Chapter 1: A Tutorial Introduction

The purpose of this tutorial is to show-case the language in order to get the user started. It does not represent in any way or form a standard in which the programs are expected to be written. Only the basis are presented such as: variables and constants, arithmetic, control flow, functions, and the rudiments of input and output.

Some of C's key features won't be convered in this chapter. That includes and is not limited to: pointers, structures, most of C's rich set of operators, several control-flow statements and standard library.

1.1 Getting Started

As always, the first program would be a 'Hello World!'.

Program: Hello World!

Every C program must end in ".c". To compile it, use the command `cc hello_world.c`. This will produce a compiled file named **a.out**.

Another rule is that every C program must have a **main** function. This is the place where the program start executing.

The first line of the program (**#include <stdio.h>**) tells the compiler to include information from this library. In this case, the standar input/output library.

In our example, we called the function `printf` from standard library and by passing the argument "Hello, World!\n" we managed to print it to standard output.

The character '\n' represents a single character. An **escape sequence** like '\n' provides a general and extensible mechanism for representing hard-to-type or invisible characters. Among the otehrs that C provides are '\t' for tab, '\b' for backspace, '\"' for the double quote, and '\' for the backslash itself.

1.2 Variables and Arithmetic Expressions

The next program uses the formula `C = (5/9)(F - 32)` to print the following table of Fahrenheit temperatures and their centigrade or Celsius equivalents:

Program: Fahrenheit-Celsius

The sizes of C data types are machine independent as they depend directly on the CPU.

NOTE: if a control flow structure contains a single statement, it can also be written as:

```
while (i < j)
    i = 2 * j;
```

The recommandation is to write one statement per line, and use blanks around operators for grouping.

The program has several problems:

1. The temperature is not very accurate because by divind two integers, the output is truncated
2. The alignment is not correct, as the numbers are all over the place

Here is the second version of the program:

Program: Fahrenheit-Celsius: version 2

We were unable to use 5/9 in the previous version because integer division would truncate it to zero. A decimal point in a constant indicates that it is floating point, however, so 5.0/9.0 is ont truncated because it is the ratio of two floating-point values.

Even though conversion are made automatically if one of the numbers is a float number, it is easier to read when applying a decimal point.

The printf conversion specification **%3.0f** says that a floating-point number is to be printed at least three characters wide, with no decimal point and no fraction digits. **%6.1f** describes another number that is to be printed at least six characters wide, with 1 digit after the decimal point.

Width and precision may be omitted from a specification: **%6f** says that the number is to be at least six characters wide; **%.2f** specifies two characters after the decimal point, but the width is not constrained; and **%f** merely says to print the number as a floating point.

Notes:

- %d : print as decimal integer
- %6d : print as decimal integer, at least 6 characters wide
- %f : print as floating point
- %6f : print as floating point, at least 6 characters wide
- %.2f : print as floating point, 2 characters after decimal point
- %6.2f : print as floating point, at least 6 wide and 2 after decimal point

Among others, `printf` also recognizes %o for octal, %x for hexadecimal, %c for character, %s for character string, and %% for % itself.

1.3 The For Statement

There are plenty of different ways of writting a program. For example, the temperature program. This is almost a one liner:

Program*: Fahrenheit-Celsius: Version 3 - For loop

1.5 Symbolic Constants

It's bad practice to bury "magic numbers" like 300 and 20 in a program; they convey little information to someone who might have to read the program later, and they are hard to change in a systematic way. One way to deal with magic numbers is to five them meaningful names. A `**#define**` line defines a symbolic name or **symbolic constant** to be a particular string of characters:

```
#define     name    replacement text
```

Thereafter, any occurrence of name (not in quotes and not part of another name) will be replaced by the corresponding **replacement text**. The **name** has the same form as a variable name: a sequence of letters and digits that begins with a letter. The **replacement text** can be any sequence of characters; it is not limited to numbers.

Program*: Fahrenheit-Celsius: Version 4 - SYMBOLIC CONSTANTS

The quantites LOWER, UPPER and STEP are symbolic constants, not variables, so they do not appear in declarations. Symbolic constant names are conventionally written in upper case so they can be readily distinguished from lower case variable name. Notice that there is no semicolon at the end of a **#define** line.

1.5 Character Input and Output

The Standard Library provides several functions for reading or writing one characyer at a time, of which **getchar** and **putchar** are the simplest. Each time it is called, **getchar** reads the *next input characyer* from a text stream and returns that as its value. That is, after `c = getchar()` the variable `c` contains the next character of input. The characters normally come from the keyboard.

The function `putchar` prints the character each time it is called `putchar(c)` priunts the contents of the integer variable `c` as a character, usually on the screen. Calls to `putchar` and `printf` may be interleaved; the output will appear in the order in which the calls are made.

1.5.1 File Copying

Given `getchar` and `putchar`, you can write a surprising amount of useful code without knowing anything more about input and output. The simplest example is a program that copies its input to its output one character at a time:

Program*: Input to Output

The problem is distinguishing the end of the input from valid data. The solution is that `getchar` returns a distinctive value when there is no more input, a value that cannot be confused with any real character. This value is called "EOF" for "end of file". We must decalre `c` to be a type big enough to gold EOF in addition to any possible `char`. Therefore we must use `int`.

EOF is an integer defined in `<stdio.h>`, but the specific numeric value doesn't matter as long as it is not the same as any `char` value. By using the symbolic constant, we are assured that nothing in the program depends on the specific numeric value.

In C, any assignment, such as `c = getchar()` is an expression and has a value, which is the value of the left hand side adter the assignment. This means that an assignment can appear as part of a larger expression. If the assignment of a character to `c` is put inside the test part of a `while` loop, the copy program can be written this way:

Program*: Input to Output - version 2

This version centralizes the input - there is now only one reference to `getchar` - and shrinks the program. The resulting program is more compact, and, once the idiom is mastered, easier to read.

1.5.2 Character Counting

The next program counts characters; it is similar to the copy program.

Program*: Character counter

The caracter counting program accumulates its count in a `long` variable instead of an `int`. `long` integers are at least 32 bits. Although on some machines, `int` and `long` are the same size, on others an `int` is 16 bits, with a maximum value of 32767, and it would take relatively little input to overflow an `int` counter.

It may be possible to cope with even bigger numbers by using a `double` (double precision `float`). We will also use a `for` statement instead of a `while`, to illustrate anothe way to write the loop.

Program*: Character counter - version2

`printf` uses %f for both float and double; %.0f suppresses printing of the decimal point and the fraction part, which is zero.

The body of this `for` loop is empty, because all of the work is done in the test and increment parts. But the grammatical rules of C require that a `for` statement have a body. The isolated semicolon, called a `null statement`, is there to satisfy that requirement. We put it on a separate line to make it visible.

Before we leave the character count program, obeserve that if the input contains no charcters, the while or for test fails on the very first call to `getchar`, an the program produces zero, the right answer. This is important. One of the nice things about while and for is that they test at the top of the loop, before proceeding with the body. If there is nothing to do, nothing is done, even if that means never going thourgh the loop body. Programs should act intelligently when given zero-length input. The while and for statements help ensure that programs do reasonable things with boundary conditions.

1.5.3 Line Counting

The next program counts input lines. As we mentioned above, the standard library ensures that an input text stream appears as a sequence of lines, each terminated by a newline. Hence, counting lines is just counting newlines:

Program*: Line counter

A character written between single quotes represents an integer value equal to the numerical value of the character in the machines's character set. This is called a `character constant`, although it is just another way to write a small integer. So, for example, 'A' is a character constant; in the ASCII character set its value is 65, the internal representation of the character **A**.

The escape sequences used in string constants are also legal in character constants, so '\n' stand for the value of the newline character, which is 10 in ASCII. You should note carefully that '\n' is a single character, and in expressions is just an integer; on the other hand, "\n" is a string constant that happens to contain only one character.

1.5.4 Word Counting

The fourth in our series of useful programs counts lines, words, and characters, with the loose definition that a word is any sequence of characters that does not contain a blank, tab or newline. This is a bare-bones version of the UNIX program `wc`.

Program*: wc program

1.6 Arrays

Let us write a program to count the number of occurences of each digit, of white space chracters (blank, tab newline), and of all other characters. This is artificial, but it permits us to illustrate several aspects of C in one program.

There are twelve categories of input, so it is convenient to use an array to hold the number of occurrences of each digit, rather than ten individual variables. Here is one version of the program:

Program*: counting

1.7 Functions