INDEX

Chapter 1. Introduction to C

1. What is C?

A high-level programming language developed by Dennis Ritchie at Bell Labs in the mid 1972s. Although originally designed as a systems programming language, C has proved to be a powerful and flexible language that can be used for a variety of applications, from business programs to engineering. C is a particularly popular language for personal computer programmers because it is relatively small -- it requires less memory than other languages.

The first major program written in C was the UNIX operating system, and for many years C was considered to be inextricably linked with UNIX. Now, however, C is an important language independent of UNIX.

Although it is a high-level language, C is much closer to assembly language than are most other high-level languages. This closeness to the underlying machine language allows C programmers to write very efficient code. The low-level nature of C, however, can make the language difficult to use for some types of applications.

2. Getting Started With C

Communicating with a computer involves speaking the language the computer understands, which immediately rules out English as the language of communication with computer. However, there is a close analogy between learning English language and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them constants, variables and keywords are constructed, and finally how are these combined to form an instruction. A group of instructions would be combined later on to form a program.

End of Chapter

2. Data Types, Variables, Constants

Constants, Variables and Keywords

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords. Let us see what are ‘constants’ and ‘variables’ in C. A constant is an entity that doesn’t change whereas a variable is an entity that may change. In any program we typically do lots of calculations. The results of these calculations are stored in computers memory. Like human memory the computer memory also consists of millions of cells. The calculated values are stored in these memory cells. To make the retrieval and usage of these values easy these memory cells (also called memory locations) are given names. Since the value stored in each location may change the names given to these locations are called variable names.

Consider the following example.

Here 3 is stored in a memory location and a name **x** is given to it. Then we are assigning a new value 5 to the same memory location **x**. This would overwrite the earlier value 3, since a memory location can hold only one value at a time. This is shown in Figure 1.3.

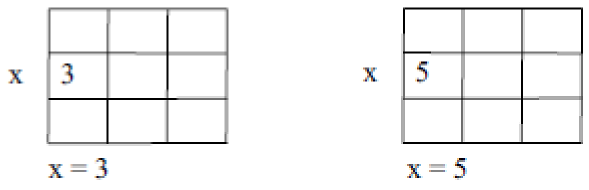


Figure 1.3

Since the location whose name is **x** can hold different values at different times **x** is known as a variable. As against this, 3 or 5 do not change, hence are known as constants.

Types of C Variables

An entity that may vary during program execution is called a variable. Variable names are names given to locations in memory. These locations can contain integer, real or character constants. In any language, the types of variables that it can support depend on the types of constants that it can handle. This is because a particular type of variable can hold only the same type of constant.

For example, an integer variable can hold only an integer constant, a real variable can hold only a real constant and a character variable can hold only a character constant. The rules for constructing different types of constants are different.

Data Type Name	Type
Char	Character
Unsigned char	Unsigned char
Sign char	Sign char (same as char)
Int	Integer
Unsigned int	Unsigned integer
Signed int	Signed integer (same as int)
Short int	Short integer
Unsigned short int	Unsigned short integer
Signed short int	Signed short integer
Long	Long integer
Long int	Long integer (same as long)
Unsigned long int	Unsigned long int
Signed long int	Signed long int (same as long int)
Float	Floating-point
Double	Double floating-point
Long double	Long double floating-point

However, for constructing variable names of all types the same set of rules apply. These rules are given below.

Rules for Constructing Variable Names

- A variable name is any combination of 1 to 31 alphabets, digits or underscores. Some compilers allow variable names whose length could be up to 247 characters. Still, it would be safer to stick to the rule of 31 characters. Do not create unnecessarily long variable names as it adds to your typing effort.

- The first character in the variable name must be an alphabet or underscore.

- No commas or blanks are allowed within a variable name.

- No special symbol other than an underscore (as in **gross_sal**) can be used in a variable name.

```
Ex.: si_int
m_hra
pop_e_89
```

These rules remain same for all the types of primary and secondary variables. Naturally, the question follows... how is C able to differentiate between these variables? This is a rather simple matter. C compiler is able to distinguish between the variable names by making it compulsory for you to declare the type of any variable name that you wish to use in a program. This type declaration is done at the beginning of the program. Following are the examples of type declaration statements:

```
Ex.: int si, m_hra ;
float bassal ;
char code ;
```

Since, the maximum allowable length of a variable name is 31 characters, an enormous number of variable names can be constructed using the above-mentioned rules. It is a good practice to exploit this enormous choice in naming variables by using meaningful variable names.

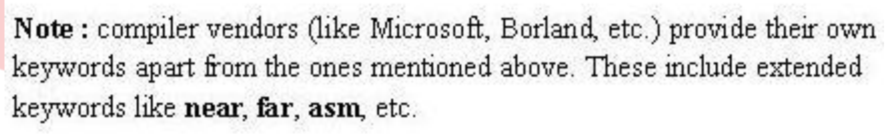
Thus, if we want to calculate simple interest, it is always advisable to construct meaningful variable names like **prin**, **roi**, **noy** to represent Principle, Rate of interest and Number of years rather than using the variables **a**, **b**, **c**.

C Keywords

Keywords are the words whose meaning has already been explained to the C compiler (or in a broad sense to the computer). The keywords **cannot** be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer. Some C compilers allow you to construct variable names that exactly resemble the keywords. However, it would be safer not to mix up the variable names and the keywords. The keywords are also called 'Reserved words'. There are only 32 keywords available in C. Figure 1.5 gives a list of these keywords for your ready reference. A detailed discussion of each of these keywords would be taken up in later chapters wherever their use is relevant.

Auto	double	int	Struct
Break	else	long	Switch
Case	enum	register	Typedef
Char	extern	return	Union
Const	float	short	Unsigned
Continue	for	signed	Void
Default	goto	sizeof	Volatile
Do	if	Static	while

Figure 1.5



Note : compiler vendors (like Microsoft, Borland, etc.) provide their own keywords apart from the ones mentioned above. These include extended keywords like **near**, **far**, **asm**, etc.

Types of C Constants

C constants can be divided into two major categories:

- Primary Constants
- Secondary Constants

These constants are further categorized as shown in Figure 1.4.

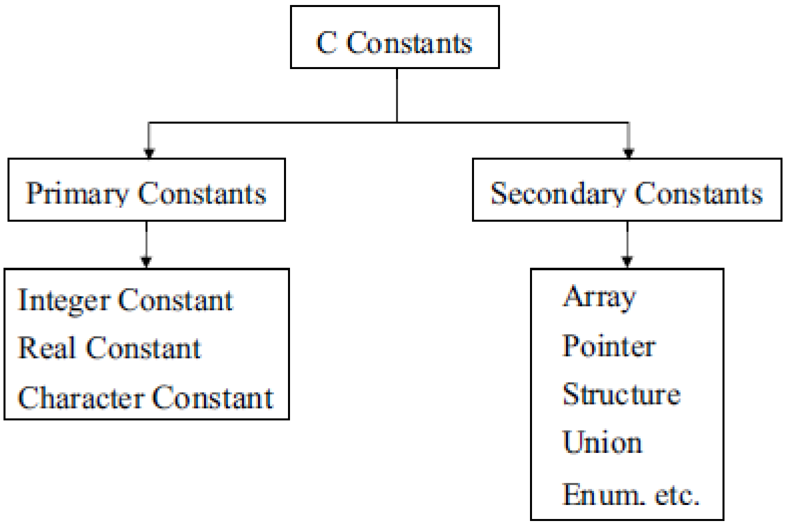


Figure 1.4

At this stage we would restrict our discussion to only Primary Constants, namely, Integer, Real and Character constants. Let us see the details of each of these constants. For constructing these
different types of constants certain rules have been laid down. These rules are as under:

Rules for Constructing Integer Constants

An integer constant must have (a) at least one digit.

- It must not have a decimal point.
- It can be either positive or negative.
- If no sign precedes an integer constant it is assumed to be positive.
- No commas or blanks are allowed within an integer constant.
- The allowable range for integer constants is -32768 to 32767.

The range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the range is –32768 to 32767. For a 32-bit compiler the range would be even greater.

Ex.: 426
+782
-8000
-7605

Rules for Constructing Real Constants

Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.

Following rules must be observed while constructing real constants expressed in fractional form:

- A real constant must have at least one digit.
- It must have a decimal point.
- It could be either positive or negative.
- Default sign is positive.
- No commas or blanks are allowed within a real constant.

Ex.: +325.34
426.0
-32.76
-48.5792

The exponential form of representation of real constants is usually used if the value of the constant is either too small or too large. It however doesn't restrict us in any way from using exponential form of representation for other real constants.

In exponential form of representation, the real constant is represented in two parts. The part appearing before 'e' is called mantissa, whereas the part following 'e' is called exponent. Following rules must be observed while constructing real constants expressed in exponential form:

- The mantissa part and the exponential part should be separated by a letter e.
- The mantissa part may have a positive or negative sign.
- Default sign of mantissa part is positive.
- The exponent must have at least one digit, which must be a positive or negative integer.
- Default sign is positive.
- Range of real constants expressed in exponential form is -3.4e38 to 3.4e38.

Ex.: +3.2e-5
4.1e8
-0.2e+3
-3.2e-5

Rules for Constructing Character Constants

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas. Both the inverted commas should point to the left. For example, 'A' is a valid character constant whereas 'A' is not.

- The maximum length of a character constant can be 1 character.

```
Ex.: 'A'
'I'
'5'
'='
```

Rules for Constructing Programs

Armed with the knowledge about the types of variables, constants & keywords the next logical step is to combine them to form instructions. However, instead of this, we would write our first C
program now. Once we have done that we would see in detail the instructions that it made use of.
Before we begin with our first C program do remember the following rules that are applicable to all C programs:

- Each instruction in a C program is written as a separate statement. Therefore a complete C program would comprise of a series of statements.

- The statements in a program must appear in the same order in which we wish them to be executed; unless of course the logic of the problem demands a deliberate 'jump' or transfer of
- Control to a statement, which is out of sequence.

- Blank spaces may be inserted between two words to improve the readability of the statement. However, no blank spaces are allowed within a variable, constant or keyword.

- All statements are entered in small case letters.

- C has no specific rules for the position at which a statement is to be written. That's why it is often called a free-form language.

- Every C statement must end with a ; Thus ; acts as a statement terminator.

Getting Started

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.

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 to produce identical output.

```
#include <stdio.h>
main()
{
printf("hello, ");
printf("world");
printf("\n");
}
```

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.

End of Chapter