Chapter 1 - A Tutorial Introduction

Let us begin with a quick introduction in C. Our aim is to show the essential elements of the language in real programs, but without getting bogged down in details, rules, and exceptions. At this point, we are not trying to be complete or even precise (save that the examples are meant to be correct). We want to get you as quickly as possible to the point where you can write useful programs, and to do that we have to concentrate on the basics: variables and constants, arithmetic, control flow, functions, and the rudiments of input and output. We are intentionally leaving out of this chapter features of C that are important for writing bigger programs. These include pointers, structures, most of C's rich set of operators, several control-flow statements, and the standard library.

This approach and its drawbacks. Most notable is that the complete story on any particular feature is not found here, and the tutorial, by being brief, may also be misleading. And because the examples do not use the full power of C, they are not as concise and elegant as they might be. We have tried to minimize these effects, but be warned. Another drawback is that later chapters will necessarily repeat some of this chapter. We hope that the repetition will help you more than it annoys.

In any case, experienced programmers should be able to extrapolate from the material in this chapter to their own programming needs. Beginners should supplement it by writing small, similar programs of their own. Both groups can use it as a framework on which to hang the more detailed descriptions that begin in Chapter 2.

1.1 Getting Started

The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:

Print the words

```
hello, world
```

This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.

In C, the program to print ``hello, world'' is

```
#include <stdio.h>

main()
{
  printf("hello, world\n");
}
```

Just how to run this program depends on the system you are using. As a specific example, on the UNIX operating system you must create the program in a file whose name ends in ``.c'', such as `hello.c`, then compile it with the command

```
cc hello.c
```

If you haven't botched anything, such as omitting a character or misspelling something, the compilation will proceed silently, and make an executable file called `a.out`. If you run `a.out` by typing the command

```
a.out
```

it will print

```
hello, world
```

On other systems, the rules will be different; check with a local expert.

Now, for some explanations about the program itself. A C program, whatever its size, consists of *functions* and *variables*. A function contains *statements* that specify the computing operations to be done, and variables store values used during the computation. C functions are like the subroutines and functions in Fortran or the procedures and functions of Pascal. Our example is a function named `main`. Normally you are at liberty to give functions whatever names you like, but ``main" is special - your program begins executing at the beginning of main. This means that every program must have a `main` somewhere.

`main` will usually call other functions to help perform its job, some that you wrote, and others from libraries that are provided for you. The first line of the program,

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

tells the compiler to include information about the standard input/output library; the line appears at the beginning of many C source files. The standard library is described in Chapter 7 and Appendix B.

One method of communicating data between functions is for the calling function to provide a list of values, called *arguments*, to the function it calls. The parentheses after the function name surround the argument list. In this example, `main` is defined to be a function that expects no arguments, which is indicated by the empty list `( )`.

```
#include <stdio.h>                  include information about standard
library
main()                              define a function called main
                                    that received no argument values
{                                   statements of main are enclosed in braces
    printf("hello, world\n");       main calls library function printf
                                    to print this sequence of characters
}                                   \n represents the newline character
```

The first C program

The statements of a function are enclosed in braces `{ }`. The function `main` contains only one statement,

```
printf("hello, world\n");
```

A function is called by naming it, followed by a parenthesized list of arguments, so this calls the function `printf` with the argument `"hello, world\n"`. `printf` is a library function that prints output, in this case the string of characters between the quotes.

A sequence of characters in double quotes, like `"hello, world\n"`, is called a *character string* or *string constant*. For the moment our only use of character strings will be as arguments for `printf` and other functions.

The sequence `\n` in the string is C notation for the *newline character*, which when printed advances the output to the left margin on the next line. If you leave out the `\n` (a worthwhile experiment), you will find that there is no line advance after the output is printed. You must use `\n` to include a newline character in the `printf` argument; if you try something like

```
printf("hello, world
");
```

the C compiler will produce an error message.

`printf` never supplies a newline character automatically, so several calls may be used to build up an output line in stages. Our first program could just as well have been written

```
#include <stdio.h>

main()
{
  printf("hello, ");
  printf("world");
  printf("\n");
}
```

to produce identical output.

Notice that `\n` represents only a single character. An *escape sequence* like `\n` provides a general and extensible mechanism for representing hard-to-type or invisible characters. Among the others that C provides are `\t` for tab, `\b` for backspace, `\"` for the double quote and `\\` for the backslash itself. There is a complete list in Section 2.3.

Exercise 1-1. Run the ``hello, world" program on your system. Experiment with leaving out parts of the program, to see what error messages you get.

Exercise 1-2. Experiment to find out what happens when `prints`'s argument string contains *c*, where *c* is some character not listed above.

1.2 Variables and Arithmetic Expressions

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

```
1    -17
20   -6
40   4
60   15
80   26
100  37
120  48
140  60
160  71
180  82
200  93
220  104
240  115
260  126
280  137
300  148
```

The program itself still consists of the definition of a single function named `main`. It is longer than the one that printed ``hello, world'', but not complicated. It introduces several new ideas, including comments, declarations, variables, arithmetic expressions, loops , and formatted output.

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
    for fahr = 0, 20, ..., 300 */
main()
{
  int fahr, celsius;
  int lower, upper, step;

  lower = 0;      /* lower limit of temperature scale */
  upper = 300;    /* upper limit */
  step = 20;      /* step size */

  fahr = lower;
  while (fahr <= upper) {
      celsius = 5 * (fahr-32) / 9;
      printf("%d\t%d\n", fahr, celsius);
      fahr = fahr + step;
  }
}
```

The two lines

```
/* print Fahrenheit-Celsius table
    for fahr = 0, 20, ..., 300 */
```

are a *comment*, which in this case explains briefly what the program does. Any characters between `/*` and `*/` are ignored by the compiler; they may be used freely to make a program easier to understand. Comments may appear anywhere where a blank, tab or newline can.

In C, all variables must be declared before they are used, usually at the beginning of the function before any executable statements. A *declaration* announces the properties of variables; it consists of a name and a list of variables, such as

```
int fahr, celsius;
int lower, upper, step;
```

The type `int` means that the variables listed are integers; by contrast with `float`, which means floating point, i.e., numbers that may have a fractional part. The range of both `int` and `float` depends on the machine you are using; 16-bits `int`s, which lie between -32768 and

+32767, are common, as are 32-bit `int`s. A `float` number is typically a 32-bit quantity, with at least six significant digits and magnitude generally between about 10^{-38} and 10^{38}.

C provides several other data types besides `int` and `float`, including:

`char`	character - a single byte
`short`	short integer
`long`	long integer
`double`	double-precision floating point

The size of these objects is also machine-dependent. There are also *arrays*, *structures* and *unions* of these basic types, *pointers* to them, and *functions* that return them, all of which we will meet in due course.

Computation in the temperature conversion program begins with the *assignment statements*

```
lower = 0;
upper = 300;
step = 20;
```

which set the variables to their initial values. Individual statements are terminated by semicolons.

Each line of the table is computed the same way, so we use a loop that repeats once per output line; this is the purpose of the `while` loop

```
while (fahr <= upper) {
    ...
}
```

The `while` loop operates as follows: The condition in parentheses is tested. If it is true (`fahr` is less than or equal to `upper`), the body of the loop (the three statements enclosed in braces) is executed. Then the condition is re-tested, and if true, the body is executed again. When the test becomes false (`fahr` exceeds `upper`) the loop ends, and execution continues at the statement that follows the loop. There are no further statements in this program, so it terminates.

The body of a `while` can be one or more statements enclosed in braces, as in the temperature converter, or a single statement without braces, as in

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

In either case, we will always indent the statements controlled by the `while` by one tab stop (which we have shown as four spaces) so you can see at a glance which statements are inside the loop. The indentation emphasizes the logical structure of the program. Although C compilers do not care about how a program looks, proper indentation and spacing are critical in making programs easy for people to read. We recommend writing only one statement per line, and using blanks around operators to clarify grouping. The position of braces is less important, although people hold passionate beliefs. We have chosen one of several popular styles. Pick a style that suits you, then use it consistently.

Most of the work gets done in the body of the loop. The Celsius temperature is computed and assigned to the variable `celsius` by the statement

```
celsius = 5 * (fahr-32) / 9;
```

The reason for multiplying by 5 and dividing by 9 instead of just multiplying by `5/9` is that in C, as in many other languages, integer division *truncates*: any fractional part is discarded. Since `5` and `9` are integers. `5/9` would be truncated to zero and so all the Celsius temperatures would be reported as zero.

This example also shows a bit more of how `printf` works. `printf` is a general-purpose output formatting function, which we will describe in detail in Chapter 7. Its first argument is a string of characters to be printed, with each `%` indicating where one of the other (second, third, ...) arguments is to be substituted, and in what form it is to be printed. For instance, `%d` specifies an integer argument, so the statement

```
printf("%d\t%d\n", fahr, celsius);
```

causes the values of the two integers `fahr` and `celsius` to be printed, with a tab (`\t`) between them.

Each `%` construction in the first argument of `printf` is paired with the corresponding second argument, third argument, etc.; they must match up properly by number and type, or you will get wrong answers.

By the way, `printf` is not part of the C language; there is no input or output defined in C itself. `printf` is just a useful function from the standard library of functions that are normally accessible to C programs. The behaviour of `printf` is defined in the ANSI standard, however, so its properties should be the same with any compiler and library that conforms to the standard.

In order to concentrate on C itself, we don't talk much about input and output until chapter 7. In particular, we will defer formatted input until then. If you have to input numbers, read the discussion of the function `scanf` in Section 7.4. `scanf` is like `printf`, except that it reads input instead of writing output.

There are a couple of problems with the temperature conversion program. The simpler one is that the output isn't very pretty because the numbers are not right-justified. That's easy to fix; if we augment each `%d` in the `printf` statement with a width, the numbers printed will be right-justified in their fields. For instance, we might say

```
printf("%3d %6d\n", fahr, celsius);
```

to print the first number of each line in a field three digits wide, and the second in a field six digits wide, like this:

```
  0     -17
 20      -6
 40       4
 60      15
 80      26
100      37
...
```

The more serious problem is that because we have used integer arithmetic, the Celsius temperatures are not very accurate; for instance, 0°F is actually about -17.8°C, not -17. To get more accurate answers, we should use floating-point arithmetic instead of integer. This requires some changes in the program. Here is the second version:

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
    for fahr = 0, 20, ..., 300; floating-point version */
main()
{
  float fahr, celsius;
  float lower, upper, step;

  lower = 0;      /* lower limit of temperatuire scale */
  upper = 300;    /* upper limit */
  step = 20;      /* step size */

  fahr = lower;
  while (fahr <= upper) {
      celsius = (5.0/9.0) * (fahr-32.0);
      printf("%3.0f %6.1f\n", fahr, celsius);
      fahr = fahr + step;
  }
}
```

This is much the same as before, except that `fahr` and `celsius` are declared to be `float` and the formula for conversion is written in a more natural way. 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 not truncated because it is the ratio of two floating-point values.

If an arithmetic operator has integer operands, an integer operation is performed. If an arithmetic operator has one floating-point operand and one integer operand, however, the integer will be converted to floating point before the operation is done. If we had written `(fahr-32)`, the `32` would be automatically converted to floating point. Nevertheless, writing floating-point constants with explicit decimal points even when they have integral values emphasizes their floating-point nature for human readers.

The detailed rules for when integers are converted to floating point are in Chapter 2. For now, notice that the assignment

```
fahr = lower;
```

and the test

```
while (fahr <= upper)
```

also work in the natural way - the `int` is converted to `float` before the operation is done.

The `printf` conversion specification `%3.0f` says that a floating-point number (here `fahr`) is to be printed at least three characters wide, with no decimal point and no fraction digits. `%6.1f` describes another number (`celsius`) that is to be printed at least six characters wide, with 1 digit after the decimal point. The output looks like this:

```
  0   -17.8
 20    -6.7
 40     4.4
...
```

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 floating point.

`%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.

Exercise 1-3. Modify the temperature conversion program to print a heading above the table.

Exercise 1-4. Write a program to print the corresponding Celsius to Fahrenheit table.

1.3 The for statement

There are plenty of different ways to write a program for a particular task. Let's try a variation on the temperature converter.

```
#include <stdio.h>

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

This produces the same answers, but it certainly looks different. One major change is the elimination of most of the variables; only `fahr` remains, and we have made it an `int`. The lower and upper limits and the step size appear only as constants in the `for` statement, itself a new construction, and the expression that computes the Celsius temperature now appears as the third argument of `printf` instead of a separate assignment statement.

This last change is an instance of a general rule - in any context where it is permissible to use the value of some type, you can use a more complicated expression of that type. Since the third argument of `printf` must be a floating-point value to match the `%6.1f`, any floating-point expression can occur here.

The `for` statement is a loop, a generalization of the `while`. If you compare it to the earlier `while`, its operation should be clear. Within the parentheses, there are three parts, separated by semicolons. The first part, the initialization

```
fahr = 0
```

is done once, before the loop proper is entered. The second part is the test or condition that controls the loop:

```
fahr <= 300
```

This condition is evaluated; if it is true, the body of the loop (here a single `ptintf`) is executed. Then the increment step

```
fahr = fahr + 20
```

is executed, and the condition re-evaluated. The loop terminates if the condition has become false. As with the `while`, the body of the loop can be a single statement or a group of statements enclosed in braces. The initialization, condition and increment can be any expressions.

The choice between `while` and `for` is arbitrary, based on which seems clearer. The `for` is usually appropriate for loops in which the initialization and increment are single statements and logically related, since it is more compact than `while` and it keeps the loop control statements together in one place.

Exercise 1-5. Modify the temperature conversion program to print the table in reverse order, that is, from 300 degrees to 0.

1.4 Symbolic Constants

A final observation before we leave temperature conversion forever. 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 give them meaningful names. A `#define` line defines a *symbolic name or symbolic constant* to be a particular string of characters:

`#define` *name replacement list*

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.

```
#include <stdio.h>

#define LOWER  0     /* lower limit of table */
#define UPPER  300   /* upper limit */
#define STEP   20    /* step size */

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

The quantities `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 ber readily distinguished from lower case variable names. Notice that there is no semicolon at the end of a `#define` line.