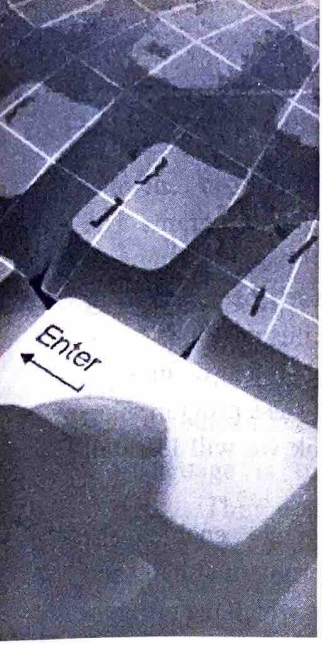


CHAPTER 8

Introduction to C



LEARNING OBJECTIVE

This chapter aims at acquainting students with writing programs in C. To start with we will read about the background of the C language, its key features, and uses in the real world. Then we will gradually move forward to understand the structure of a C program and the types of files that will be used to execute the program. Then moving a step further, we will learn about data types, identifiers, constants, variables, and operators available in the C language.

8.1 INTRODUCTION

The programming language C was developed in the early 1970s by Dennis Ritchie at Bell Laboratories to be used by the UNIX operating system. It was named 'C' because many of its features were derived from an earlier language called 'B'. Although C was designed for implementing system software, it was later widely used for developing portable application software.

C is one of the most popular programming languages. It is being used on several different software platforms. In a nutshell, there are few computer architectures for which a C compiler does not exist.

It is a good idea to learn C because it has been around for a long time which means there is a lot of information available on it. Quite a few other programming languages such as C++ and Java are also based on C which means that one will be able to learn them more easily in the future.

8.1.1 Background

Like many other modern languages, C is derived from ALGOL (the first language to use a block structure). Although ALGOL was not accepted widely in the United States but it was widely used in Europe. ALGOL's introduction in the 1960s led the way for the development of structured programming concepts.

Before C, several other programming languages were developed. For example, in 1967 Martin Richards developed a language called BCPL (Basic Combined Programming Language). BCPL was basically a type-less (had no concept of data types) language which facilitated the user with direct access of memory. This made it useful for system programmers. Then in 1970, Ken Thompson developed a language called B. B was used to develop the first version of UNIX. C was developed by Dennis Ritchie in 1972 that took concepts from ALGOL, BCPL, and B. In addition to the concepts of these languages, C also supports the concept of data types. Since UNIX operating system was also developed at Bell Laboratories along with

the C language, C and UNIX are strongly associated with each other.

For many years, C was mainly used in academic institutions, but with the release of different C compilers for commercial use and popularity of UNIX, C was widely accepted by computer professionals.

C (also known as Traditional C) was documented and popularized in the book *The C Programming Language* by Brian W. Kernighan and Dennis Ritchie in 1978. This book was so popular that the language came to be known as 'K & R C'. The tremendous growth of the C language resulted in the development of different versions of the language that were similar but incompatible with each other. Therefore, in the year 1983, the American National Standards Institute (ANSI) started working on defining the standard for C. This standard was approved in December 1989 and came to be known as ANSI C. In 1990, the International Standards Organization (ISO) adopted the ANSI standard. This version of C came to be known as C89. In 1995, some minor changes were made to C89, the new modified version was known as C95. Figure 8.1 shows the taxonomy of the C language. During the 1990s C++ and Java became popular programming languages among the users so the standardization committee of C felt that a few features of C++/Java, if added to C, would enhance its usefulness. So, in 1999 when some significant changes were made to C95, the modified version came to be known as C99. Some of the changes made in the C99 version are as follows:

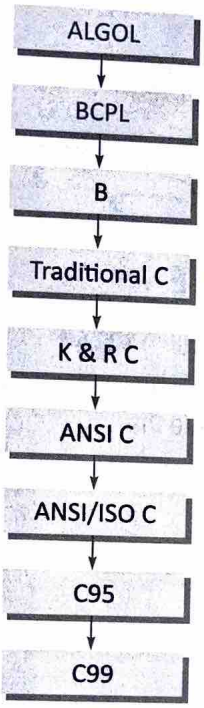


Figure 8.1 Taxonomy of C language

- Extension to the character types, so that they can support even non-English characters
- Boolean data type
- Extension to the integer data type
- Including type definitions in the for statement
- Inclusion of imaginary and complex types
- Addition of //, better known as C++ style line comment

8.1.2 Characteristics of C

C is a robust language whose rich set of built-in functions and operators can be used to write complex programs. The C compiler combines the features of assembly language and high-level language which makes it best suited for writing system software as well as business packages. Some basic characteristics of C language that defines the language and have lead to its popularity as a programming language are listed below. In this book we will learn all these aspects.

- A high level programming language enables the programmer to concentrate on the problem at hand and not worry about the machine code on which the program would be run.
- Small size—C has only 32 keywords. This makes it relatively easy to learn as compared to other languages.
- It makes extensive use of function calls.
- C is well suited for structured programming. In this programming approach, C enables the users to think of the problem in terms of functions/modules where the collection of all the modules makes up a complete program. This feature facilitates easy program debugging, testing, and maintenance.
- Unlike PASCAL it supports loose typing (as a character can be treated as an integer and vice versa).
- Structured language as the code can be organized as a collection of one or more functions.
- It is a stable language. ANSI C was created in 1983 and since then it has not been revised.
- It is a quick language as a well written C program is likely to be as quick as or quicker than a program written in any other language. Since C programs make use of operators and data types, they are fast and efficient. For example, a program written to increment a value from 0–15000 using BASIC would take 50 seconds whereas a C program would do the same in just 1 second.
- Facilitates low level (bitwise) programming
- Supports pointers to refer computer memory, array, structures, and functions.
- C is a core language as many other programming languages (such as C++, Java, or Perl) are based on C. If one knows C, learning other computer languages becomes much easier.
- C is a portable language, i.e., a C program written for one computer can be run on another computer with little or no modification.
- C is an extensible language as it enables the user to add his/her own functions to the C library.

- C is often treated as the second best language for any given programming task. While the best language depends on the particular task to be performed, the second best language, will always be C.

8.1.3 Uses of C

C is a very simple language that is widely used by software professionals around the globe. The uses of C language can be summarized as follows.

- Clanguage is primarily used for system programming. The portability, efficiency, the ability to access specific hardware addresses and low runtime demand on system resources makes it a good choice for implementing operating systems and embedded system applications.
- C has been so widely accepted by professionals that compilers, libraries, and interpreters of other programming languages are often implemented in C.
- For portability and convenience reasons, C is sometimes used as an intermediate language for implementations of other languages. Examples of compilers who use C this way are BitC, Gambit, the Glasgow Haskell Compiler, Squeak, and Vala.
  Basically, C was designed as a programming language and was not meant to be used as a compiler target language. Therefore, although C can be used as an intermediate language it is not an ideal option. This led to the development of C-based intermediate languages such as C--.
- C is widely used to implement end-user applications.

8.2 STRUCTURE OF A C PROGRAM

A C program is composed of preprocessor commands, a global declaration section, and one or more functions (Figure 8.2).

The preprocessor directives contain special instructions that indicate how to prepare the program for compilation. One of the most important and commonly used preprocessor commands is `include` which tells the compiler that to execute the program, some information is needed from the specified header file.

In this section we will omit the global declaration part and will revisit it in Chapter 10.

A C program contains one or more functions, where a function is defined as a group of C statements that are executed together. The statements in a C program are written in a logical sequence to perform a specific task. The `main()` function is the most important function and

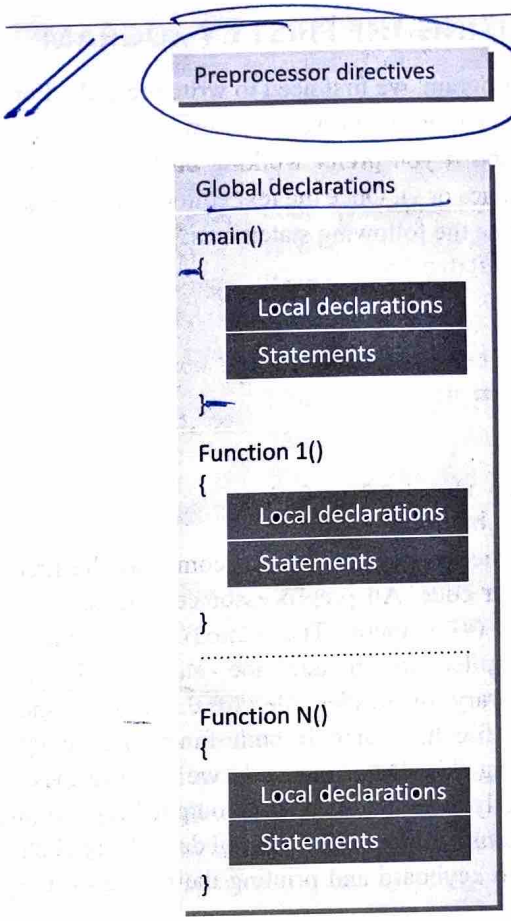


Figure 8.2 Structure of a C program

is a part of every C program. The execution of a C program begins at this function.

All functions (including `main()`) are divided into two parts—the declaration section and the statement section. The declaration section precedes the statements and is used to describe the data that will be used in the function. Note that data declared within a function are known as local declaration as that data will be visible only within that function. Stated in other terms, the life-time of the data will be only till the function do ends. The statement section in a function contains the code that manipulates the data to perform a specified task.

From this structure we can conclude that a C program can have any number of functions depending on the tasks that have to be performed and each function can have any number of statements arranged according to a specific meaningful sequence.

Note Programmers can choose any name for the functions. It is not mandatory to name them `Function1`, `Function2`, etc., but every program must contain one function that has its name as `main()`.

8.3 WRITING THE FIRST C PROGRAM

To write a C program, we first need to write the code. For this, open a text editor. If you are a Windows user you may use Notepad and if you prefer working on UNIX/Linux you can use emacs or vi. Once the text editor is opened on your screen, type the following statements:

```
#include <stdio.h>
   int main()
   {
      printf("\n Welcome to the world of C");
      return 0;
   }
```

Output

```
Welcome to the world of C
```

#include <stdio.h>

This is a preprocessor command that comes as the first statement in our code. All preprocessor commands start with the hash (#) symbol. The `#include` statement tells the compiler to include the standard input/output(I/O) library or header file `(stdio.h)` in the program. This file has some in-built functions. So by simply including this file in our code we can use these functions directly. The standard input/output header file contains functions for input and output of data like reading values from the keyboard and printing the results on the screen.

int main()

`int` is the return value of the main function. After all the statements in the program have been written, the last statement of the program will return an integer value to the operating system. The concepts will be clear to us when we read Chapter 10. So even if you do not understand certain things, do not worry.

Programming Tip: If you do not place a parenthesis after 'main', a compiler error will be generated.

{ } The two curly brackets are used to group all the related statements of the `main` function. All the statements between the braces form the function body. The function body contains a set of instructions to perform the given task.

printf("\n Welcome to the world of C");

The `printf` function is defined in the `stdio.h` file and is used to print text on the screen. The message that has to be displayed on the screen is enclosed within double quotes and put inside brackets. The message is quoted because in C a text (also known as a string or a sequence of characters) is always placed between inverted commas. The `'\n'` is an escape sequence and represents a `newline` character. It is used to print the message on a new line on the screen. Similar to the `newline` character, the other escape sequences supported by C language are shown in Table 8.1.

Programming Tip: Placing a semi-colon after the parenthesis of main will generate a compiler error.

Table 8.1 Escape sequences

Escape sequence	Purpose	Escape sequence	Purpose
\a	Audible signal	\?	Question mark
\b	Backspace	\\	Back slash
\t	Tab	\'	Single quote
\n	Newline	\"	Double quote
\v	Vertical tab	\0	Octal constant
\f	New page\ Clear screen	\x	Hexadecimal constant
\r	Carriage return		

Note Escape sequences are actually non-printing control characters that begin with a backslash (\).

return 0;

This is a return command that is used to return the value 0 to the operating system to give an indication that there were no errors during the execution of the program.

Note Every statement in the main function ends with a semicolon (;).

Now that you have written all the statements using the text editor, save the text file as `first.c`. If you are a Windows user then open the command prompt by clicking Start->Run and typing 'command' and clicking OK. Using the command prompt, change to the directory in which you had saved your file and then type:

```
C:\>tc first.c
```

In case you are working on UNIX/Linux operating system, then exit the text editor and type

```
$cc first.c -ofirst
```

The `-o` is for the output file name. If you leave out the `-o` then the file name `a.out` is used.

This command is used to compile your C program. If there are any mistakes in the program then the compiler will tell you the mistake you have made and on which line you made it. In case of errors you need to re-open your .C file and correct those mistakes. However, if everything is right then no error(s) will be reported and the compiler

will create an exe file for your program. This .exe file can be directly run by typing `'hello.exe'` for Windows and `./hello'` for UNIX/Linux operating system.

When you run the exe file, the output of the program will be displayed on screen. That is,

```
Welcome to the world of C
```

Note: The printf and return statements have been indented or moved away from the left side. This is done to make the code more readable.

8.4 FILES USED IN A C PROGRAM

Every C program has four kinds of files associated with it (Figure 8.3). These include:

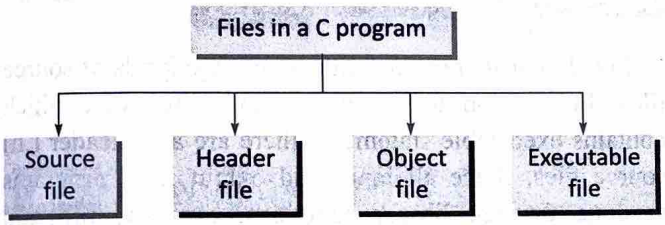


Figure 8.3 Files in a C program

8.4.1 Source Code File

The source code file contains the source code of the program. The file extension of any C source code file is '.c'. This file contains C source code that defines the main function and maybe other functions. The `main()` is the starting point of execution when you successfully compile and run the program. A C program in general may include even other source code files (with the file extension .c).

8.4.2 Header files

When working with large projects, it is often desirable to separate out certain sub routines from the `main()` of the program. There also may be a case that the same subroutine has to be used in different programs. In the latter case, one option is to copy the code of the desired subroutine from one program to another. However, copying the code is often tedious as well as error-prone and makes maintainability more difficult.

So, another option is to make subroutines and store them in a different file known as header file. The advantage of header files can be realized in the following cases:

- The programmer wants to use the same subroutines in different programs. For this, one simply has to compile the source code of the subroutines once, and then link to the resulting object file in any other program in which the functionalities of these sub-routines are required.
- The programmer wants to change or add subroutines, and have those changes reflected in all the other programs. In this case, one just needs to change the source file for the subroutines, recompile its source code, and then recompile and re-link programs that use them. This way, enormous time can be saved as compared to editing the subroutines in every individual program that uses them.

Programming Tip: Missing the inclusion of appropriate header file in the C program will generate an error. Such a program may compile but the linker will give an error message as it will not be able to find the functions used in the program.

Thus, we see that using a header file produces the same results as copying the header file into each source file that needs it. When a header file is included, the related declarations appear in only one place. If in future we need to modify the subroutines, we just need to make the changes in one place, and programs that include the header file will automatically use the new version when recompiled later. There is no need to find and change all the copies of the subroutine that has to be changed.

Conventionally, header files names end with a 'dot h' (.h) extension and its name can use only letters, digits, dashes, and underscores. Although some standard header files are automatically available to C programmers but in addition to those header files, the programmer may have his/her own user defined header files.

Standard header files In all our programs that we had been writing till now, we were using many functions that were not written by us. For example, `strcmp()` function which compares two strings. We just pass string arguments and retrieve the result. We do not know the details of how these functions work. Such functions that are provided by all C compilers are included in standard header files. Examples of these standard header files include the following:

- `string.h` : for string handling functions
- `stdlib.h` : for some miscellaneous functions
- `stdio.h` : for standardized input and output functions
- `math.h` : for mathematical functions
- `alloc.h` : for dynamic memory allocation
- `conio.h` : for clearing the screen

All the header files are referenced at the start of the source code file that uses one or more functions from that file.

8.4.3 Object Files

Object files are generated by the compiler as a result of processing the source code file. They contain compact

binary code of the function definitions. Linker uses this object file to produce an executable file (.exe file) by combining the object files together. Object files have a '.o' extension, although some operating systems including Windows and MS-DOS have a '.obj' extension for the object file.

8.4.4 Binary Executable File

The binary executable file is generated by the linker. The linker links the various object files to produce a binary file that can be directly executed. On Windows operating system, the executable files have '.exe' extension.

8.5 COMPILING AND EXECUTING C PROGRAMS

C is a compiled language. So once the C program is written, you must run it through a C compiler that can create an executable file to be run by the computer. While the C program is human-readable, the executable file on the other hand, is a machine-readable file available in an executable form.

The mechanical part of running a C program begins with one or more program source files, and ends with an executable file, which can be run on a computer.

The programming process starts with creating a source file that consists of the statements of the program written in C language. This source file usually contains ASCII characters and can be produced with a text editor, such as Windows notepad, or in an Integrated Design Environment.

The source file is then processed by a special program called a compiler.



Every programming language has its own compiler.

The compiler translates the source code into an object code. The object code contains the machine instructions for the CPU and calls to the operating system application programming interface (API).

However, even the object file is not an executable file. Therefore, in the next step, the object file is processed with another special program called a linker. While there is a different compiler for every individual language, the same linker is used for object files regardless of the original language in which the new program was written. The output of the linker is an executable or runnable file. The process is shown in Figure 8.4.

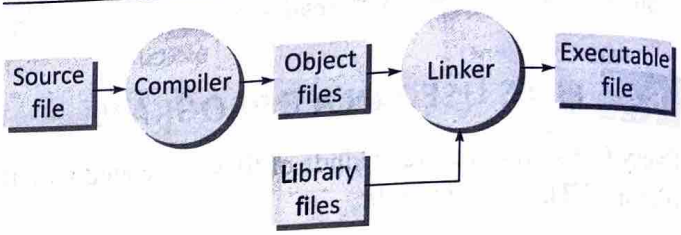


Figure 8.4 Overview of compilation and execution process

In C language programs, there are two kinds of source files. In addition to the main (.c) source file, which contains executable statements there are also header (.h) source files. Since all input and output in C programs is done through library functions, every C program therefore uses standard header files. These header files should be written as part of the source code for modular C programs.

The compilation process shown in Figure 8.5 is done in two steps. In the first step, the preprocessor program reads the source file as text, and produces another text file as output. Source code lines which begin with the # symbol are actually not written in C but in the preprocessor language. The output of the preprocessor is a text file which does not contain any preprocessor statements. This file is ready to be processed by the compiler. The linker combines the object file with library routines (supplied with the compiler) to produce the final executable file.

In modular programming the source code is divided into two or more source files. All these source files are compiled separately thereby producing multiple object

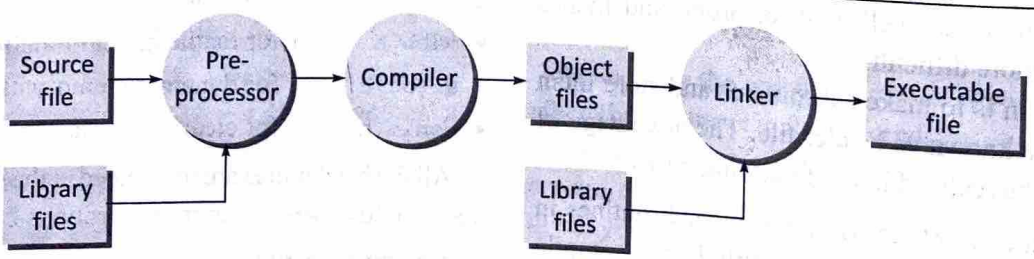


Figure 8.5 Elaborate compilation before execution

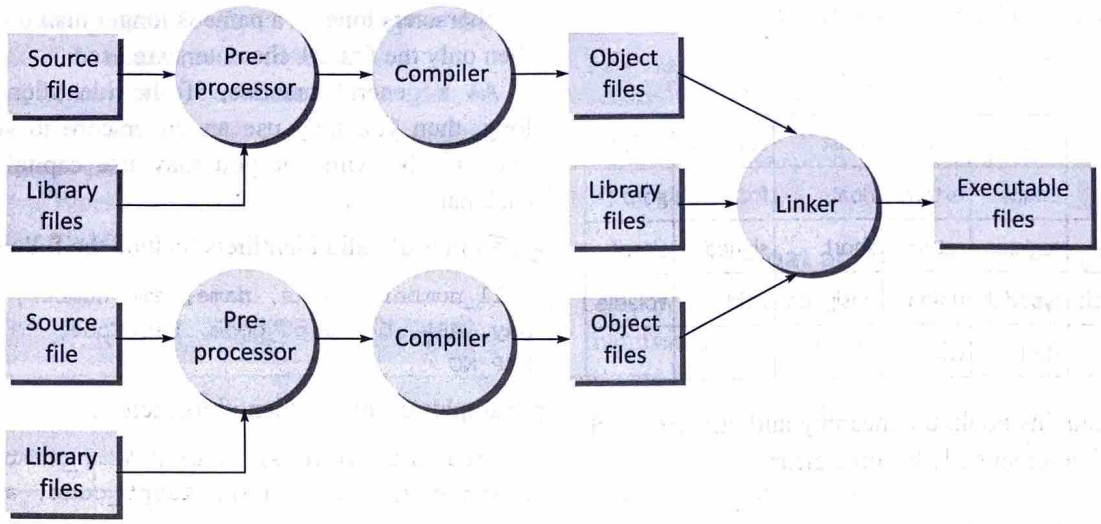


Figure 8.6 Modular programming—the complete compilation and execution process

files. These object files are combined by the linker to produce an executable file (Figure 8.6).

8.6 USING COMMENTS

Many a time the meaning or the purpose of the program code is not clear to the reader. Therefore, it is a good programming practice to place some comments in the code to help the reader understand the code clearly. Comments are just a way of explaining what a program does. It is merely an internal program documentation. The compiler ignores the comments when forming the object file. This means that the comments are non-executable statements. C supports two types of commenting.

- `//` is used to comment a single statement. This is known as a *line comment*. A line comment can be placed anywhere on the line and it does not require to be specifically ended as the end of the line automatically *ends the line.*
- `/*` is used to comment multiple statements. A `/*` is ended with `*/` and all statements that lie within these characters are commented. This type of comment is known as *block comment.*

Note that commented statements are not executed by the compiler. Rather, they are ignored by the compiler as they are simply added in the program to make the code understandable by the programmer as well as other people who read it. It is a good habit to always put a comment at the top of a program that tells you what the program does. This will help in defining the usage of the program the moment you open it.

Programming Tip:
Not including the */ after the termination of the block comment is a compiler error.

Commented statements can be used anywhere in the program. You can also use comments in between your code to explain a piece of code that is a bit complicated. The following code shows the way in which we can make use of comments in our first program.

```
/* Author: Reema Thareja
   Description: To print 'Welcome to the world
        of C' on the screen */
#include <stdio.h>
int main()
{
     printf("\n Welcome to the world of C ");
// prints message
   return 0; // returns a value 0 to the
             // operating system
}
```

Output

```
Welcome to the world of C
```

Since comments are not executed by the compiler, they do not affect the execution speed and the size of the compiled program. Therefore, use comments liberally in your programs so that other users can understand the operations of the program and will serve as an aid to debugging and testing.

8.7 KEYWORDS

Like every computer language, C has a set of reserved words often known as keywords that cannot be used as an identifier. All keywords are basically a sequence of characters that have a fixed meaning. By convention all

keywords must be written in lowercase (small) letters. Table 8.2 contains a list of keywords in C.

Table 8.2 Keywords in C language

auto	break	case	char	const	continue	default
double	else	enum	extern	float	for	goto
int	long	register	return	short	signed	sizeof
struct	switch	typedef	union	unsigned	void	volatile
do	if	static	while			

When you read this book, the meaning and utility of each keyword will automatically become clear.

8.8 IDENTIFIERS

Identifiers, as the name suggests helps us to identify data and other objects in the program. Identifiers are basically the names given to program elements such as variables, arrays, and functions. They may consist of an alphabet, digit, or an underscore.

8.8.1 Rules for Forming Identifier Names

Some rules have to be followed while forming identifier names. They are as follows:

- It cannot include any special characters or punctuation marks (e.g., #, $, ^, ?, ., etc.) except the underscore '_'.
- There cannot be two successive underscores.
- Keywords cannot be used as identifiers.
- The case of alphabetic characters that form the identifier name is significant. For example, `'FIRST'` is different from `'first'` and `'First'`.
- The identifier name must begin with an alphabet or an underscore. However, use of underscore as the first character must be avoided because several compiler-defined identifiers in the standard C library have underscore as their first character. Hence, inadvertently duplicated names may cause definition conflicts.

Programming Tip:
C is a case-sensitive language. If you type the printf function as Printf, then an error will be generated.

- Identifiers can be of any reasonable length. They should not contain more than 31 characters. It can actually be longer than 31, but the compiler looks at only the first 31 characters of the name.

Although it is a good practice to use meaningful identifier names, it is not compulsory. Good identifiers are descriptive but short. To cut short the identifier, you may use abbreviations. C allows identifiers (names) to be up to 63 characters long. If a name is longer than 63 characters, then only the first 31 characters are used.

As a general practice, if the identifier is a little long, then you may use an underscore to separate the parts of the name or you may use capital letters for each part.

Examples of valid identifiers include the following:

```
roll_number, marks, name, emp_number, basic_
pay, HRA, DA, dept_code, DeptCode, RollNo,
EMP_NO
```

Examples of invalid identifiers include:

```
23_student, %marks, @name, #emp_number,
basic.pay, -HRA, (DA), &dept_code, auto
```

Note C is a case-sensitive language. Therefore rno, Rno, RNo, RNO are different considered as identifiers.

8.9 BASIC DATA TYPES IN C

C language provides very few basic data types. Table 8.3 lists the data type, their size, range, and usage for a C programmer on a 16-bit computer. Table 8.3 shows the basic data types. In addition to this, we also have variants of `int` and `float` data types.

Table 8.3 Basic data types in C

Data type	Keyword used	Size in bytes	Range	Use
Character	char	1	–128 to 127	To store characters
Integer	int	2	–32768 to 32767	To store integer numbers
Floating point	float	4	3.4E–38 to 3.4E+38	To store floating point numbers
Double	double	8	1.7E–308 to 1.7E+308	To store big floating point numbers
Valueless	void	0	Valueless	—

The `char` data type is one byte long and is used to store single characters. Note that C does not provide any data type for storing text. This is because text is made up of individual characters.

You will be surprised to see that the range of `char` is given as –128 to 127. `char` is supposed to store characters not numbers, so why this range? The answer is that, in memory

characters are stored using their ASCII codes. For example, the character 'A' has the ASCII code 65. In memory we will not store 'A' but 65 (in binary number format).
Table 8.4 shows the variants of basic data types in detail.

In Table 8.4, we have `unsigned char` and `signed char`. Do we have negative characters?—no, then why do we have such data types? The answer is that we use these data types to ensure portability of programs that store non-character data as `char`.

Table 8.4 Detailed list of data types

Data type	Size in bytes	Range
char	1	–128 to 127
unsigned char	1	0 to 255
signed char	1	–128 to 127
int	2	–32768 to 32767
unsigned int	2	0 to 65535
signed short int	2	–32768 to 32767
signed int	2	–32768 to 32767
short int	2	–32768 to 32767
unsigned short int	2	0 to 65535
long int	4	–2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	–2147483648 to 2147483647
float	4	3.4E–38 to 3.4E+38
double	8	1.7E–308 to 1.7E+308
long double	10	3.4E–4932 to 1.1E+4932

While the smaller data types take less memory, the larger types incur a performance penalty. Although the data type we use for our variables does not have a big impact on the speed or memory usage of the application, we should always try to use `int` unless there is a special need to use any other data type.

Last but not least the `void` type holds no value. It is primarily used in three cases:

- To specify the return type of a function (when the function returns no value)
- To specify the parameters of the function (when the function accepts no arguments from the caller)
- To create generic pointers. We will read about generic pointers in Chapter 13.

We will discuss the `void` data type in detail in the coming chapters.

Note Unsigned int/char keeps the sign bit free and makes the entire word available for storage of the non-negative numbers.
Sign bit is the leftmost bit of a memory word which is used to determine the sign of the content stored in that word. When it is 0, the value is positive and when it is 1, the value is negative.

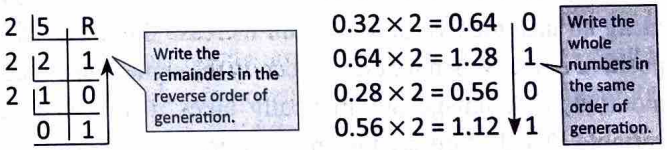
8.9.1 Storing Float and Double Data Types

In computer memory, float and double values are stored in mantissa and exponent forms where the exponent represents the power of 2 (not 10). The number of bytes used to represent a floating point number generally depends on the precision of the value. While float is used to declare single-precision values, double is used to represent double-precision values.

Floating-point numbers use the IEEE (Institute of Electrical and Electronics Engineers) format to represent mantissa and exponents. According to this format, a floating point value in its binary form is known as a *normalized form*. In the normalized form, the exponent is adjusted in such a way that the binary point in the mantissa always lies to the right of the most significant non-zero digit.

Example 8.1 Convert the floating point number 5.32 into an IEEE normalized form.

Solution



Thus, the binary equivalent of 5.32 = 101.0101.
The normalized form of this binary number is obtained by adjusting the exponent until the decimal point is to the right of the most significant 1.
Therefore, the normalized binary equivalent = 1.010101×2^2.

Moreover, the IEEE format for storing floating point numbers uses a sign bit, mantissa, and the exponent (Figure 8.7). The sign bit denotes the sign of the value. If the value is positive, the sign bit contains 0 and in case the value is negative it stores 1.

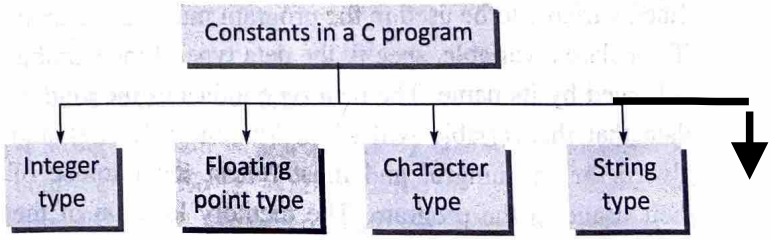


Figure 8.7 IEEE format for storing floating point numbers

Generally, exponent is an integer value stored in unsigned binary format after adding a positive bias. In other words, because exponents are stored in an unsigned form, the exponent is biased by half its possible value. For float, the bias is 127; for double, it is 1023. You can compute the actual exponent value by subtracting the bias value from the exponent value. Finally, the normalized binary equivalent is stored in such a way that lower byte is stored at higher memory address. For example, ABCD is actually stored as DCBA.

8.10 VARIABLES

A *variable* is defined as a meaningful name given to the data storage location in computer memory. When using a variable, we actually refer to an address of the memory where the data is stored. C language supports two basic kinds of variables—numeric and character.

8.10.1 Numeric Variables

Numeric variables can be used to store either integer values or floating point values. While an integer value is a whole number without a fraction part, a floating point value can have a decimal point.

Numeric variables may also be associated with modifiers such as short, long, signed, and unsigned. The difference between signed and unsigned numeric variables is that signed variables can be either negative or positive, but unsigned variables can only be positive. Therefore, by using an unsigned variable we can increase the maximum positive range. When we do not specify the signed/unsigned modifier, C language automatically takes it as a signed variable. To declare an unsigned variable, the unsigned modifier must be explicitly added during the declaration of the variable.

8.10.2 Character Variables

Character variables can include any letter from the alphabet or from the ASCII chart and numbers 0–9 that are given within single quotes. In C, a number that is given in single quotes is not the same as a number without them. This is because 2 is treated as an integral value but '2' is considered a character not an integer.

8.10.3 Declaring Variables

Each variable to be used in the program must be declared. To declare a variable, specify the data type of the variable followed by its name. The data type indicates the kind of data that the variable will store. Variable names should always be meaningful and must reflect the purpose of their usage in the program. The memory location of the variable is of importance to the compiler only and not to the programmer. Programmers must only be concerned with accessing data through their symbolic names. In C, variable declaration always ends with a semicolon, for example:

```
int emp_num;  float salary;
char grade;   double balance_amount;
unsigned short int acc_no;
```

In C, variables can be declared at any place in the program but two things must be kept in mind. First, variables should be declared before using them. Second, variables should be declared closest to their first point of use to make the source code easier to maintain.

C allows multiple variables of the same type to be declared in one statement. So the following statement is absolutely legal in C.

```
float temp_in_celsius, temp_in_farenheit;
```

In C, variables are declared at three basic places as follows:

- When a variable is declared inside a function it is known as a *local variable*.
- When a variable is declared in the definition of function parameters it is known as *formal parameter* (we will study this in Chapter 10).
- When the variable is declared outside all functions, it is known as a *global variable*.



A variable cannot be of type void.

8.10.4 Initializing Variables

While declaring the variables, we can also initialize them with some value. For example,

```
int emp_num = 7;
float salary = 5000;
char grade = 'A';
double balance_amount = 100000000;
```

The initializer applies only to the variable defined immediately before it. Therefore, the statement

```
int count, flag = 1;
```

initializes the variable `flag` and not `count`. If you want both the variables to be declared in a single statement then write,

```
int count = 0, flag = 1;
```

When variables are declared but not initialized they usually contain garbage values (there are exceptions to this that we will study later).

8.11 CONSTANTS

Constants are identifiers whose value does not change. Variables can change their value at any time but constants can never change their value. Constants are used to define fixed values such as pi or the charge on an electron so that their value does not get changed in the program even by mistake.

A constant is an explicit data value specified by the programmer. The value of the constant is known to the compiler at the compile time. C allows the programmer to specify constants of integer type, floating point type, character type, and string type (Figure 8.8).

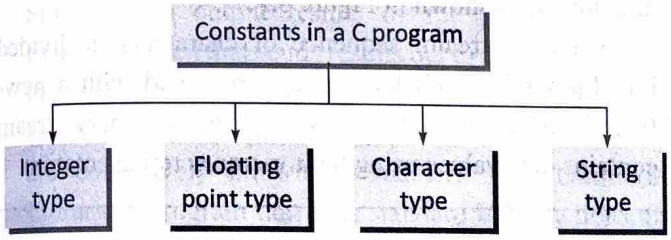


Figure 8.8 Constants in C

8.11.1 Integer Constant

A constant of integer type consists of a sequence of digits. For example, 1, 34, 567, 8907 are valid integer constants. A literal integer such as 1234 is of type `int` by default. For a long integer constant the literal is succeeded with either `'L'` or `'l'` (e.g., 1234567L). Similarly, an unsigned int literal is written with a `'U'` or `'u'` suffix (e.g., 12U). Therefore, 1234L, 1234l, 1234U, 1234u, 1234LU, 1234ul are all valid integer constants.

Integer literals can be expressed in decimal, octal, or hexadecimal notation. By default an integer is expressed in decimal notation. Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or + sign. Examples of decimal integer constants include: 123, –123, +123, and 0.

While writing integer constants, embedded spaces, commas, and non-digit characters are not allowed. Therefore, integer constants are invalid in C.

```
123  456  12,34,567  $123
```

An integer constant preceded by a zero (0) is an octal number. Octal integers consist of a set of digits, 0 through 7. Examples of octal integers include 012, 0, 01234. Similarly, an integer constant is expressed in hexadecimal notation if it is preceded with 0x or 0X. Hexadecimal numbers contains digits from 0–9 and alphabets A through F. The alphabets A through F represent numbers 10 through 15. For example, decimal 72 is equivalent to 0110 in octal notation and 0x48 in hexadecimal notation. Examples of hexadecimal integers are 0X12 0x7F 0xABCD 0X1A3B

> **Note** In C, a decimal integer constant is treated as an unsigned long if its magnitude exceeds that of signed long. An octal or hexadecimal integer that exceeds the limit of int is taken to be unsigned. If this limit is exceeded, it is taken as long; and in case this limit is also exceeded, it is treated as unsigned long.

8.11.2 Floating Point Constant

Integer numbers are inadequate to express numbers that have a fractional part. A floating point constant therefore consists of an integer part, a decimal point, a fractional part, and an exponent field containing an e or E (e means exponent) followed by an integer where the fraction part and integer part are a sequence of digits. However, it is not necessary that every floating point constant must contain all these parts. Some floating point numbers may have certain parts missing. Some valid examples of floating point numbers are: 0.02, –0.23, 123.456, +0.34 123, 0.9, –0.7, +0.8, etc.

A literal such as 0.07 is treated as type double by default. To make it a float type literal, you must specify it using suffix `'F'` or `'f'`. Some valid floating point literals are 0.02F, 0.34f, 3.141592654L, 0.002146, and 2.146E–3. (Note that suffix `L` is for `long double`).

A floating point number may also be expressed in scientific notation. In this notation, the mantissa is either a floating point number or an integer and exponent is an integer with an optional plus or minus sign. Therefore, the following are valid floating point numbers:

0.5e2 14E-2 1.2e+3 2.1E-3 -5.6e-2

Thus, we see that scientific notation is used to express numbers that are either very small or very large. For example,

120000000 = 1.2E8 and –0.000000025 = –2.5E–8

8.11.3 Character Constant

A character constant consists of a single character enclosed in single quotes. For example, `'a'`, `'@'` are character constants. In computers, characters are stored using the machine's character set using ASCII codes. All escape sequences mentioned in Table 8.1 are also character constants.

8.11.4 String Constant

A string constant is a sequence of characters enclosed in double quotes. So `"a"` is not the same as `'a'`. The characters comprising the string constant are stored in successive memory locations. When a string constant is encountered

in a C program, the compiler records the address of the first character and appends a null character (`'\0'`) to the string to mark the end of the string. Thus, length of a string constant is equal to number of characters in the string plus 1 (for the null character). Therefore, the length of string literal `"hello"` is 6.

8.11.5 Declaring Constants

To declare a constant, precede the normal variable declaration with `const` keyword and assign it a value. For example,

```
const float pi = 3.14;
```

The `const` keyword specifies that the value of `pi` cannot change.

However, another way to designate a constant is to use the preprocessor command `define`. Like other preprocessor commands, it is preceded with a # symbol. Although `#define` statements can be placed anywhere in a C program, it is always recommended that these statements be placed at the beginning of the program to make them easy to find and modify at a later stage. Look at the following example which defines values using `define`.

```
#define PI 3.14159
#define service_tax 0.12
```

In these examples, the values of `PI` and `service_tax` may change. Whenever the values are altered, it needs to be corrected only in the `define` statement.

When the preprocessor reformats the program to be compiled by the compiler, it replaces each defined name (such as `PI`, `service_tax`) in the source program with its corresponding value. Hence, it just works like the Find and Replace command available in a text editor.

Let us take a look at some rules that need to be applied to a `#define` statement which defines a constant.

Rule 1 Constant names are usually written in capital letters to visually distinguish them from other variable names which are normally written in lower case characters. Note that this is just a convention and not a rule.

Rule 2 No blank spaces are permitted in between the # symbol and `define` keyword.

Rule 3 Blank space must be used between `#define` and constant name and between constant name and constant value.

Rule 4 `#define` is a pre-processor compiler directive and not a statement. Therefore, it does not end with a semi-colon.

8.12 INPUT/OUTPUT STATEMENT IN C

Before going straight away into performing input and output in C programs let us first understand the concept from scratch. This section deals with the basic understanding of the phenomena involved in accepting input and printing output in C programs.

8.12.1 Streams

A stream acts in two ways. It is the source of data as well as the destination of data. C programs input and output data from a stream. Streams are associated with a physical device such as the monitor or with a file stored on the secondary memory. C uses two forms of streams—text and binary, as shown in Figure 8.9.

In a text stream, sequence of characters is divided into lines with each line being terminated with a new-line character (`\n`). On the other hand, a binary stream contains data values using their memory representation.

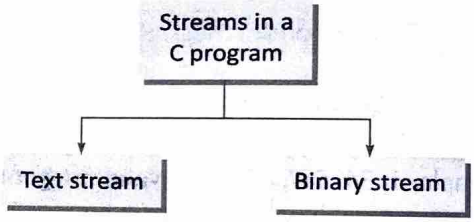


Figure 8.9 Streams in C

We can do input/output from the keyboard/monitor or from any file but in this chapter we will assume that the source of data is the keyboard and destination of the data is the monitor (Figure 8.10). File handling, that is handling input and output via C programs will be discussed later as a separate chapter.

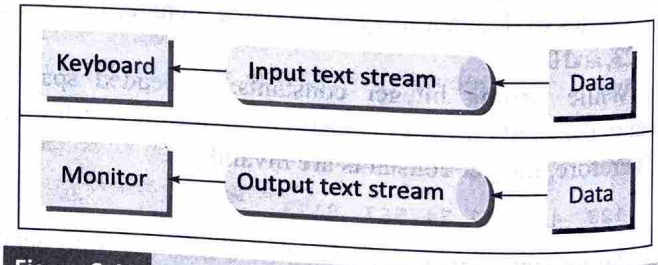


Figure 8.10 Input and output streams in C

8.12.2 Formatting Input/Output

C language supports two formatting functions `printf` and `scanf`. `printf` is used to convert data stored in the program into a text stream for output to the monitor, and `scanf` is used to convert the text stream coming from

the keyboard to data values and stores them in program variables. In this section, we will discuss these functions.

Background

The most fundamental operation in a C program is to accept input values to the program from standard input device (keyboard) and output the data produced by the program to a standard output device (monitor). So far we had been assigning values to variables using the assignment operator =. For example,

```
int a = 3;
```

However, what if we want to assign values to variables that is input by the user at run-time? This is done by using the `scanf` function that reads data from the keyboard. Similarly, to output results of the program, `printf` function is used that sends results out to a terminal. Similar to `printf` and `scanf`, there are different functions in C that can carry out the input/output operations. These functions are collectively known as standard Input/Output(I/O) Library. A program that uses standard I/O functions must contain the statement

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

at the beginning of the program.

8.12.3 printf()

The `printf` function (stands for print formatting) is used to display information required by the user and also prints the values of the variables. For this, the `printf` function takes data values, converts them to a text stream using formatting functions specified in a control string and passes the resulting text stream to the standard output. The control string may contain zero or more conversion specifications, textual data, and control characters to be displayed (Figure 8.11).

Each data value to be formatted into the text stream is described using a separate conversion specification in the control string. The specification in the control string describes the data value's type size and specific format information as shown in Figure 8.11.

The syntax of `printf` function can be given as

```
printf ("control string", variable list);
```

The function accepts two parameters—control string and variable list. The control string may also contain text to be printed such as instructions to the user, captions, identifiers, or any other text to make the output readable. In some `printf` statements you may find only a text string that has to be displayed on screen (as seen in the first program that prints `hello world`). The control characters can also be included in the `printf` statement. These control characters include `\n, \t, \r, \a,` etc.

After the control string, the function can have as many additional arguments as specified in the control string. The parameter control string in the `printf()` is nothing but a C string that contains the text that has to be written on to the standard output device.

Note that there must be enough arguments otherwise, the result will become completely unpredictable. However, if by mistake you specify more number of arguments, the excess arguments will simply be ignored. The prototype of the control string can be as given as follows:

```
%[flags] [width] [.precision] [length] specifier
```

Each control string must begin with a `%` sign. The `%` character specifies how the next variable in the list of variables has to be printed. After `%` sign follows the following parameters:

Flags specify output justification such as decimal point, numerical sign, trailing zeros, or octal, decimal, or

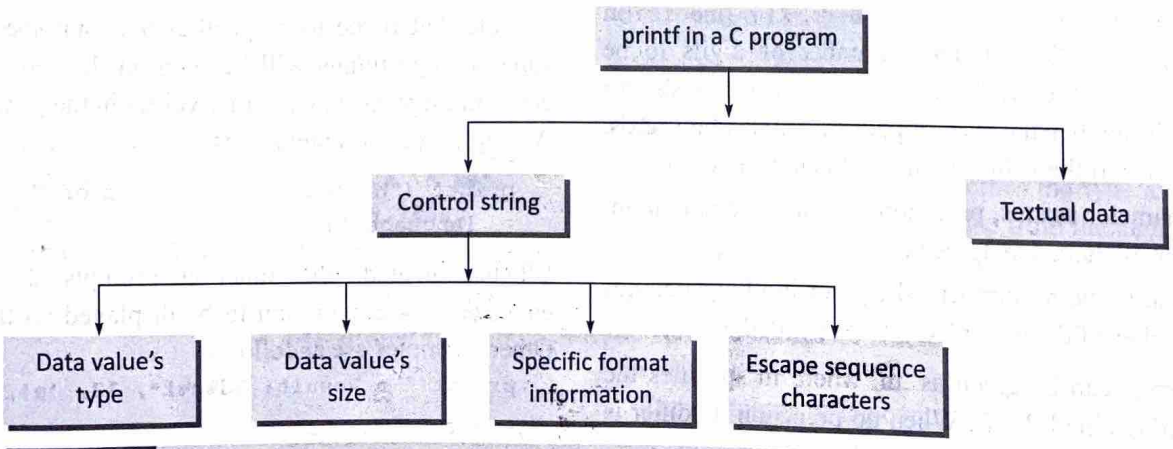


Figure 8.11 The printf() in C

hexadecimal prefixes. Table 8.5 shows the different types of flags with their decription.

Table 8.5 Flags in printf()

Flags	Description
–	Left-justify within the data given field width
+	Displays the data with its numeric sign (either + or –)
#	Used to provide additional specifiers like o, x, X, 0, 0x, or 0X for octal and hexadecimal values, respectively, for values different than zero.
0	The number is left-padded with zeros (0) instead of spaces

Note that when data is shorter than the specified width then by default the data is right justified. To left justify the data use minus sign (–) in the flags field.

When the data value to be printed is smaller than the width specified; then padding is used to fill the unused spaces. By default, the data is padded with blank spaces. If zero is used in the flag field then the data is padded with zeros. One thing to remember here is that zero flag is ignored when used with left justification because adding zeros after a number changes its value.

Width specifies the minimum number of characters to print after being padded with zeros or blank spaces, that is it specifies the minimum number of positions in the output. If data needs more space than specified, then `printf` overrides the width specified by the user. Width is a very important field especially when you have to align the output in columns. However, if the user does not mention any width then the output will take just enough room for data.

Precision specifies the maximum number of characters to print.

- For integer specifiers (`d, i, o, u, x, X`): `precision` flag specifies the minimum number of digits to be written. However, if the value to be written is shorter than this number, the result is padded with leading zeros. Otherwise, if the value is longer, it is not truncated.
- For character strings, precision specifies the maximum number of characters to be printed.
- For floating point numbers, the precision flag specifies the number of decimal places to be printed.

Its format can be given as .m, where m specifies the number of decimal digits. When no precision modifier is specified, `printf` prints six decimal positions.

When both width and precision fields are used, width must be large enough to contain the integral value of the number, the decimal point and the number of digits after the decimal point. Therefore, a conversion specification `%7.3f` means print a floating point value of maximum 7 digits where 3 digits are allotted for the digits after the decimal point.

Length field can be explained as given in Table 8.6.

Table 8.6 Length Field in printf()

Length	Description
h	When the argument is a short int or unsigned short int.
l	When the argument is a long int or unsigned long int for integer specifiers.
L	When the argument is a long double (used for floating point specifiers)

Specifier is used to define the type and the interpretation of the value of the corresponding argument (Table 8.7).

Table 8.7 Specifier field in printf()

Type	Qualifying input
c	For single character
d	For decimal values
F	For floating point numbers
E, e	Floating point numbers in exponential format
G, G	Floating point numbers in the shorter of e format
o	For octal number
s	For a sequence of (string of) characters
u	For unsigned decimal value
x, X	For hexadecimal value

Note that if the user specifies a wrong specifier then some strange things will be seen on the screen and the error might propagate to other values in the `printf()` list. A simple `printf` statement is

```
printf ("Welcome to the world of C
   language");
```

When executed, the function prompts the message enclosed in the quotation to be displayed on the screen. Other examples are as follows:

```
printf("\n Result: %d%c%f", 12, 'a', 2.3);
Result: 12a2.3
printf("\n Result: %d %c %f", 12, 'a', 2.3);
Result: 12 a 2.3
```

```
printf("\n Result: %d\t%c\t%f", 12, 'a', 2.3);
Result: 12 a  2.3
printf("\n Result: %d\t%c\t%6.2f", 12, 'a',
  245.37154);
Result: 12 a  245.37
printf("\n Result: %5d \t %x \t %#x", 234,
  234, 234);
Result:   234 EA 0xEA
printf("\n The number is %6d", 12);
The number is     12
printf("\n The number is %2d", 1234);
The number is 1234
printf("\n The number is %6d", 1234);
The number is   1234
printf("\n The number is %-6d", 1234);
The number is 1234__  //*2 _ indicates 2
  white space s */
printf("\n The number is %06d", 1234);
The number is 001234
printf("\n The price of this item is %09.2f
  rupees", 123.456);
```

Programming Tip: Not placing a comma after the format string in a read or write statement is a compiler error.

```
The price of this item is
00123.45 rupees
printf("\n This is \'so\'
beautiful");
This is 'so' beautiful
printf("\n This is \"so\"
beautiful");
This is "so" beautiful
printf("\n This is \\ so beautiful ");
This is \ so beautiful
printf("\n a = |%-+7.2f| b = %0+7.2f c =
  %-0+8.2f", 1.2, 1.2, 1.2);
a = +1.20     b = 0001.20 c = 1.20
```

(Note that in this example, – means left justify, + means display the sign, 7 specifies the width, and 2 specifies the precision.)

```
printf("\n %7.4f \n %7.2f \n %-7.2f \n %f
  \n %10.2e \n %11.4e \n %-10.2e \n %e",
  98.7654, 98.7654, 98.7654, 98.7654,
  98.7654, 98.7654, 98.7654, 98.7654);
98.7654
  98.77
98.77
98.7654
   9.88e+01
9.8765e+01
9.88e+01
9.876540e+01
```

Programming Tip: Placing an address operator with a variable in the printf statement will generate a run-time error.

```
char ch = 'A';
printf("\n %c \n %3c \n %5c", ch, ch, ch);
A
   A
     A
char str[] = "Good Morning";
printf("\n %s", str);
printf("\n %20s", str);
printf("\n %20.10s", str);
printf("\n %.7s", str);
printf("\n %-20.10s", str);
printf("\n %7s", str);

Good Morning
          Good Morning
            Good Morni
Good Mo
Good Morni
Good Morning
```

(Note that in the last printf statement the complete string "Good Morning" is printed. This is because if data needs more space than specified, then printf overrides the width specified by the user.)

8.12.4 scanf()

The function scanf() stands for scan formatting and is used to read formatted data from the keyboard. This function takes a text stream from the keyboard, extracts and formats data from the stream according to a format control string and then stores the data in specified program variables. The syntax of the scanf() can be given as,

```
scanf("control string", arg1, arg2, arg3, ...,
  argn);
```

Note The minimum field width and precision specifiers are usually constants. However, they may also be provided by arguments to printf(). This is done by using the * modifier as shown in the following statement:

```
printf("%*.*f", 10, 4, 1234.34);
```

Here, the minimum field width is 10, the precision is 4, and the value to be displayed is 1234.34.

The control string specifies the type and format of the data that has to be obtained from the keyboard and stored in the memory locations pointed by the arguments arg1, arg2,..., argn, i.e., the arguments are actually the variable addresses where each piece of data are to be stored.

The prototype of the control string can be given as:

```
[=%[*] [width] [modifiers] type=]
```

Here * is an optional argument that suppresses assignment of the input field, i.e., it indicates that data should be read from the stream but ignored (not stored in the memory location).

Width It is an optional argument that specifies the maximum number of characters to be read. However, fewer characters will be read if the `scanf` function encounters a white space or an inconvertible character because the moment `scanf` encounters a white space character it will stop processing further.

Modifier It is an optional argument that can be `h`, `l`, or the field `L` for the data pointed by the corresponding additional arguments. Modifier `h` is used for `short int` or `unsigned short int`, `l` is used for `long int`, `unsigned long int`, or `double` values. Finally, `L` is used for `long double` data values.

Type This specifies the type of data that has to be read. It also indicates how this data is expected to be read from the user. The type specifiers for `scanf` function are given in Table 8.8.

Table 8.8 Type specifiers for scanf()

Type	Qualifying input
c	For single character
d	For decimal values
F	For floating point numbers
E, e	Floating point numbers in exponential format
G, G	Floating point numbers in the shorter of e format
o	For octal number
s	For a sequence of (string of) characters
u	For unsigned decimal value
x, X	For hexadecimal value

The `scanf` function ignores any blank spaces, tabs, and newlines entered by the user. The function simply returns the number of input fields successfully scanned and stored.

We will not discuss functions in detail in this chapter. So understanding `scanf` function in depth might be a bit difficult here, but for now just understand that the `scanf` function is used to store values in memory locations associated with variables. For this, the function should have the address of the variables. The address of the variable is denoted by an '&' sign followed by the name of the variable.

Note Whenever data is read from the keyboard, there is always a return character from a previous read operation. So we should always code at least one white space character in the conversion specification in order to flush that whitespace character. For example, to read two or more data values together in a single scanf statement, we must insert a white space between two fields as shown below:

```
scanf("%d %c", &i, &ch);
```

Now let us quickly summarize the rules to use the `scanf` function in our C programs.

Rule 1 The `scanf` function works until

(a) the maximum number of characters has been processed,

(b) a white space character is encountered, or

(c) an error is detected.

Rule 2 Every variable that has to be processed must have a conversion specification associated with it. Therefore, the following `scanf` statement will generate an error as `num3` has no conversion specification associated with it.

```
scanf("%d %d", &num1, &num2, &num3);
```

Rule 3 There must be a variable address for each conversion specification. Therefore, the following `scanf` statement will generate an error as no variable address is given for the third conversion specification.

```
scanf("%d %d %d", &num1, &num2);
```

Remember that the ampersand operator `(&)` before each variable name specifies the address of that variable name.

Rule 4 A fatal error would be generated if the format string is ended with a white space character.

Rule 5 The data entered by the user must match the character specified in the control string (except white space or a conversion specification), otherwise an error will be generated and `scanf` will stop its processing. For example, consider the following `scanf` statement:

```
scanf("%d/%d", &num1, &num2);
```

Here, the slashes in the control string are neither white space characters nor a part of conversion specification, so the users must enter data of the form `21/46`.

Rule 6 Input data values must be separated by spaces.

Rule 7 Any unread data value will be considered as a part of the data input in the next call to `scanf`.

Rule 8 When the field width specifier is used, it should be large enough to contain the input data size.

Look at the following code that shows how we input values in variables of different data types.

```
int num;
scanf(" %d ", &num);
```

The scanf function reads an integer value (because the type specifier is %d) into the address or the memory location pointed by num.

```
float salary;
scanf(" %f ", &salary);
```

The scanf function reads a floating point number (because the type specifier is %f) into the address or the memory location pointed by salary.

```
char ch;
scanf(" %c ", &ch);
```

The scanf function reads a single character (because the type specifier is %c) into the address or the memory location pointed by ch.

```
char str[10];
scanf(" %s ", str);
```

Programming Tip:
A compiler error will be generated if the read and write parameters are not separated by commas.

The scanf function reads a string or a sequence of characters (because the type specifier is %s) into the address or the memory location pointed by str. Note that in the case of reading the string, we do not use the & sign in the scanf function. This will be discussed in Chapter 12.

Look at the following code which combines reading of variables of different data types in one single statement.

```
int num;
float fnum;
char ch;
char str[10];
scanf("%d %f %c %s", &num, &fnum, &ch, str);
```

Look at the following scanf statement for the same code. The statement ignores the character variable and does not store it (as it is preceded by *).

```
scanf("%d %f %*c %s", &num, &fnum, &ch, str);
```

Remember that if an attempt is made to read a value that does not match the expected data type, the scanf function will not read any further and would immediately return the values read.

8.12.5 Examples of printf/scanf

Look at the following code that shows how we output values of variables of different data types.

```
int num;
scanf(" %d ", &num);
printf("%d", num);
```

The printf function prints an integer value (because the type specifier is %d) pointed by num on the screen.

```
float salary;
scanf(" %f ", &salary);
printf(".2%f", salary);
```

The printf function prints the floating point number (because the type specifier is %f) pointed by salary on the screen. Here, the control string specifies that only two digits must be displayed after the decimal point.

Programming Tip:
A float specifier cannot be used to read an integer value.

```
char ch;
scanf(" %c ", &ch);
printf("%c", ch);
```

The printf function prints a single character (because the type specifier is %c) pointed by ch on the screen.

```
char str[10];
scanf(" %s ", str);
```

The printf function prints a string or a sequence of characters (because the type specifier is %s) pointed by str on the screen.

```
scanf("%2d %5d", &num1, &num2);
```

The scanf statement will read two integer numbers. The first integer number will have two digits while the second can have maximum of 5 digits.

Look at the follwoing code which combines printing all these variables of different data types in one single statement.

```
int num;
float fnum;
char ch;
char str[10];
double dnum;
short snum;
long int lnum;
printf("\n Enter the values : ");
scanf("%d %f %c %s %e %hd %ld", &num, &fnum,
   &ch, str, &dnum, &snum, &lnum);
```

```
printf("\n num = %d \n fnum = %f \n ch = %c
  \n str = %s \n dnum = %e \n snum = %hd \n
  lnum = %ld", num, fnum, ch, str, dnum,
  snum, lnum);
```

Note: In the printf statement, '\n', is called the newline character and is used to print the succeding text on the new line. The following output will be generated on execution of the print function.

```
Enter the values
2 3456.443 a abcde 24.321E-2 1 12345678
num = 2
fnum = 3456.44
ch = a
str = abcde
dnum = 0.24321
snum = 1
lnum = 12345678
```

Remember that scanf terminates as soon as it encounters a white space character so if you enter the string as abc def, then only abc is assigned to str.

1. Find out the output of the following program.

```
#include <stdio.h>
main()
{
   int a, b;
   printf("\n Enter two four digit numbers : ");
   scanf("%2d %4d", &a, &b);
   printf("\n The two numbers are : %d and
     %d", a, b);
   return 0;
}
```

Output

```
Enter two four digit numbers : 1234 5678
The two numbers are : 12 and 34
```

Programming Tip: Using an incorrect specifier for the data type being read or written will generate a run-time error.

Here, the variable a is assigned the value 12 because it is specified as %2d, so it will accept only the first two digits. The rest of the number will be assigned to b. The value 5678 that is unread will be assigned to the first variable in the next call to the scanf function.

Note: The %n specifier is used to assign the number of characters read till the point at which the %n was encountered by the variable pointed to by the corresponding argument. The following code fragment illustrates its usage:

```
int count;
printf("Hello %n World!", &count);
printf("%d", count);
```

The output would be Hello World!6 because 6 (H, e, l, l, o,) is the number of characters read before the %n modifier.

2. Write a program to demonstrate the use of printf statement to print values of variables of different data types.

```
#include <stdio.h>
main()
{
   // Declare and initialize variables
   int num = 7;
   float amt = 123.45;
   char code = 'A';
   double pi = 3.1415926536;
   long int population_of_india =
   10000000000;
   char msg[] = "Hi";

   // Print the values of variables
   printf("\n NUM = %d \t AMT = %f \t CODE
   = %c \n PI = %e \t POPULATION OF INDIA =
   %ld \n MESSAGE = %s", num, amt, code, pi,
   population_of_india, msg);
   return 0;
}
```

Output

```
NUM = 7
AMT = 123.45000
CODE = A
PI = 3.141590e+00
POPULATION OF INDIA = 10000000000
MESSAGE = Hi
```

3. Write a program to demonstrate the use of printf and scanf statements to read and print values of variables of different data types.

```
#include <stdio.h>
main()
{
   int num;
   float amt;
   char code;
```

```
    double pi;
    long int population_of_india;
    char msg[10];

    printf("\n Enter the value of num : ");
    scanf("%d", &num);
    printf("\n Enter the value of amt : ");
    scanf("%f", &amt);
    printf("\n Enter the value of pi : ");
    scanf("%e", &pi);
    printf("\n Enter the population of india : ");
    scanf("%ld", &population_of_india);
    printf("\n Enter the value of code : ");
    scanf("%c", &code);
    printf("\n Enter the message : ");
    scanf("%s", msg);
    printf("\n NUM = %d \n AMT = %f \n PI =
      %e \n POPULATION OF INDIA = %ld \n
      CODE = %c \n MESSAGE = %s", num,
      amt, code, pi, population_of_india,
      msg);
    return 0;
}
```

Output

```
Enter the value of num : 5
Enter the value of amt : 123.45
Enter the value of pi : 3.14159
Enter the population of india : 12345
Enter the value of code : c
Enter the message : Hello

NUM = 5
AMT = 123.45000
PI = 3.141590e+00
POPULATION OF INDIA = 12345
CODE = c
MESSAGE = Hello
```

4. Write a program to calculate the area of a triangle using Hero's formula.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
int main()
{
   float a, b, c, area, S;

   printf("\n Enter the lengths of the three
     sides of the triangle : ");
   scanf("%f %f %f", &a, &b, &c);
   S = ( a + b + c)/2;

   /* sqrt is a mathematical function defined
     in math.h header file
   area = sqrt(S*(S-a)*(S-b)*(S-c));
   printf("\n Area = %f", area);
   return 0;
}
```

Output

```
Enter the lengths of the three sides of the
  triangle : 12 16 20
Area = 96
```

5. Write a program to calculate the distance between two points.

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

int main()
{
   int x1, x2, y1, y2;
   float distance;
   printf("\n Enter the x and y coordinates
     of the first point : ");
   scanf("%d %d", &x1, &y1);
   printf("\n Enter the x and y coordinates
     of the second point :");
   scanf("%d %d", &x2, &y2);

   /* sqrt and pow are mathematical
     functions defined in math.h header file
   distance = sqrt(pow((x2-x1), 2)+
     pow((y2-y1), 2));
   printf("\n Distance = %f", distance);
   return 0;
}
```

Output

```
Enter the x and y coordinates of the first
  point : 2 5
Enter the x and y coordinates of the second
  point : 3 7
Distance = 2.236068
```

8.12.6 Detecting Errors During Data Input

When the scanf function completes reading all the data values, it returns number of values that are successfully read. This return value can be used to determine whether there was any error while reading the input. For example, the statement,

```
scanf("%d %f %c", &a, &b, &c);
```

will return 3 if the user enters, say,

```
12 12.34 A
```

It will return 1 if the user enters erroneous data such as

```
12 ABC 12.34
```

This is because a string was entered while the user was expecting a floating point value. So, scanf reads only first data value correctly and then terminates as soon as it encounters a mismatch between the type of data expected and the type of data entered.

8.13 OPERATORS IN C

An operator is defined as a symbol that specifies the mathematical, logical, or relational operation to be performed. C language supports a lot of operators to be used in expressions. These operators can be categorized into the following major groups:

- Arithmetic operators
- Relational operators
- Equality operators
- Logical operators
- Unary operators
- Conditional operators
- Bitwise operators
- Assignment operators
- Comma operator
- Sizeof operator

In this section, we will discuss about all these operators.

8.13.1 Arithmetic Operators

Consider three variables declared as,

```
int a = 9, b = 3, result;
```

We will use these variables to explain arithmetic operators. Table 8.9 shows the arithmetic operators, their syntax, and usage in C language.

Table 8.9 Arithmetic operators

Operation	Operator	Syntax	Comment	Result
Multiply	*	a * b	result = a * b	27
Divide	/	a / b	result = a/b	3
Addition	+	a + b	result = a + b	12
Subtraction	–	a – b	result = a – b	6
Modulus	%	a % b	result = a % b	0

In Table 8.9, a and b (on which the operator is applied) are called operands. Arithmetic operators can be applied to any integer or floating-point number. The addition, subtraction, and multiplication (+, -, and *) operators perform the usual arithmetic operations in C programs.

However, the operator % must be new to you. The modulus operator (%) finds the remainder of an integer division. This operator can be applied only to integer operands and cannot be used on float or double operands. Therefore, the code given below generates a compiler error.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   float c = 20.0;
   printf("\n Result = %f", c % 5);
/* WRONG. Modulus operator is being applied
   to a float operand */
   return 0;
}
```

While performing modulo division, the sign of the result is always the sign of the first operand (the dividend). Therefore,

16 % 3 = 1 – 16 % 3 = –1

16 % – 3 = 1 – 16 % -3 = –1

When both operands of the division operator (/) are integers, the division is performed as an integer division. Integer division always results in an integer result. So, the result is always rounded-off by ignoring the remainder. Therefore,

9/4 = 2 and –9/4 = –3

From this observation, we can conclude two things. If op1 and op2 are integers and the quotient is not an integer, then we have two cases:

- If op1 and op2 have the same sign, then op1/op2 is the largest integer less than the true quotient.
- If op1 and op2 have opposite signs, then op1/op2 is the smallest integer greater than the true quotient.

Note that it is not possible to divide any number by zero. This is an illegal operation that results in a run-time division-by-zero exception, thereby terminating the program.

Except for the modulus operator, all other arithmetic operators can accept a mix of integer and floating point numbers. If both operands are integers, the result will be an integer. If one or both operands are floating point numbers, then the result would also be a floating point number.

All the arithmetic operators bind from left to right. As in mathematics the multiplication, division, and modulus operators have higher precedence over the addition and subtraction operators, i.e., if an arithmetic expression consists of a mix of operators, then multiplication, division, and modulus will be carried out first in a left to right order, before any addition and subtraction could be performed. For example,

```
3 + 4 * 7 = 3 + 28 = 31
```

6. Write a program to perform addition, subtraction, division, integer division, multiplication, and modulo division on two integer numbers.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int num1, num2;
   int add_res = 0, sub_res = 0, mul_res =
     0, idiv_res = 0, modiv_res = 0;
   float fdiv_res = 0.0;
   clrscr();
   printf("\n Enter the first number : ");
```

```
    scanf("%d", &num1);
    printf("\n Enter the second number : ");
    scanf("%d", &num2);

    add_res = num1 + num2;
    sub_res = num1 - num2;
    mul_res = num1 * num2;
    idiv_res = num1 / num2;
    modiv_res = num1 % num2;
    fdiv_res = (float)num1 / num2;
    printf("\n %d + %d = %d", num1, num2,
      add_res);
    printf("\n %d - %d = %d", num1, num2,
      sub_res);
    printf("\n %d x %d = %d", num1, num2,
      mul_res);
    printf("\n %d / %d = %d (Integer
      division)", num1, num2, idiv_res);
    printf("\n %d %% %d = %d (Moduluo
      division)", num1, num2, modiv_res);
    printf("\n %d / %d = %.2f (Normal
      division)", num1, num2, fdiv_res);
    return 0;
}
```

Output

```
Enter the first number : 9
Enter the second number : 7
9 + 7 = 16
9 - 7 = 2
9 * 7 = 63
9 / 7 = 1 (Integer division)
9 % 7 = 2 (Moduluo division)
9 / 7 = 1.29 (Normal division)
```

7. Write a program to perform addition, subtraction, division, and multiplication on two floating point numbers.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   float num1, num2;
   clrscr();
   printf("\n Enter the first number: ");
   scanf("%f", &num1);
   printf("\n Enter the second number: ");
   scanf("%f", &num2);

   printf("\n %f + %f = %f", num1, num2,
     num1 + num2);
   printf("\n %f - %f = %f", num1, num2,
     num1 - num2);
   printf("\n %f x %f = %f", num1, num2,
     num1 * num2);
   printf("\n %f / %f = %f ", num1, num2,
     num1 / num2);
   return 0;
}
```

Output

```
Enter the first number : 3.5
Enter the second number : 1.2
3.5 + 1.2 = 4.700000
3.5 - 1.2 = 2.300000
3.5 * 1.2 = 4.200000
3.5 / 1.2 = 2.196667
```

8. Write a program to subtract two long integers.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   long int num1 = 1234567, num2, diff = 0;
   clrscr();
   printf("\n Enter the number: ");
   scanf("%ld", &num2);
   diff = num1 - num2;
   printf("\n Difference = %ld", diff);
   return 0;
}
```

Output

```
Enter the number: 1234
Difference = 1233333
```

8.13.2 Relational Operators

A relational operator, also known as a comparison operator, is one that compares two values. Expressions that contain relational operators are called *relational expressions*. Relational operators return true or false value, depending on whether the conditional relationship between the two operands holds or not.

For example, to test the expression, if x is less than y, relational operator < is used as x < y. This expression will return TRUE if x is less than y; otherwise the value of the expression will be FALSE.

Relational operators can be used to determine the relationships between the operands. These relationships are illustrated in Table 8.10.

Table 8.10 Relational operators

Operator	Meaning	Example
<	Less than	3 < 5 gives 1
>	Greater than	7 > 9 gives 0
>=	Less than or equal to	100 >= 100 gives 1
<=	Greater than equal to	50 >= 100 gives 0

The relational operators are evaluated from left to right. The operands of a relational operator must evaluate to a number. Characters are considered valid operands since

they are represented by numeric values in the computer system. So, if we say, `'A' < 'B'`, where A is 65 and B is 66 then the result would be `1` as `65 < 66`.

When arithmetic expressions are used on either side of a relational operator, then first the arithmetic expression will be evaluated and then the result will be compared. This is because arithmetic operators have a higher priority over relational operators.

However, relational operators should not be used for comparing strings as this will result in comparing the address of the string and not their contents. You must be wondering why so? The answer to this question will be clear to you in the later chapters. A few examples of relational operators are as follows.

If `x = 1, y = 2,` and `z = 3`, then expressions that evaluate to

TRUE (value not zero)	FALSE (value is zero)
`(x)`	`(x - 1)`
`(x + y)`	`(!(z))`
`(z * 9)`	`( 0 * y)`
`(z + 10 - 5 * a)`	`(y == 1)`
`( z - x + y)`	`( y % 2)`

Note: Although blank spaces are allowed between an operand and an operator, no space is permitted between the components of an operator (> = is not allowed, it should be >=). Therefore, x==y is correct, but x = = y is not acceptable in C language.

9. Write a program to show the use of relational operators.

```
#include <stdio.h>
{
   main()
   int x = 10, y = 20;
   printf("\n %d < %d = %d", x, y, x<y);
   printf("\n %d == %d = %d", x, y, x==y);
   printf("\n %d != %d = %d", x, y, x!=y);
   printf("\n %d > %d = %d", x, y, x>y);
   printf("\n %d >= %d = %d", x, y, x>=y);
   printf("\n %d <= %d = %d", x, y, x<=y);
   return 0;
}
```

Output

```
10 < 20 = 1
10==20 = 1
10 != 20 = 1
10 > 20 = 0
10 >= 20 = 0
10 <= 20 = 1
```

8.13.3 Equality Operators

C language supports two kinds of equality operators to compare their operands for strict equality or inequality. They are equal to `(==)` and not equal to `(!=)` operators. The equality operators have lower precedence than the relational operators.

The equal to operator `(==)` returns `true (1)` if operands on both the sides of the operator have the same value; otherwise, it returns `false (0)`. On the contrary, the not-equal-to operator `(!=)` returns `true(1)` if the operands do not have the same value; else it returns `false (0)`. Table 8.11 summarizes equality operators.

Table 8.11 Equality operators

Operator	Meaning
==	Returns 1 if both operands are equal, 0 otherwise
!=	Returns 1 if operands do not have the same value, 0 otherwise

8.13.4 Logical Operators

C language supports three logical operators—logical AND `(&&)`, logical OR `(||)`, and logical NOT `(!)`. As in case of arithmetic expressions, the logical expressions are evaluated from left to right.

Logical AND

Logical AND operator is used to simultaneously evaluate two conditions or expressions with relational operators. If expressions on both the sides (left and right side) of the logical operator is true then the whole expression is true. The truth table of logical AND operator is given in Table 8.12.

Table 8.12 Truth table of logical AND

A	B	A &&B
0	0	0
0	1	0
1	0	0
1	1	1

For example, `(a < b) && (b > c)`

The expression to the left is `(a<b)` and that on the right is `(b > c)`. The whole expression is true only if both expressions are true, i.e., if b is greater than a and c.

Logical OR

Logical OR operator is used to simultaneously evaluate two conditions or expressions with relational operators. If one or both the expressions on the left side and right side of the logical operator is true then the whole expression is true. The truth table of logical OR operator is given in Table 8.13.

Table 8.13 Truth table of logical OR

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

For example, `(a < b) || (b > c)`

The expression to the left is `(a < b)` and that on the right is `(b > c)`. The whole expression is true if either `b` is greater than `a` or `b` is greater than `c`.

Logical NOT

The logical NOT operator takes a single expression and negates the value of the expression. That is, logical NOT produces a `zero` if the expression evaluates to a non-zero value and produces a `1` if the expression produces a zero. In other words, it just reverses the value of the expression. The truth table of logical NOT operator is given in Table 8.14.

Table 8.14 Truth table of Logical NOT

A	!A
0	1
1	0

For example,

```
int a = 10, b;
b = !a;
```

Now the value of `b = 0`. This is because value of `a = 10, !a = 0`. The value of `!a` is assigned to `b`, hence, the result.

Logical expressions operate in a short cut fashion and stop the evaluation when it is known for sure what the final outcome would be. For example, in a logical expression involving logical AND, if the first operand is false, then the second operand is not evaluated as it is for sure that the result will be false. Similarly, for a logical expression involving logical OR, if the first operand is true, then the second operand is not evaluated as it is for sure that the result will be true.

However this approach has a side effect. For example, consider the following expressions:

```
(x > 9) && (y > 0)
(x > 9) || (y > 0)
```

In the logical AND expression, if the first operand is false then the entire expression will not be evaluated and thus the value of `y` will never be incremented. Same is the case with the logical OR expression. If the first expression is true then the second will never be evaluated and value of `y` will never be incremented.

8.13.5 Unary Operators

Unary operators act on single operands. C language supports three unary operators—unary minus, increment, and decrement operators.

Unary Minus

Unary minus (–) operator is strikingly different from the binary arithmetic operator that operates on two operands and subtracts the second operand from the first operand. When an operand is preceded by a minus sign, the unary operator negates its value. For example, if a number is positive then it becomes negative when preceded with a unary minus operator. Similarly, if the number is negative, it becomes positive after applying the unary minus operator. For example,

```
int a, b = 10;
a = -(b);
```

The result of this expression is `a = -10`, because a variable `b` has a positive value. After applying unary minus operator `(-)` on the operand `b`, the value becomes `–10`, which indicates it as a negative value.

Increment (++) and Decrement (––) Operators

The increment operator is a unary operator that increases the value of its operand by `1`. Similarly, the decrement operator decreases the value of its operand by 1. For example, `--x` is equivalent to writing `x = x - 1`.

The increment/decrement operators have two variants—prefix and postfix. In a prefix expression `(++x or --x)`, the operator is applied before an operand is fetched for computation and thus, the altered value is used for the computation of the expression in which it occurs. On the contrary, in a postfix expression `(x++ or x--)` an operator is applied after an operand is fetched for computation. Therefore, the unaltered value is used for the computation of the expression in which it occurs.

Therefore, an important point to note about unary increment and decrement operators is that `++x` is not the same as `x++`. Similarly, `--x` is not the same as `x--`. Both `++x` and `x++` increment the value of `x` by `1`. In the former case, the value of `x` is returned before it is incremented whereas, in the latter case, the value of `x` is returned after it is incremented. For example, `int x = 10, y;y = x++;` is equivalent to writing

```
y = x;
x = x + 1;
```

whereas, `y = ++x;` is equivalent to writing

```
x = x + 1;
y = x;
```

The same principle applies to unary decrement operators. The unary operators have a higher precedence than the binary operators. If in an expression, we have more than one unary operator, then unlike arithmetic operators, they are evaluated from right to left.

When applying the increment or decrement operator, the operand must be a variable. This operator can never be applied to a constant or an expression. Therefore the following codes will generate a compiler error.

Note When postfix ++ or −− is used with a variable in an expression, then the expression is evaluated first using the original value of the variable and then the variable is incremented or decremented by one. Similarly, when prefix ++ or −− is used with a variable in an expression, then the variable is first incremented or decremented and then the expression is evaluated using the new value of the variable.

10. Write a program to illustrate the use of unary prefix increment and decrement operators.

```
#include <stdio.h>
main()
{
   int num = 3;
   // Using unary prefix increment operator
   printf("\n The value of num = %d", num);
   printf("\n The value of ++num = %d", ++num);
   printf("\n The new value of num = %d", num);
   // Using unary prefix decrement operator
   printf("\n\n The value of num = %d", num);
   printf("\n The value of --num = %d", --num);
   printf("\n The new value of num = %d", num);
   return 0;
}
```

Output

```
The value of num = 3
The value of ++num = 4
The new value of num = 4

The value of num = 4
The value of --num = 3
The new value of num = 3
```

11. Write a program to illustrate the use of unary postfix increment and decrement operators.

```
#include <stdio.h>
main()
{
   int num = 3;
   // Using unary postfix increment operator
   printf("\n The value of num = %d", num);
   printf("\n The value of num++ = %d", num++);
   printf("\n The new value of num = %d", num);
   //Using unary postfix decrement
     operator
   printf("\n\n The value of num = %d", num);
   printf("\n The value of num = %d", num--);
   printf("\n The new value of num = %d", num);
   return 0;
}
```

Output

```
The value of num = 3
The value of num++ = 3
The new value of num = 4

The value of num = 4
The value of num-- = 4
The new value of num = 3
```

8.13.6 Conditional Operator

The conditional operator or the ternary `(?:)` is just like an `if–else` statement that can be within expressions. Such an operator is useful in situations in which there are two or more alternatives for an expression. The syntax of the conditional operator is

```
exp1 ? exp2 : exp3
```

`exp1` is evaluated first. If it is true, then `exp2` is evaluated and becomes the result of the expression, otherwise `exp3` is evaluated and becomes the result of the expression. For example, `large = ( a > b) ? a : b`

The conditional operator is used to find the largest of two given numbers. First `exp1`, that is `(a > b)` is evaluated. If `a` is greater than `b`, then `large = a`, else `large = b`. Hence, `large` is equal to either `a` or `b` but not both.

Hence, conditional operator is used in certain situations, replacing `if-else` condition phrases. Conditional operators make the program code more compact, more readable, and safer to use, as it is easier to check any error (if present) in one single line itself. Conditional operator is also known as ternary operator as it is neither a unary nor a binary operator; it takes three operands.

Since a conditional operator is itself an expression, it can be used as an operand of another conditional operation. That means C allows you to have nested conditional expressions. Consider the following expression which illustrates this concept.

```
int a = 5, b = 3, c = 7, small;
small = (a<b? (a<c?a:c) : (b<c?b:c));
```

12. Write a program to find the largest of two numbers using the ternary operator.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int num1, num2, large;
   clrscr();
   printf("\n Enter the first number: ");
   scanf("%d", &num1);
   printf("\n Enter the second number: ");
   scanf("%d", &num2);
   large = num1>num2? num1 : num2;
   printf("\n The largest number is: %d", large);
   return 0;
}
```

Output

```
Enter the first number: 23
Enter the second number: 34
The largest number is: 34
```

13. Write a program two find the largest of three numbers using ternary operator.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int num1, num2, num3, large;
   clrscr();
   printf("\n Enter the first number: ");
   scanf("%d", &num1);
   printf("\n Enter the second number: ");
   scanf("%d", &num2);
   printf("\n Enter the third number: ");
   scanf("%d", &num3);
   large = num1 > num2 ? (num1 > num3 ? num1
     : num3) : (num2 > num3 ? num2 : num3);
   printf("\n The largest number is: %d",
     large);
   retun 0;
}
```

Output

```
Enter the first number: 12
Enter the second number: 34
Enter the third number: 23
The largest number is: 34
```

8.13.7 Bitwise Operators

As the name suggests, bitwise operators are those operators that perform operations at the bit level. These operators include bitwise AND, bitwise OR, bitwise XOR, and shift operators. The bitwise operators expect their operands to be integers and treat them as a sequence of bits.

Bitwise AND

The bitwise AND operator (&) is a smaller version of the boolean AND (&&) as it performs operation on bits instead of bytes, chars, integers, etc. When we use the bitwise AND operator, the bit in the first operand is ANDed with the corresponding bit in the second operand. The truth table is same as we had seen in the logical AND operation, i.e., the bitwise AND operator compares each bit of its first operand with the corresponding bit of its second operand. If both bits are 1, the corresponding bit in the result is 1 or 0 otherwise. For example,

```
10101010 & 01010101 = 00000000
```

In a C program, the & operator is used as follows.

```
int a = 10, b = 20, c = 0;
c = a&b;
```

Bitwise OR

The bitwise OR operator (|) is a smaller version of the boolean OR (||) as it performs operation on bits instead of bytes, chars, integers, etc. When we use the bitwise OR operator, the bit in the first operand is ORed with the corresponding bit in the second operand. The truth table is same as we had seen in the logical OR operation, i.e., the bitwise OR operator compares each bit of its first operand with the corresponding bit of its second operand. If one or both bits are 1, the corresponding bit in the result is 1 or 0 otherwise. For example,

```
10101010 101010101 = 11111111
```

In a C program, the | operator is used as follows.

```
int a = 10, b = 20, c = 0;
c = a|b
```

Bitwise XOR

The bitwise XOR operator (^) performs operation on individual bits of the operands. When we use the

bitwise XOR operator, the bit in the first operand is XORed with the corresponding bit in the second operand. The truth table of the bitwise XOR operator is as shown in Table 8.15.

The bitwise XOR operator compares each bit of its first operand with the corresponding bit of its second operand. If one of the bits is 1, the corresponding bit in the result is 1 or 0 otherwise. For example,

```
10101010 ^ 01010101
= 11111111
```

Table 8.15 Truth table of bitwise XOR

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

In a C program, the ^ operator is used as follows:

```
int a = 10, b = 20,
c = 0;
c = a^b
```

Bitwise NOT

The bitwise NOT, or complement, is a unary operation that performs logical negation on each bit of the operand. By performing negation of each bit, it actually produces the 1s complement of the given binary value. Bitwise NOT operator sets the bit to 1 if it was initially 0 and sets it to 0 if it was initially 1. For example,

```
~10101011 = 01010100
```

Note Bitwise operators are used for testing the bits or shifting them left or right. Always remember that bitwise operators cannot be applied to float or double variables.

Shift Operator

C supports two bitwise shift operators. They are shift-left (<<) and shift-right (>>). These operations are simple and are responsible for shifting bits either to the left or to the right. The syntax for a shift operation can be given as

```
operand op num
```

where the bits in operand are shifted left or right depending on the operator (left if the operator is << and right if the operator is >>) by the number of places denoted by num. For example, if we have `x = 0001 1101`, then `x << 1` produces `0011 1010`

When we apply a left-shift, every bit in x is shifted to the left by one place. So, the MSB (most significant bit) of x is lost, the LSB of x is set to 0.

Therefore, if we have `x = 0001 1101`, then `x << 4` produces `1010 0000`.

If you observe carefully, you will notice that shifting once to the left multiplies the number by 2. Hence, multiple shifts of 1 to the left, results in multiplying the number by 2 over and over again.

On the contrary, when we apply a shift-right operator, every bit in x is shifted to the right by one place. So, the LSB (least significant bit) of x is lost, the MSB of x is set to 0. For example, if we have `x = 0001 1101`, then `x >> 1` produces `= 0000 1110`

Similarly, if we have `x = 0001 1101`, then `x << 4` produces `0000 0001`.

If you observe carefully, you will notice that shifting once to the right divides the number by 2. Hence, multiple shifts of 1 to the right, results in dividing the number by 2 over and over again.

8.13.8 Assignment Operators

In C, the assignment operator is responsible for assigning values to the variables. While the equal sign (=) is the fundamental assignment operator, C language also supports other assignment operators that provide simpler ways to represent common variable assignments.

When an equal sign is encountered in an expression, the compiler processes the statement on the right side of the sign and assigns the result to the variable on the left side. For example,

```
int x;
x = 10;
```

assigns the value 10 to variable x. If we have,

```
int x = 2, y = 3, sum = 0;
sum = x + y;
```

then sum = 5.

The assignment operator has right-to-left associativity, so the expression

```
a = b = c = 10;
```

is evaluated as

```
(a = (b = (c = 10)));
```

First 10 is assigned to c, then the value of c is assigned to b. Finally, the value of b is assigned to a.

The operand to the left of the assignment operator must always be a variable name. C does not allow any expression, constant, or function to be placed to the left of the assignment operator. Therefore, the statement a + b = 0, is invalid in C language.

To the right of the assignment operator you may have an arbitrary expression. In that case, the expression would be evaluated and the result would be stored in the location denoted by the variable name.

Other Assignment Operators

C language supports a set of shorthand assignment operators of the form

```
variable op = expression
```

where op is a binary arithmetic operator.

Table 8.16 contains a list of other assignment operators that are supported by C. The advantage of using shorthand assignment operators are as follows:

- Shorthand expressions are easier to write as the expression on the left side need not be repeated.
- The statements involving shorthand operators are easier to read as they are more concise.
- The statements involving shorthand operators are more efficient and easy to understand.

14. Write a program to demonstrate the use of assignment operators.

```
#include <stdio.h>
main()
{
   int num1 = 3, num2 = 5;
   printf("\n Initial value of num1 = %d and
     num2 = %d", num1, num2);
   num1 += num2 * 4 - 7;
   printf("\n After evaluation of the
     expression num1 = %d and num2 = %d",
     num1, num2);
   return 0;
}
```

Output

```
Initial value of num1 = 3 and num2 = 5
After evaluation of the expression num1 = 16
   and num2 = 5
```

8.13.9 Comma Operator

The comma operator in C takes two operands. It works by evaluating the first and discarding its value, and then evaluates the second and returns the value as the result of the expression. Comma separated operands when chained together are evaluated in left-to-right sequence with the right-most value yielding the result of the expression. Among all the operators, the comma operator has the lowest precedence.

Therefore, when a comma operator is used, the entire expression evaluates to the value of the right expression. For example, the following statement assigns the value of b to x, then increments a, and then increments b.

```
int a = 2, b = 3, x = 0;
x = (++a, b+=a);
```

Now, the value of `x = 6`.

8.13.10 Sizeof Operator

The `sizeof` operator is a unary operator used to calculate the size of data types. This operator can be applied to all data types. When using this operator, the keyword `sizeof` is followed by a type name, variable, or expression. The operator returns the size of the variable, data type, or expression in bytes, i.e., the `sizeof` operator is used to determine the amount of memory space that the variable/expression/data type will take.

When a type name is used, it is enclosed in parentheses, but in case of variable names and expressions they can be specified with or without parentheses. A `sizeof` expression returns an unsigned value that specifies the space in bytes required by the data type, variable, or expression. For example, `sizeof(char)` returns 1, i.e., the size of a character data type. If we have,

```
int a = 10;
unsigned int result;
result = sizeof(a);
```

then `result = 2`, which is the space required to store the variable a in memory. Since a is an integer, it requires 2 bytes of storage space.

8.13.11 Operator Precedence Chart

C operators have two properties: *priority* and *associativity*. When an expression has more than one operator then it is the relative priorities of the operators with respect to each other that determine the order in which the expression will be evaluated. Associativity defines the direction in which the operator having the same precedence acts on the operands. It can be either left-to-right or right-to-left. Priority is given precedence over associativity to determine the order in which the expressions are evaluated. Associativity is then applied, if the need arises.

Table 8.17 lists the operators that C language supports in the order of their precedence (highest to lowest). The associativity indicates the order in which the operators of equal precedence in an expression are evaluated.

You must be wondering why the priority of the assignment operator is so low. This is because the action of assignment is performed only when the entire computation is done. It is not uncommon for a programmer to forget the priority of the operators while writing any program. So it is recommended that you use the parentheses operator to override default priorities. From table 8.17 you can see that the parenthesis operator has the highest priority. So any operator placed within the parenthesis will be evaluated before any other operator. The following

Table 8.16 Assignment Operators

Operator	Syntax	Equivalent to	Meaning	Example

/=	variable /= expression	variable = variable / expression	Divides the value of a variable by the value of an expression and assigns the result to the variable.	float a = 9.0;
				float b = 3.0;
				a /= b;

%=	variable %= expression	variable = variable % expression	Divides the value of a variable by the value of an expression and assigns the integer result to the variable.	int a= 9;
				int b = 3;
				a %= b;

*=	variable *= expression	variable = variable * expression	Multiplies the value of a variable by the value of an expression and assigns the result to the variable.	int a= 9;
				int b = 3;
				a *= b;

+=	variable += expression	variable = variable + expression	Adds the value of a variable to the value of an expression and assigns the result to the variable.	int a= 9;
				int b = 3;
				a += b;

–=	variable –= expression	variable = variable – expression	Subtracts the value of the expression from the value of the variable and assigns the result to the variable.	int a= 9;
				int b = 3;
				a –= b;

&=	variable &= expression	variable = variable & expression	Perform the bitwise AND with the value of variable and value of the expression and assign the result in the variable	int a = 10;
				int b = 20;
				a &= b;

^=	variable ^= expression	variable = variable ^ expression	Perform the bitwise XOR with the value of variable and value of the expression and assign the result in the variable	int a = 10;
				int b = 20;
				a ^= b;

<<=	variable <<= amount	variable = variable << amount	Performs an arithmetic left shift (amount times) on the value of a variable and assigns the result back to the variable.	int a= 9;
				int b = 3;
				a <<= b;

>>=	variable >>= amount	variable = variable >> amount	Performs an arithmetic right shift (amount times) on the value of a variable and assigns the result back to the variable.	int a= 9;
				int b = 3;
				a >>= b;

Table 8.17 Operator precedence

Rank	Operator	Associativity	Rank	Operator	Associativity
1	() [] . –> ++ --	left-to-right	9	^	left-to-right
2	++ -- + – ! ~ (type) * & sizeof	right-to-left	10	|	left-to-right
3	* / % ,	left-to-right	11	&&	left-to-right
4	+ –	left-to-right	12	||	left-to-right
5	<< >>	left-to-right	13	?:	right-to-left
6	< <= > >=	left-to-right	14	= += –= *= /= %= &= ^= |= <<= >>=	right-to-left
7	== !=	left-to-right	15	,	left-to-right
8	&	left-to-right	16		

examples illustrate the evaluation of expressions using the precedence chart:

1. `x = 3 * 4 + 5 * 6`
   `= 12 + 5 * 6`
   `= 12 + 30`
   `= 42`
2. `x = 3 * (4 + 5) * 6`
   `= 3 * 9 * 6`
   `= 27 * 6`
   `= 162`
3. `x = 3 * 4 % 5 / 2`
   `= 12 % 5 / 2`
   `= 2 / 2`
   `= 1`
4. `x = 3 * (4 % 5) / 2`
   `= 3 * 4 / 2`
   `= 12 / 2`
   `= 6`
5. `x = 3 * 4 % (5 / 2)`
   `= 3 * 4 % 2`
   `= 12 % 2`
   `= 0`
6. `x = 3 * ((4 % 5) / 2)`
   `= 3 * (4 / 2)`
   `= 3 * 2`
   `= 6`

Take the following variable declaration,

```
int a = 0, b = 1, c = -1;
float x = 2.5, y = 0.0;
```

If we write,

```
a = b = c = 7;
```

Since the assignment operator works from right-to-left, therefore `c = 7`. Then since `b = c`, `b = 7`. Now `a = b`, so `a = 7`.

7. `a += b -= c * 10`

This is expanded as

```
a = a + (b = b - (c = c * 10))
  = a + ( b = 1 - (-10)
  = a + ( b = 11)
  = 0 + 11
    = 11
```

8. `--a * ( 5 + b) / 2 - c++ * b`

```
= --a * 6 / 2 - c++ * b
= --a * 6 / 2 - -1 * b
```

(Value of c has been incremented but its altered value will not be visible for the evaluation of this expression)

```
= -1 * 6 / 2 - -1 * 1
```

(Value of a has been incremented and its altered value will be used for the evaluation of this expression)

```
= -1 * 3 - -1 * 1= -3 - -1 * 1 = -3 - -1 = -2
```

9. `a * b * cs`
   `= (a * b) * c` (because associativity of * is from left-to-right)
   `= 0`
10. `a && b = 0`
11. `a < b && c < b = 1`
12. `b + c || ! a = ( b + c) || (!a)`
    `= 1 ||1 = 1`
13. `x * 5 && 5 || ( b / c)`
    `= ((x * 5) && 5) || (b / c)`
    `= (1.25 && 5) || (1/-1) = 1`
14. `a <= 10 && x >= 1 && b`
    `= ((a <= 10) && (x >= 1)) && b`
    `= 1 && 1 && 1 = 1`
15. `!x || !c || b + c`
    `= ((!x) || (!c)) || (b + c)`
    `= 0 || 0 || 0 = 0`
16. `x * y < a + b || c`
    `= ((x * y) < (a + b)) || c`
    `= 0 < 1 || -1 = 1`
17. `(x > y) + !a || c++`
    `= ((x > y) + (!j)) || (c++)`
    `= 2.5 + 1 || 0 = 1`

8.14 PROGRAMMING EXAMPLES

15. Write a program to calculate the area of a circle.

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

```
int main()
{
   float radius;
   double area, circumference;
   clrscr();
   printf("\n Enter the radius of the circle: ");
   scanf("%f", &radius);
   area = 3.14 * radius * radius;
      circumference = 2 * 3.14 * radius;
   printf(" Area = %.2le", area);
      printf("\n CIRCUMFERENCE = %.2e",
   circumference);
   return 0;
}
```

Output

```
Enter the radius of the circle:12.4
Area = 153.86
CIRCUMFERENCE = 4.40e+01
```

16. Write a program to print the ASCII value of a character.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   char ch;
   clrscr();
   printf("\n Enter any character: ");
   scanf("%c", &ch);
   printf("\n The ascii value of %c is: %d",
     ch, ch);
   return 0;
}
```

Output

```
Enter any character: A
The ascii value of A is: 65
```

17. Write a program to read a character in upper case and then print it in lower case.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   char ch;
   clrscr();
   printf("\n Enter any character in upper
     case: ");
   scanf("%c", &ch);
   printf("\n The character in lower case
     is: %c", ch+32);
   return 0;
}
```

Output

```
Enter any character in upper case: A
The character in lower case is: a
```

18. Write a program to print the digit at ones place of a number.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int num, digit_at_ones_place;
   clrscr();
   printf("\n Enter any number: ");
   scanf("%d", &num);
   digit_at_ones_place = num % 10;
   printf("\n The digit at ones place of %d
     is %d", num, digit_at_ones_place);
   return 0;
}
```

Output

```
Enter any number: 123
The digit at ones place of 123 is 3
```

19. Write a program to swap two numbers using a temporary variable.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int num1, num2, temp;
   clrscr();
   printf("\n Enter the first number: ");
   scanf("%d",&num1);
   printf("\n Enter the second number: ");
   scanf("%d",&num2);
   temp = num1;
   num1 = num2;
   num2 = temp;
   printf("\n The first number is %d", num1);
   printf("\n The second number is %d", num2);
   return 0;
}
```

Output

```
Enter the first number : 3
Enter the second number : 5
The first number is 5
The second number is 3
```

20. Write a program to swap two numbers without using a temporary variable.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int num1, num2;
   clrscr();
   printf("\n Enter the first number: ");
```

```
    scanf("%d",&num1);

    printf("\n Enter the second number: ");
    scanf("%d",&num2);

    num1 = num1 + num2;
    num2= num1 - num2;
    num1 = num1 - num2;

    printf("\n The first number is %d", num1);
    printf("\n The second number is %d", num2);
    return 0;
}
```

Output

```
Enter the first number: 3
Enter the second number: 5
The first number is 5
The second number is 3
```

21. Write a program to calculate the average of two numbers. Also print their deviation.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int num1, num2;
   float avg, dev1, dev2;
   printf("\n Enter the two numbers:
   scanf("%d %d", &num1, &num2);
   avg = (num1 + num2) / 2;
   dev1 = num1 - avg;
   dev2 = num2 - avg;
   printf("\n AVERAGE = %.2f", avg);
   printf("\n Deviation of first number =
     %.2f", dev1);
   printf("\n Deviation of second number =
     %.2f", dev2);
   return 0;
}
```

Output

```
Enter the two numbers: 45 32
AVERAGE = 38.00
Deviation of first number = 7.00
Deviation of first number = -6.00
```

22. Write a program to convert degrees Fahrenheit into degrees celsius.

```
#include <stdio.h>
#include <conio.h>
main()
{
   float fahrenheit, celsius;
   printf("\n Enter the temperature in
     fahrenheit: ");
   scanf("%f", &fahrenheit);
   celsius = (0.56) * (fahrenheit - 32);
   printf("\n Temperature in degrees celsius
     = %f", celsius);
   return 0;
}
```

Output

```
Enter the temperature in fahrenheit: 108
Temperature in degrees celsius: 42.56
```

23. Write a program that displays the size of every data type.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   clrscr();
   printf("\n The size of short integer is:
     %d", sizeof(short int));
   printf("\n The size of unsigned integer
     is: %d", sizeof(unsigned int));
   printf("\n The size of signed integer is:
     %d", sizeof(signed int));
   printf("\n The size of integer is: %d",
     sizeof(int));
   printf("\n The size of long integer is:
     %d", sizeof(long int));
   printf("\n The size of character is: %d",
     sizeof(char));
   printf("\n The size of unsigned character
     is: %d", sizeof(unsigned char));
   printf("\n The size of signed character
     is: %d", sizeof(signed char));
   printf("\n The size of floating point
     number is: %d", sizeof(float));
   printf("\n The size of double number is:
     %d", sizeof(double));
   return 0;
}
```

Output

```
The size of short integer is: 2
The size of unsigned integer is: 2
The size of signed integer is: 2
The size of integer is: 2
The size of long integer is: 2
The size of character is: 1
The size of unsigned character is: 1
The size of signed character is: 1
The size of floating point number is: 4
The size of double number is: 8
```

24. Write a program to calculate the total amount of money in the piggybank, given the coins of Rs 10, Rs 5, Rs 2, and Re 1.

```
#include <stdio.h>
#include <conio.h>
main()
{
   int num_of_10_coins, num_of_5_coins, num_
     of_2_coins, num_of_1_coins;
   float total_amt = 0.0;
   clrscr();
   printf("\n Enter the number of Rs10 coins
     in the piggybank: ");
   scanf("%d", &num_of_10_coins);
   printf("\n Enter the number of Rs5 coins
     in the piggybank: ");
   scanf("%d", &num_of_5_coins);
   printf("\n Enter the number of Rs2 coins
     in the piggybank: ");
   scanf("%d", &num_of_2_coins);
   printf("\n Enter the number of Re1 coins
     in the piggybank: ");
   scanf("%d", &num_of_1_coins);

   total_amt = num_of_10_coins * 10 + num_
     of_5_coins * 5 + num_of_2_coins * 2 +
     num_of_1_coins;

   printf("\n Total amount in the piggybank
     = %f", total_amt);
   getch();
   return 0;
}
```

Output

```
Enter the number of Rs10 coins in the
  piggybank: 10
Enter the number of Rs5 coins in the
  piggybank: 23
Enter the number of Rs2 coins in the
  piggybank: 43
Enter the number of Re1 coins in the
  piggybank: 6
Total amount in the piggybank = 307
```

25. Write a program to calculate the bill amount for an item given its quantity sold, value, discount, and tax.

```
#include <stdio.h>
#include <conio.h>
main()
{
   float total_amt, amt, sub_total,
     discount_amt, tax_amt, qty, val,
     discount, tax;
   printf("\n Enter the quantity of item
     sold: ");
   scanf("%f", &qty);
   printf("\n Enter the value of item: ");
   scanf("%f", &val);
   printf("\n Enter the discount percentage: ");
   scanf("%f", &discount);
   printf("\n Enter the tax: ");
   scanf("%f", &tax);

   amt = qty * val;
   discount_amt = (amt * discount)/100.0;
   sub_total = amt - discount_amt;
   tax_amt = (sub_total * tax) /100.0;
   total_amt = sub_total + tax_amt;

   printf("\n\n\n ****** BILL ******");
   printf("\n Quantity sold: %f", qty);
   printf("\n Price per item: %f", val);
   printf("\n\t\t -------------");
   printf("\n Amount: %f", amt);
   printf("\n Discount: - %f", discount_amt);
   printf("\n Discounted Total: %f", sub_
     total);
   printf("\n Tax: + %f", tax_amt);
   printf("\n \t\t-------------");
   printf("\n Total Amount %f", total_amt);
   return 0;
}
```

Output

```
Enter the quantity of item sold: 20
Enter the value of item: 300
Enter the discount percentage: 10
Enter the tax: 12
************ BILL **************"
Quantity sold :              20
Price per item :             300
                    ---------------
Amount :                    6000
Discount :               -   600
Discounted Total :          5400
Tax :                    +   648
                    -------------
Total Amount                6048
```

8.15 TYPE CONVERSION AND TYPECASTING

Till now we have assumed that all the expressions involved data of the same type; but what happens when expressions involve two different data types, such as multiplying a floating point number and an integer? Such type of situations are handled either through type conversion or typecasting.

Type conversion or typecasting of variables refers to changing a variable of one data type into another. Type conversion is done implicitly whereas, typecasting has

to be done explicitly by the programmer. We will discuss both of them in this section.

8.15.1 Type Conversion

Type conversion is done when the expression has variables of different data types. To evaluate the expression, the data type is promoted from a lower to a higher level where the hierarchy of data types (from higher to lower) can be given as: double, float, long, int, short, and char. Figure 8.12 shows the conversion hierarchy of data types.

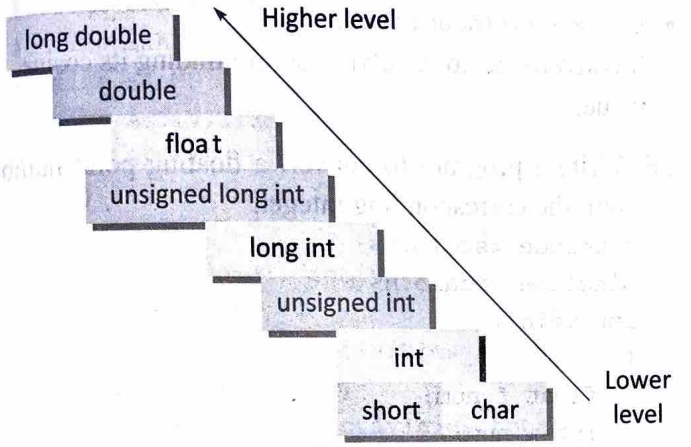


Figure 8.12 Conversion hierarchy of data types

Type conversion is automatically done when we assign an integer value to a floating point variable. Consider the following code in which an integer data type is promoted to float. This is known as *promotion* (when a lower level data type is promoted to a higher type).

```
float x;
int y = 3;
x = y;
```

Now, `x = 3.0`, as the integer value is automatically converted into its equivalent floating point representation. In some cases, when an integer is converted into a floating point number, the resulting floating point number may not exactly match the integer value. This is because the floating point number format used internally by the computer cannot accurately represent every possible integer number. So even if the value of `x = 2.99999995`, you must not worry. The loss of accuracy because of this feature would always be insignificant for the final result. Let us summarize how promotion is done:

- `float` operands are converted to `double`.
- `char` or `short` operands whether signed or unsigned are converted to `int`.
- If any one operand is `double`, the other operand is also converted to `double`. Hence, the result is also of type `double`.
- If any one operand is `long`, the other operand is also converted to `long`. Hence, the result is also of type `long`.

Figure 8.13 exhibits type conversions in an expression.

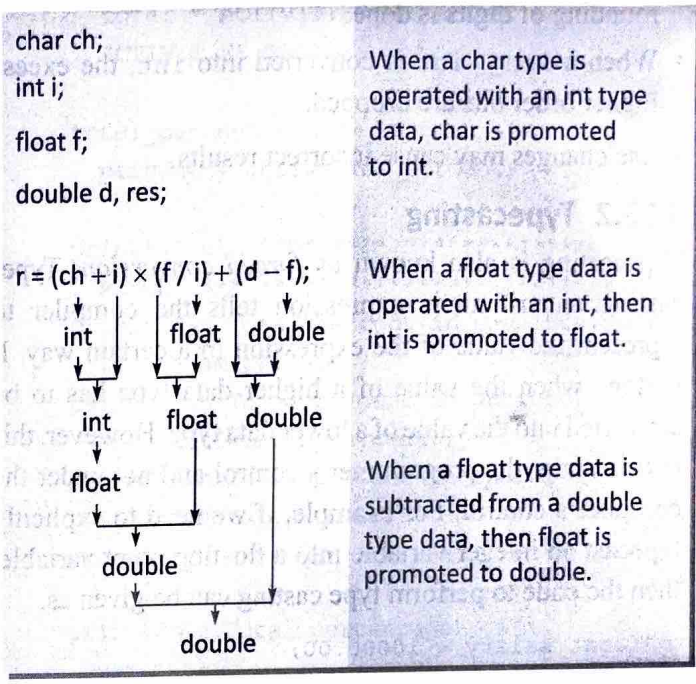


Figure 8.13 Type conversion

Consider the following group of statements.

```
float f = 3.5;
int i;
i = f;
```

The statement `i = f` results in `f` to be demoted to type `int`, i.e., the fractional part of `f` will be lost and `i` will contain 3 (not 3.5). In this case demotion takes place, i.e., a higher level data type is converted into a lower type. Whenever demotion occurs, some information is lost. For example, in this case the fractional part of the floating point number is lost.

Similarly, if we convert an `int` to a `short int` or a `long int` to `int`, or `int` to `char`, the compiler just drops the extra bits (Figure 8.14).

Note No compile time warning message is generated when information is lost while demoting the type of data.

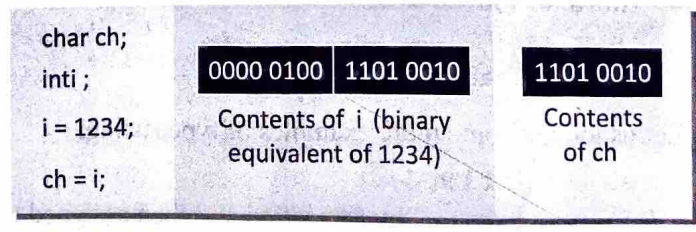


Figure 8.14 Implicit conversion example

Thus we can observe the following changes that are unavoidable when performing type conversions.

- When a `float` value is converted to an integer value, the fractional part is truncated.
- When a `double` value is converted to a `float` value, rounding of digits is done.
- When a `long int` is converted into `int`, the excess higher order bits are dropped.

These changes may cause incorrect results.

8.15.2 Typecasting

Typecasting is also known as *forced conversion*. Typecasting an arithmetic expression tells the compiler to represent the value of the expression in a certain way. It is done when the value of a higher data type has to be converted into the value of a lower data type. However, this cast is under the programmer's control and not under the compiler's control. For example, if we need to explicitly typecast an integer variable into a floating point variable, then the code to perform type casting can be given as,

```
float salary = 10000.00;
int sal;
sal = (int) salary;
```

When floating point numbers are converted to integers (as in type conversion), the digits after the decimal are truncated. Therefore, data is lost when floating-point representations are converted to integral representations. So in order to avoid such type of inaccuracies, int type variables must be typecast to float type.

As we see in the code, typecasting can be done by placing the destination data type in parentheses followed by the variable name that has to be converted. Hence, we conclude that typecasting is done to make a variable of one data type to act like a variable of another type.

We can also typecast integer values to its character equivalent (as per ASCII code) and vice versa. Typecasting is also done in arithmetic operations to get correct result. For example, when dividing two integers, the result can be of floating type. Also when multiplying two integers the result can be of long int. So to get correct precision value, typecasting can be done. For instance,

```
int a = 500, b = 70 ;
float res;
res = (float) a / b;
```

Let us look at some more examples of typecasting.

- `res = (int)9.5;`
  9.5 is converted to 9 by truncation and then assigned to `res`.
- `res = (int)12.3 / (int)4.2;`
  It is evaluated as 12/4 and the value 3 is assigned to `res`.
- `res = (double)total / n;`
  `total` is converted to double and then division is done in floating point mode.
- `res = (int)(a + b);`
  The value of `a + b` is converted to integer and then assigned to `res`.
- `res = (int)a + b;`
  `a` is converted to int and then added with `b`.
- `res = cos((double)x);`
  It converts `x` to `double` before finding its cosine value.

26. Write a program to convert a floating point number into the corresponding integer.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   float f_num;
   int i_num;
   clrscr();
   printf("\n Enter any floating point
     number: ");
   scanf("%f", &f_num);
   i_num = (int)f_num;
   printf("\n The integer variant of %f is
     = %d", f_num, i_num);
   return 0;
}
```

Output

```
Enter any floating point number: 23.45
The integer variant of 23.45 is = 23
```

27. Write a program to convert an integer into the corresponding floating point number.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   float f_num;
   int i_num;
   clrscr();
   printf("\n Enter any integer: ");
   scanf("%d", &i_num);
   f_num = (float)i_num;
   printf("\n The floating point variant of
     %d is = %f", i_num, f_num);
   return 0;
}
```

Output

```
Enter any integer: 12
The floating point variant of 12 is =
  12.00000
```

28. Write a program to calculate a student's result based on two examinations, one sports event, and three activities conducted. The weightage of activities = 30%, sports = 20%, and examination = 50%.

```
#include <stdio.h>
#include <conio.h>
#define ACTIVITIES_WEIGHTAGE 30
#define SPORTS_WEIGHTAGE 20
#define EXAMS_WEIGHTAGE 50
#define EXAMS_TOTAL 200
#define ACTIVITIES_TOTAL 60
#define SPORTS_TOTAL 50
main()
{
  int exam_score1, activities_score1,
    sports_score;
  int exam_score2, activities_score2,
    activities_score3;
  float exam_total, activities_total;
  float total_percent, exam_percent,
    sports_percent, activities_percent;
  clrscr();
  printf("\n Enter the score obtained in
    two examination (out of 100): ");
  scanf("%d %d", &exam_score1, &exam_score2);
  printf("\n Enter the score obtained in
    sports events (out of 50): ");
  scanf("%d", &sports_score);
  printf("\n Enter the score obtained in
    three activities (out of 20): ");
  scanf("%d %d %d", &activities_score1,
    &activities_score2, &activities_score3);

  exam_total = exam_score1 + exam_score2;
  activities_total = (activities_score1 +
    activities_score2 + activities_score3);
  exam_percent = (float)exam_total * EXAMS_
    WEIGHTAGE / EXAMS_TOTAL;
  sports_percent = (float)sports_score *
    SPORTS_WEIGHTAGE / SPORTS_TOTAL;
  activities_percent = (float)activities_
    total * ACTIVITIES_WEIGHTAGE /
    ACTIVITIES_TOTAL;

  total_percent = exam_percent + sports_
    percent + activities_percent;

  printf("\n\n **********************
    RESULT ***************************");
  printf("\n Total percent in examintaion :
    %f", exam_percent);
  printf("\n Total percent in activities :
    %f", activities_percent);
  printf("\n Total percent in sports :
    %f", sports_percent);
  printf("\n ------------------------------
    ----------------------------");
  printf("\n Total percentage : %f", total_
    percent);
  return 0;
}
```

Output

```
Enter the score obtained in two examination
   (out of 100): 78 89
Enter the score obtained in sports events
   (out of 50): 34
Enter the score obtained in three activities
   (out of 20): 19 18 17
****************** RESULT ******************
Total percent in examintaion: 41.75
Total percent in activities : 27
Total percent in sports     : 13
-----------------------------------------
Total percentage            : 82%
```

SUMMARY

- C was developed in the early 1970s by Dennis Ritchie at Bell Laboratories.
- A function is defined as a group of C statements that are executed together. The execution of a C program begins at the main function.
- Every word in a C program is either a keyword or an identifier. C has a set of reserved words often known as keywords that cannot be used as an identifier.
- A variable is defined as a meaningful name given to the data storage location in computer memory. When using a variable, we actually refer to address of the memory where the data is stored.
- The difference between signed and unsigned numeric variables is that signed variables can be either negative or positive but unsigned variables can only be positive. By default, C takes a signed variable.
- The statement 'return 0;' returns the value 0 to the operating system to give an indication that there were no errors encountered during the execution of the program.

SUMMARY

- The conditional or the ternary operator (?:) is just like an if–else statement that can be within expressions. Conditional operator is also known as ternary operator as it is neither a unary nor a binary operator; it takes three operands.
- The bitwise NOT, or complement, produces the 1s complement of the given binary value.
- The comma operator evaluates the first expression and discards its value, and then evaluates the second and returns the value as the result of the expression.
- sizeof is a unary operator used to calculate the size of data types. This operator can be applied to all data types.
- While type conversion is done implicitly, typecasting has to be done explicitly by the programmer. Typecasting is done when the value of a higher data type has to be converted to a lower data type.

GLOSSARY

ANSI C American National Standards Institute's definition of the C programming language. It is the same as the ISO definition.

Constant A value that cannot be changed.

Data type Definition of the data. For example, int, char, float.

Escape sequence Control codes that comprises of combinations of a backslash followed by letters or digits which represent non printing characters.

Expression A sequence of operators and operands that may yield a single value as the result of its computation.

Executable program Program which will run in the environment of the operating system or within an appropriate run time environment.

Floating point number Number that comprises of a decimal place and exponent.

Format specification A string which controls the manner in which input or output of values has to be done.

Identifier The names used to refer to stored data values as in case of constants, variables or functions.

Integer A number that has no fractional part.

Keyword A word which has a predefined meaning to a C compiler and therefore must not be used for any other purpose.

Library file The file which comprises of compiled versions of commonly used functions that can be linked to an object file to make an executable program.

Library function A function whose source code is stored in the external library file.

Linker The tool that connects object code and libraries to form a complete, executable program.

Operator precedence The order in which operators are applied to operands during the evaluation of an expression.

Preprocessor A processor that manipulates the initial directives of the source file. The source file contains instructions that specifies how the source file shall be processed and compiled.

Preprocessor directive Instructions in the source file that specifies how the file shall be processed and compiled.

Program A text file that contains the source code to be compiled.

Runtime error A program that is encountered when a program is executed.

Source code A text file that contains the source code to be compiled.

Statement A simple statement in C language that is followed by a semicolon.

Syntax error An error or mistake in the source code that prevents the compiler from converting it into object code.

Variable An identifier (and storage) for a data type. The value of a variable may change as the program runs.

EXERCISES

Fill in the Blanks

1. C was developed by D. Ritchie
2. Fu is a group of C statements that are executed together.
3. Execution of the C program begins at main()
4. In memory characters are stored as ASCII Codes
5. The statement return 0, returns 0 to the OS.
6. Modulus operator finds the remainder of an integer division.
7. Logical NOT operator reverses the value of the expression.
8. sizeof is a unary operator used to calculate the size of data types.