (or Decision)
11. Type `double` & Floating-Point
12. Mixing `int` and `double`, and
13. Summary

C programming Tutorial

Introduction to C Programming (for Novices & First-Time Programmers)

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/Lunix/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/Lunix/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/Lunix/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/Lunix/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/Lunix/Mac):

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

  // UNIX/Lunix/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`).

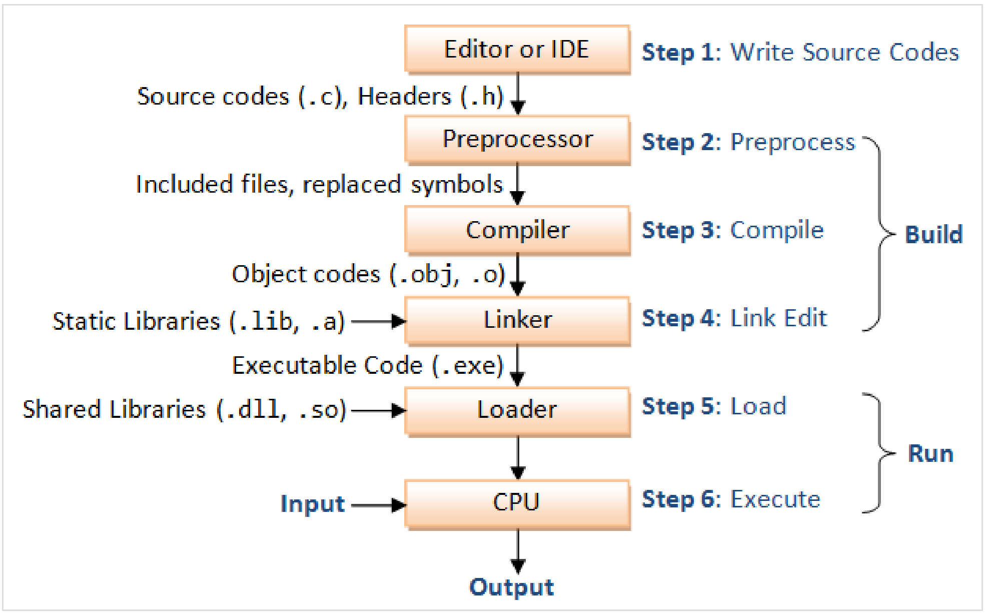
Step 2: Pre-process the source codes according to the *preprocessor directives*. The preprocessor directives begin with a hash sign (`#`), such as `#include` and `#define`. They indicate that certain manipulations (such as including another file or replacement of symbols) are to be performed BEFORE compilation.

Step 3: Compile the preprocessed 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.

```
/*
 * Comment to state the purpose of this program (filename.c)
 */
#include <stdio.h>

int main() {
   // Your Programming statements HERE!

   return 0;
}
```

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

```
/*
 * Add two integers and print their sum (Add2Integers.c)
 */
#include <stdio.h>

int main() {
   int integer1; // Declare a variable named integer1 of the type integer
   int integer2; // Declare a variable named integer2 of the type integer
   int sum;      // Declare a variable named sum of the type integer

   integer1 = 55;   // Assign value to variable integer1
   integer2 = 66;   // Assign value to variable integer1
   sum = integer1 + integer2;   // Compute the sum

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

   return 0;
}
```

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

Disecting 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 `\n` denotes 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

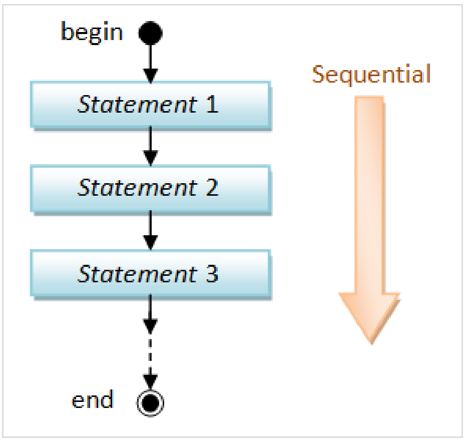
1. Print each of the following patterns. Use one `printf()` statement for each line of outputs. End each line by printing a newline (`\n`).

```
* * * * *       * * * * *     * * * * * *
 * * * * *      *       *      *       *
* * * * *       *       *       *    *
 * * * * *      *       *        * *
* * * * *       * * * * *         *
   (a)             (b)           (c)
```

2. Print the above patterns using ONE `printf()` statement.
3. 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.
4. 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.

```
/*
 * Prompt user for the radius of a circle and compute its area and circumference
 * (CircleComputation.c)
 */
#include <stdio.h>

int main() {
   double radius, circumference, area; // Declare 3 floating-point variables
   double pi = 3.14159265;             // Declare and define PI

   printf("Enter the radius: ");  // Prompting message
   scanf("%lf", &radius);         // Read input into variable radius

   // Compute area and circumference
   area = radius * radius * pi;
   circumference = 2.0 * radius * pi;

   // Print the results
   printf("The radius is %lf.\n", radius);
   printf("The area is %lf.\n", area);
   printf("The circumference is %lf.\n", circumference);

   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);
```

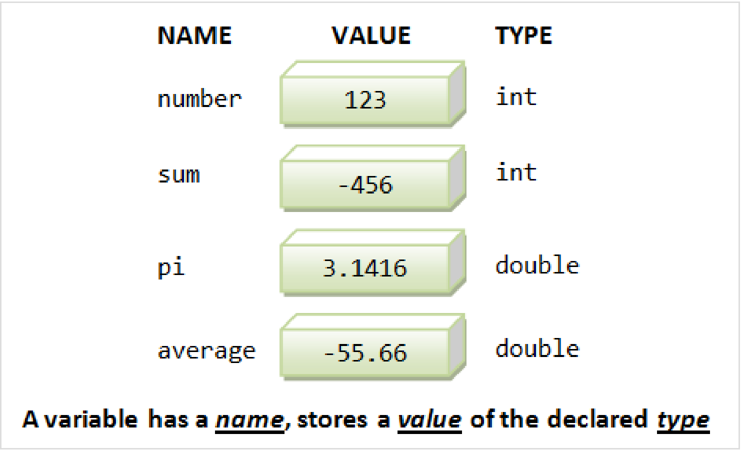
Again, we use `%lf` conversion specifier to print a `double`.

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.

```
/*
 * Test arithemtic operations (TestArithmetics.c)
 */
#include <stdio.h>

int main() {

   int number1, number2;  // Declare 2 integer variable number1 and number2
   int sum, difference, product, quotient, remainder;  // declare 5 int variables

   // Prompt user for the two numbers
   printf("Enter two integers (separated by space): ");
   scanf("%d%d", &number1, &number2);  // Use 2 %d to read 2 integers

```

```
    // Do arithmetic Operations
    sum = number1 + number2;
    difference = number1 - number2;
    product = number1 * number2;
    quotient = number1 / number2;
    remainder = number1 % number2;

    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);

    // Increment and Decrement
    ++number1;    // Increment the value stored in variable number1 by 1
                  // same as "number1 = number1 + 1"
    --number2;    // Decrement the value stored in variable number2 by 1
                  // same as "number2 = number2 - 1"
    printf("number1 after increment is %d.\n", number1);
    printf("number2 after decrement is %d.\n", number2);

    quotient = number1 / number2;
    printf("The new quotient of %d and %d is %d.\n", number1, number2, quotient);

    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`.

9. What If Your Need To Add a Thousand Numbers? Use a Loop!

Suppose that you want to add all the integers from 1 to 1000. If you follow the previous examples, you would require a thousand-line program! Instead, you could use a *loop* in your program to perform a *repetitive* task, that is what the dumb computers are good at.

Example

Try the following program `SumNumbers.c`, which sums all the integers from 1 to an upperbound provided by the user, using a so-called *while-loop*.

```
 1  /*
 2   * Sum from 1 to an upperbound using a while-loop (SumNumbers.c).
 3   */
 4  #include <stdio.h>
 5  
 6  int main() {
 7     int sum = 0;     // Declare an int variable sum to accumulate the numbers
 8                      // Set the initial sum to 0
 9     int upperbound; // Sum from 1 to this upperbound
10  
11     // Prompt user for an upperbound
12     printf("Enter the upperbound: ");
13     scanf("%d", &upperbound);  // Use %d to read an int
14  
15     // Use a loop to repeatedly add 1, 2, 3,..., up to upperbound
16     int number = 1;
17     while (number <= upperbound) {
18        sum = sum + number;    // accumulate number into sum
19        ++number;              // increment number by 1
20     }
21     // Print the result
22     printf("The sum from 1 to %d is %d.\n", upperbound, sum);
23  
24     return 0;
25  }
```

```
Enter the upperbound: 1000
The sum from 1 to 1000 is 500500.
```

Dissecting the Program

```
int sum = 0;
```

declares an `int` variable named `sum` and initializes it to 0. This variable will be used to *accumulate* numbers over the steps in the repetitive loop.

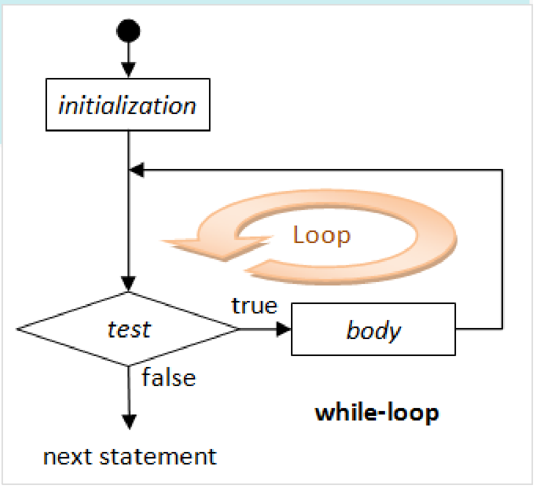
```
printf("Enter the upperbound: ");
scanf("%d", &upperbound);
```

prompt user for an upperbound to sum.

```
int number = 1;
while (number <= upperbound) {
   sum = sum + number;
   ++number;
}
```

This is the so-called *while-loop*. A *while-loop* takes the following syntax:

```
initialization-statement;
while (test) {
   loop-body;
}
next-statement;
```



As illustrated in the flow chart, the *initialization* statement is first executed. The *test* is then checked. If the *test* is true, the *body* is executed. The *test* is checked again and the process repeats until the *test* is false. When the *test* is false, the loop completes and program execution continues to the *next statement* after the loop.

In our program, the *initialization* statement declares an `int` variable named `number` and initializes it to 1. The *test* checks if `number` is equal to or less than the `upperbound`. If it is true, the current value of `number` is added into the `sum`, and the statement `++number` increases the value of `number` by 1. The *test* is then checked again and the process repeats until the *test* is false (i.e., `number` increases to `upperbound+1`), which causes the loop to terminate. Execution then continues to the next statement (in Line 22).

In this example, the loop repeats `upperbound` times. After the loop is completed, Line 22 prints the result with a proper description.

Exercises

1. Modify the above program to sum all the number between a *lowerbound* and an *upperbound* provided by the user.
2. Modify the above program to sum all the *odd* numbers between `1` to an *upperbound*. (*Hint*: Use "`number = number + 2`".)
3. Modify the above program to sum all the numbers between `1` to an *upperbound* that are divisible by `7`. (*Hint*: Use "`number = number + 7`")
4. Modify the above program to find the sum of the *square* of all the numbers from `1` to an *upperbound*, i.e. `1*1 + 2*2 + 3*3 +...`
5. Modify the above program to compute the *product* of all the numbers from `1` to `10`. (*Hint*: Use a variable called `product` instead of `sum` and initialize `product` to 1. *Ans*: `3628800`.) Based on this code, write a program to display the *factorial* of *n*, where *n* is an integer between `1` to `12`.

10. Conditional (or Decision)

What if you want to sum all the odd numbers and also all the even numbers between 1 and 1000? There are many way to do this. You could declare two variables: `sumOdd` and `sumEven`. You can then use a *conditional statement* to check whether the number is odd or even, and accumulate the number into the respective sum. The program `SumOddEven.c` is as follows:

```c
/*
 * Sum the odd and even numbers from 1 to an upperbound (SumOddEven.c)
 */
#include <stdio.h>

int main() {
   int sumOdd  = 0; // For accumulating odd numbers, init to 0
   int sumEven = 0; // For accumulating even numbers, init to 0
   int upperbound;  // Sum from 1 to this upperbound

   // Prompt user for an upperbound
   printf("Enter the upperbound: ");
   scanf("%d", &upperbound);   // Use %d to read an int

   // Use a loop to repeatedly add 1, 2, 3,..., up to upperbound
   int number = 1;
   while (number <= upperbound) {
      if (number % 2 == 0) {  // even number
         sumEven = sumEven + number;
      } else {                // odd number
         sumOdd = sumOdd + number;
      }
      ++number; // increment number by 1
   }

   // Print the results
   printf("The sum of odd numbers is %d.\n", sumOdd);
   printf("The sum of even numbers is %d.\n", sumEven);
   printf("The difference is %d.\n", (sumOdd - sumEven));

   return 0;
}
```

```
Enter the upperbound: 10000
The sum of odd numbers is 25000000.
The sum of even numbers is 25005000.
The difference is -5000.
```

Dissecting the Program

```c
int sumOdd = 0;
int sumEven = 0;
```

declare two `int` variables named `sumOdd` and `sumEven` and initialize them to 0, for accumulating the odd and even numbers, respectively.

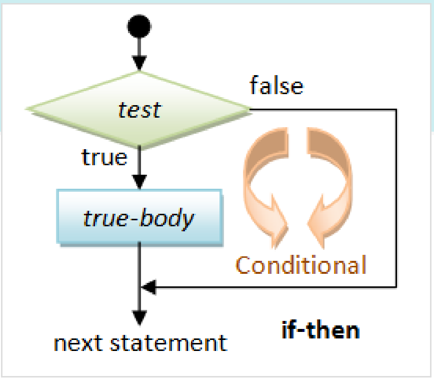
```c
if (number % 2 == 0) {
   sumEven = sumEven + number;
} else {
   sumOdd = sumOdd + number;
}
```

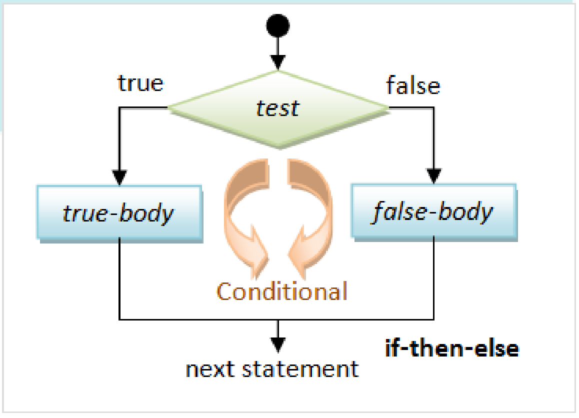
This is a *conditional statement*. The conditional statement can take one these forms: *if-then* or *if-then-else*.

```
// if-then
if ( test ) {
   true-body;
}
// if-then-else
```

```
if ( test ) {
   true-body;
} else {
   false-body;
}
```

For a *if-then* statement, the *true-body* is executed if the *test* is true. Otherwise, nothing is done and the execution continues to the next statement. For a *if-then-else* statement, the *true-body* is executed if the *test* is true; otherwise, the *false-body* is executed. Execution is then continued to the next statement.





In our program, we use the *remainder operator* `(%)` to compute the remainder of `number` divides by `2`. We then compare the remainder with `0` to test for even number.

Comparison Operators

There are six comparison (or relational) operators:

Operator	Meaning	Example
`==`	Equal to	`x == y`
`!=`	Not equal to	`x != y`
`>`	Greater than	`x > y`
`>=`	Greater than or equal to	`x >= y`
`<`	Less than	`x < y`
`<=`	Less than or equal to	`x <= y`

Take note that the comparison operator for equality is a double-equal sign `(==)`; whereas a single-equal sign `(=)` is the assignment operator.

Combining Simple Conditions

Suppose that you want to check whether a number `x` is between `1` and `100` (inclusive), i.e., `1 <= x <= 100`. There are two *simple conditions* here, `(x >= 1) AND (x <= 100)`. In programming, you cannot write `1 <= x <= 100`, but need to write `(x >= 1) && (x <= 100)`, where "`&&`" denotes the "`AND`" operator. Similarly, suppose that you want to check whether a number `x` is divisible by 2 OR by 3, you have to write `(x % 2 == 0) || (x % 3 == 0)` where "`||`" denotes the "`OR`" operator.

There are three so-called *logical operators* that operate on the *boolean* conditions:

Operator	Meaning	Example
`&&`	Logical AND	`(x >= 1) && (x <= 100)`
`||`	Logical OR	`(x < 1) || (x > 100)`
`!`	Logical NOT	`!(x == 8)`

For examples:

```
// Return true if x is between 0 and 100 (inclusive)
(x >= 0) && (x <= 100)  // AND (&&)
// Incorrect to use 0 <= x <= 100

// Return true if x is outside 0 and 100 (inclusive)
(x < 0) || (x > 100)       // OR (||)
!((x >= 0) && (x <= 100))  // NOT (!), AND (&&)

// Return true if "year" is a leap year
// A year is a leap year if it is divisible by 4 but not by 100, or it is divisible by 400.
((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
```

Exercises

1. Write a program to sum all the integers between 1 and 1000, that are divisible by 13, 15 or 17, but not by 30.
2. Write a program to print all the leap years between AD1 and AD2010, and also print the number of leap years. (Hints: use a variable called `count`, which is initialized to zero. Increment the `count` whenever a leap year is found.)

11. Type `double` & Floating-Point Numbers

Recall that a *variable* in C has a *name* and a *type*, and can hold a *value* of only that particular *type*. We have so far used a type called `int`. A `int` variable holds only integers (whole numbers), such as `123` and `-456`.

In programming, real numbers such as `3.1416` and `-55.66` are called *floating-point numbers*, and belong to a type called `double`. You can express floating-point numbers in *fixed notation* (e.g., `1.23`, `-4.5`) or *scientific notation* (e.g., `1.2e3`, `-4E5.6`) where `e` or `E` denote the exponent of base 10.

Example

```
/*
 * Convert temperature between Celsius and Fahrenheit
 * (ConvertTemperature.c)
 */
#include <stdio.h>

int main() {
   double celsius, fahrenheit;

   printf("Enter the temperature in celsius: ");
   scanf("%lf", &celsius);  // Use %lf to read an double
   fahrenheit = celsius * 9.0 / 5.0 + 32.0;
   printf("%.2lf degree C is %.2lf degree F.\n\n", celsius, fahrenheit);
      // %.2lf prints a double with 2 decimal places

   printf("Enter the temperature in fahrenheit: ");
   scanf("%lf", &fahrenheit);
   celsius =  (fahrenheit - 32.0) * 5.0 / 9.0;
   printf("%.2lf degree F is %.2lf degree C.\n\n", fahrenheit, celsius);

   return 0;
}
```

```
Enter the temperature in celsius: 37.2
37.20 degree C is 98.96 degree F.

Enter the temperature in fahrenheit: 100
100.00 degree F is 37.78 degree C.
```

12. Mixing `int` and `double`, and Type Casting

Although you can use a `double` to keep an integer value (e.g., `double count = 5`), you should use an `int` for integer. This is because `int` is far more efficient than `double`, in terms of running times and memory requirement.

At times, you may need both `int` and `double` in your program. For example, keeping the *sum* from `1` to `100` (`=5050`) as an `int`, and their *average* `50.5` as a `double`. You need to be *extremely careful* when different types are mixed.

It is important to note that:

- Arithmetic operations (`'+'`, `'-'`, `'*'`, `'/'`) of two `int`'s produce an `int`; while arithmetic operations of two `double`'s produce a `double`. Hence, `1/2 → 0` (take note!) and `1.0/2.0 → 0.5`.
- Arithmetic operations of an `int` and a `double` produce a `double`. Hence, `1.0/2 → 0.5` and `1/2.0 → 0.5`.

You can assign an integer value to a `double` variable. The integer value will be converted to a double value automatically, e.g., `3 → 3.0`. For example,

```
int i = 3;
double d;
d = i;              // 3 → 3.0, d = 3.0
d = 88;             // 88 → 88.0, d = 88.0
double nought = 0;  // 0 → 0.0; there is a subtle difference between int of 0 and double of 0.0
```

However, if you assign a `double` value to an `int` variable, the *fractional* part will be lost. For example,

```
double d = 55.66;
int i;
i = d;   // i = 55 (truncated)
```

Some C compilers signal a warning for truncation, while others do not. You should study the "warning messages" (if any) carefully - which signals a potential problem in your program, and rewrite the program if necessary. C allows you to ignore the warning and run the program. But, the fractional part will be lost during the execution.

Type Casting Operators

If you are certain that you wish to carry out the type conversion, you could use the so-called *type cast* operator. The type cast operation returns an equivalent value in the *new-type* specified.

```
(new-type)expression;
```

For example,

```
double d = 5.5;
int i;
i = (int)d;         // (int)d -> (int)5.5 -> 5
i = (int)3.1416;    // 3
```

Similarly, you can *explicitly* convert an `int` value to `double` by invoking type-casting operation too.

Example

Try the following program and explain the outputs produced:

```
/*
 * Testing type cast (TestCastingAverage.c)
 */
#include <stdio.h>

int main() {
   int sum = 0;       // Sum in "int"
   double average;    // average in "double"

   // Compute the sum from 1 to 100 (in "int")
   int number = 1;
   while (number <= 100) {
      sum = sum + number;
      ++number;
   }
   printf("The sum is %d.\n", sum);

   // Compute the average (in "double")
   average = sum / 100;
   printf("Average 1 is %lf.\n", average);
   average = (double)sum / 100;
   printf("Average 2 is %lf.\n", average);
   average = sum / 100.0;
   printf("Average 3 is %lf.\n", average);
   average = (double)(sum / 100);
   printf("Average 4 is %lf.\n", average);

   return 0;
}
```

```
The sum is 5050.
Average 1 is 50.000000.  <== incorrect
Average 2 is 50.500000.
Average 3 is 50.500000.
Average 4 is 50.000000.  <== incorrect
```

The first average is incorrect, as `int/int` produces an `int` (of `50`).

For the second average, the value of `sum` (of `int`) is first converted to `double`. Subsequently, `double/int` produces `double`.

For the third average, `int/double` produces `double`.

For the fourth average, `int/int` produces an `int` (of `50`), which is then casted to `double` (of `50.0`) and assigned to `average` (of `double`).

Exercises

1. Write a program called `HarmonicSeriesSum` to compute the sum of a harmonic series `1 + 1/2 + 1/3 + 1/4 + .... + 1/`*n*, where *n* `= 1000`. Your program shall prompt user for the value of *n*. Keep the sum in a `double` variable, and take note that `1/2` gives `0` but `1.0/2` gives `0.5`.
   Try computing the sum for *n*=`1000`, `5000`, `10000`, `50000`, `100000`.
   Hints:

   ```
   /*
    * Sum harmonics Series (HarmonicSeriesSum.c)
    */
   #include <stdio.h>

   int main() {
      int maxDenominator;   // max denominator to sum to
      double sum = 0.0;  // For accumulating sum in double

      // Prompt user for the maxDenominator
      ......

      int denominator = 1;
   ```

```
      while (denominator <= maxDenominator) {
         // Beware that int/int gives int
         ......
         ++denominator;  // next
      }
      // Print the sum
      ......
   }
```

2. Write a program called `GeometricSeriesSum` to compute the sum of a geometric series `1 + 1/2 + 1/4 + 1/8 + .... + 1/`*n*. You program shall prompt for the value of *n*. (Hints: Use post-processing statement of `denominator = denominator * 2`.)

13. Summary

I have presented the basics for you to get start in programming. To learn programming, you need to understand the syntaxes and features involved in the programming language that you chosen, and you have to practice, practice and practice, on as many problems as you could.

Link to "C Language References & Resources"

Latest version tested: GCC 4.6.2
Last modified: June, 2013