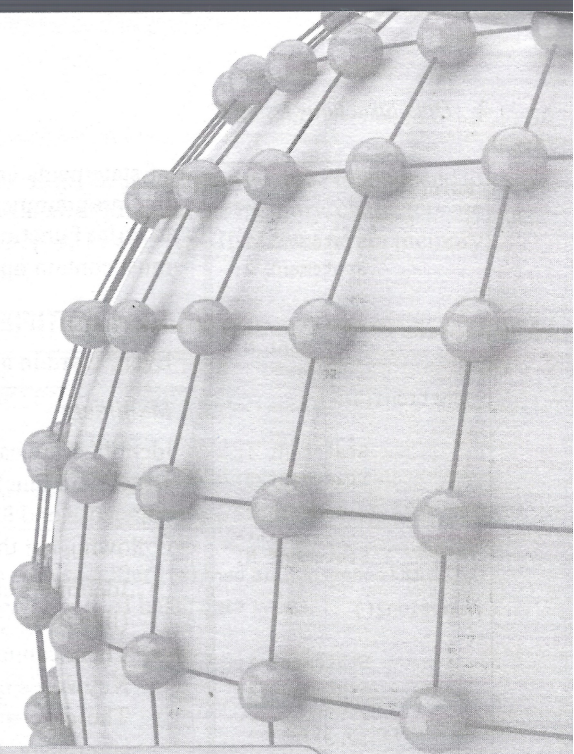
Method 467
Method 468
od 468

olution by Open
469
olution By
1
shing 485
ing 485
ons of Hashing 486

491
492
3
nization 493
ganization 494
tial File
495
496
e Indices 497
Indexing 497
es 498
499
500
501

CHAPTER 1

Introduction to C



LEARNING OBJECTIVE

This book deals with the study of data structures through C. Before going into a detailed analysis of data structures, it would be useful to familiarize ourselves with the basic knowledge of programming in C. Therefore, in this chapter we will learn about the various constructs of C such as identifiers and keywords, data types, constants, variables, input and output functions, operators, control statements, functions, and pointers.

1.1 INTRODUCTION

The programming language 'C' was developed in the early 1970s by Dennis Ritchie at Bell Laboratories. Although C was initially developed for writing system software, today it has become such a popular language that a variety of software programs are written using this language. The greatest advantage of using C for programming is that it can be easily used on different types of computers. Many other programming languages such as C++ and Java are also based on C which means that you will be able to learn them easily in the future. Today, C is widely used with the UNIX operating system.

Structure of a C Program

A C program contains one or more functions, where a function is defined as a group of statements that perform a well-defined task. Figure 1.1 shows the structure of a C program. The statements in a function are written in a logical sequence to perform a specific task. The `main()` function is the most important function and is a part of every C program. Rather, the execution of a C program begins with this function.

From the structure given in Fig. 1.1, 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

```
main()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
Function1()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
Function2()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
.................
.................
FunctionN()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
```

Figure 1.1 Structure of a C program

of statements arranged according to specific meaningful sequence. Note that programmers can choose any name for functions. It is not mandatory to write Function1, Function2, etc., with an exception that every program must contain one function that has its name as `main()`.

1.2 IDENTIFIERS AND KEYWORDS

Every word in a C program is either an identifier or a keyword.

Identifiers

Identifiers are basically names given to program elements such as variables, arrays, and functions. They are formed by using a sequence of letters (both uppercase and lowercase), numerals, and underscores.
Following are the rules for forming identifier names:

- Identifiers cannot include any special characters or punctuation marks (like #, $, ^, ?, ., 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'.
- Identifiers must begin with a letter or an underscore. However, use of underscore as the first character must be avoided because several complier-defined identifiers in the standard C library have underscore as their first character. So, inadvertently duplicated names may cause definition conflicts.
- Identifiers can be of any reasonable length. They should not contain more than 31 characters. (They can actually be longer than 31, but the compiler looks at only the first 31 characters of the name.)

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 lower case letters. Table 1.1 contains the list of keywords in C.

Table 1.1 Keywords in C language

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

1.3 BASIC DATA TYPES

Data type determines the set of values that a data item can take and the operations that can be performed on the item. C language provides four basic data types. Table 1.2 lists the data types, their size, range, and usage for a C programmer.

The `char` data type is of one byte 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

might have been 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 the memory, characters are stored in their ASCII codes. For example, the character 'A' has the ASCII code of 65. In memory we will not store 'A' but 65 (in binary number format).

Table 1.2 Basic data types in C

Data Type	Size in Bytes	Range	Use
char	1	-128 to 127	To store characters
int	2	-32768 to 32767	To store integer numbers
float	4	3.4E-38 to 3.4E+38	To store floating point numbers
double	8	1.7E-308 to 1.7E+308	To store big floating point numbers

In addition, C also supports four modifiers—two sign specifiers (`signed` and `unsigned`) and two size specifiers (`short` and `long`). Table 1.3 shows the variants of basic data types.

Table 1.3 Basic data types and their variants

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 int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
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

Note When the basic data type is omitted from a declaration, then automatically type `int` is assumed. For example,

```
long var;    //int is implied
```

While the smaller data types take less memory, the larger data 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 need to use any other data type.

1.4 VARIABLES AND CONSTANTS

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

Numeric Variables

Numeric variables can be used to store either integer values or floating point values. Modifiers like `short`, `long`, `signed`, and `unsigned` can also be used with numeric variables. 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 omit the `signed/unsigned` modifier, C language automatically makes it a signed variable. To declare an unsigned variable, the `unsigned` modifier must be explicitly added during the declaration of the variable.

Character Variables

Character variables are just single characters enclosed within single quotes. These characters could be any character from the ASCII character set—letters ('a', 'A'), numerals ('2'), or special characters ('&').

Declaring Variables

To declare a variable, specify the data type of the variable followed by its name. The data type indicates the kind of values that the variable can store. Variable names should always be meaningful and must reflect the purpose of their usage in the program. In C, variable declaration always ends with a semi-colon. 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 so that the source code is easier to maintain.

Initializing Variables

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

```
int emp_num = 7;
float salary = 9800.99
char grade = 'A';
double balance_amount = 100000000;
```

Constants

Constants are identifiers whose values do not change. While values of variables can be changed at any time, values of constants can never be changed. Constants are used to define fixed values like *pi* or the charge on an electron so that their value does not get changed in the program even by mistake.

Declaring Constants

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

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

1.5 WRITING THE FIRST C PROGRAM

To write a C program, we first need to write the code. For that, 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 `emac` 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 ");// prints the message on the screen
        return 0;// returns a value 0 to the operating system
}
```

After writing the code, select the directory of your choice and save the file as `first.c`.

`#include <stdio.h>` This is the first statement in our code that includes a file called `stdio.h`. This file has some in-built functions. By simply including this file in our code, we can use these functions directly. `stdio` basically stands for Standard Input/Output, which means it has functions for input and output of data like reading values from the keyboard and printing the results on the screen.

`int main()` Every C program `contains a main()` function which is the starting point of the program. `int` is the return value of the `main` function. After all the statements in the program have been executed, 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 this chapter in toto. So even if you do not understand certain things, do not worry.

`{ }` The two curly brackets are used to group all the related statements of the `main` function.

`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.

`\n` is an escape sequence and represents a newline character. It is used to print the message on a new line on the screen. Other escape sequences supported by C language are shown in Table 1.4.

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

Table 1.4 Escape sequences

Escape Sequence	Purpose
\a	Audible signal
\b	Backspace
\t	Tab
\n	New line
\v	Vertical tab
\f	New page\Clear screen
\r	Carriage return

Note Every statement in the main function ends with a semi-colon (;).

`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 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 what mistake(s) you have made and on which line the error has occurred. In case of errors, you need to re-open your `.c` file and correct the 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

`"first.exe"` for Windows and `"./first"` 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.

Using Comments

Comments are a way of explaining what a program does. C supports two types of comments.

- // is used to comment a single statement.
- /* is used to comment multiple statements. A /* is ended with */ and all statements that lie between these characters are commented.

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

Standard Header Files

Till now we have used `printf()` function, which is defined in the `stdio.h` header file. Even in other programs that we will be writing, we will use many functions that are not written by us. For example, to use the `strcmp()` function that compares two strings, we will 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:

- `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 these files.

1.6 INPUT AND OUTPUT FUNCTIONS

The most fundamental operation in a C program is to accept *input* values from a standard input device and *output* the data produced by the program to a standard output device. As shown in Section 1.4, we can assign values to variables using the assignment operator '='. For example,

```
int a = 3;
```

What if we want to assign value to variable `a` that is inputted from the user at run-time? This is done by using the `scanf` function that reads data from the keyboard. Similarly, for outputting results of

the program, `printf` function is used that sends results to a terminal. Like `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 Library. A program that uses standard input/output functions must contain the following statement at the beginning of the program:

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

scanf()

The `scanf()` function is used to read formatted data from the keyboard. The syntax of the `scanf()` function can be given as,

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

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`. The prototype of the control string can be given as,

```
%[*][width][modifier]type
```

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

width is an optional argument that specifies the maximum number of characters to be read. However, if the `scanf` function encounters a white space or an unconvertible character, input is terminated.

modifier is an optional argument (**h, l,** or **L**) , which modifies the type specifier. 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 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 1.5.

Table 1.5 Type specifiers

Type	Qualifying Input
%c	For single characters
%d, %i	For integer values
%e,%E,%f,%g,%G	For floating point numbers
%o	For octal numbers
%s	For a sequence of (string of) characters
%u	For unsigned integer values
%x,%X	For hexadecimal values

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.

As we have not studied functions till now, understanding `scanf` function in depth will 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. Look at the following code that shows how we can input value in a variable of `int` data type:

```
int num;
scanf(" %4d ", &num);
```

The `scanf` function reads first four digits into the address or the memory location pointed by `num`.

Note In case of reading strings, we do not use the & sign in the `scanf` function.

printf()

The `printf` function is used to display information required by the user and also prints the values of the variables. Its syntax can be given as:

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

After the control string, the function can have as many arguments as specified in the control string. The control string contains format specifiers which are arranged in the order so that they correspond with the arguments in the variable list. It may also contain text to be printed such as instructions to the user, identifier names, or any other text to make the text readable.

Note that there must be enough arguments because if there are not enough arguments, then the result will be 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 given as below:

```
%[flags][width][.precision][modifier]type
```

Each control string must begin with a % sign.

flags is an optional argument, which specifies output justification like decimal point, numerical sign, trailing zeros or octadecimal or hexadecimal prefixes. Table 1.6 shows different types of flags with their descriptions.

Table 1.6 Flags in printf()

Flags	Description
-	Left-justify within the 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 zeroes (0) instead of spaces

width is an optional argument which specifies the minimum number of positions that the output characters will occupy. If the number of output characters is smaller than the specified width, then the output would be right justified with blank spaces to the left. However, if the number of characters is greater than the specified width, then all the characters would be printed.
precision is an optional argument which specifies the number of digits to print after the decimal point or the number of characters to print from a string.
modifier field is same as given for `scanf()` function.
type is used to define the type and the interpretation of the value of the corresponding argument. The type specifiers for `printf` function are given in Table 1.5.

The most simple `printf` statement is

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

The function when executed prompts the message enclosed in the quotation to be displayed on the screen.

For `float x = 8900.768`, the following examples show output under different format specifications:

```
printf ("%f", x);
```

8	9	0	0	.	7	6	8

```
printf("%10f", x);
```

		8	9	0	0	.	7	6	8

```
printf("%9.2f", x);
```

		8	9	0	0	.	7	7

```
printf("%6f", x);
```

8	9	0	0	.	7	6	8

1.7 OPERATORS AND EXPRESSIONS

C language supports different types of operators, which can be used with variables and constants to form expressions. These operators can be categorized into the following major groups:

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

We will now discuss all these operators.

Arithmetic Operators

Consider three variables declared as,

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

We will use these variables to explain arithmetic operators. Table 1.7 shows the arithmetic operators, their syntax, and usage in C language.

Table 1.7 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 1.7, `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, multiplication, and division (`+`, `-`, `*`, and `/`) operators are the usual arithmetic operators, so you are already familiar with these operators.

However, the operator % might be new to you. The modulus operator (%) finds the remainder of an integer division. This operator can be applied only on integer operands and cannot be used on float or double operands.

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
```

Note 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 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 be a floating point number.

All the arithmetic operators bind from left to right. Multiplication, division, and modulus operators have higher precedence over addition and subtraction operators. Thus, 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 can be performed. For example,

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

Relational Operators

A relational operator, also known as a comparison operator, is an operator that compares two values or 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 if x is less than y, relational operator < is used as x < y. This expression will return true value if x is less than y; otherwise the value of the expression will be false. C provides four relational operators which are illustrated in Table 1.8. These operators are evaluated from left to right.

Table 1.8 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

Equality Operators

C language also supports two equality operators to compare 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 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 1.9 summarizes equality operators.

Table 1.9 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

Logical Operators

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

Table 1.10 Truth table of logical AND

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

Logical AND (&&)

Logical AND is a binary operator, which simultaneously evaluates two values or relational expressions. If both the operands are true, then the whole expression is true. If both or one of the operands is false, then the whole expression evaluates to false. The truth table of logical AND operator is given in Table 1.10.

For example,

```
(a < b) && (b > c)
```

The whole expression is true only if both expressions are true, i.e., if `b` is greater than `a` and `c`.

Table 1.11 Truth table of logical OR

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

Logical OR (||)

Logical OR returns a false value if both the operands are false. Otherwise it returns a true value. The truth table of logical OR operator is given in Table 1.11. For example,

```
(a < b) || (b > c)
```

The whole expression is true if either `b` is greater than `a` or `b` is greater than `c` or `b` is greater than both `a` and `c`.

Table 1.12 Truth table of logical NOT

A	! A
0	1
1	0

Logical NOT (!)

The logical NOT operator takes a single expression and produces a zero if the expression evaluates to a non-zero value and produces a 1 if the expression produces a zero. The truth table of logical NOT operator is given in Table 1.12. 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.

Unary Operators

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

Unary Minus (–)

Unary minus operator negates the value of its operand. 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 variable `b` has a positive value. After applying unary minus operator (–) on the operand `b`, the value becomes –10, which indicates it has a negative value.

Increment Operator (++) and Decrement Operator (--)

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 the operand while in a postfix expression (`x++` or `x--`), the operator is applied after the operand.

An important point to note about unary increment and decrement operators is that `++x` is not same as `x++`. Similarly, `--x` is not the same as `x--`. Although, `x++` and `++x` both 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. Note that unary operators have a higher precedence than the binary operators. And if in an expression we have more than one unary operator then they are evaluated from right to left.

Conditional Operator

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 larger 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.

Conditional operators make the program code more compact, more readable, and safer to use as it is easier to both check and guarantee the arguments that are used for evaluation. Conditional operator is also known as ternary operator as it takes three operands.

Bitwise Operators

As the name suggests, bitwise operators perform operations at the bit level. These operators include: bitwise AND, bitwise OR, bitwise XOR, and shift operators.

Bitwise AND

Like boolean AND (&&), bitwise AND operator (&) 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 the same as we had seen in logical AND operation. 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 and 0 otherwise. For example,

```
10101010 & 01010101 = 00000000
```

Bitwise OR

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 the same as we had seen in logical OR operation. 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 and 0 otherwise. For example,

```
10101010 | 01010101 = 11111111
```

Bitwise XOR

When we use the bitwise XOR operator, the bit in the first operand is XORed with the corresponding

Table 1.13 Truth table of bitwise XOR

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

bit in the second operand. The truth table of bitwise XOR operator is shown in Table 1.13. 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 and 0 otherwise. For example,

```
10101010 ^ 01010101 = 11111111
```

Bitwise NOT (~)

The bitwise NOT or complement is a unary operator that performs logical negation on each bit of the operand. By performing negation of each bit, it actually produces the one's 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
```

Shift Operators

C supports two bitwise shift operators. They are shift left (<<) and shift right (>>). The syntax for a shift operation can be given as

```
operand op num
```

where the bits in the operand are shifted left or right depending on the operator (left, if the operator is << and right, if the operator is >>) by 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 (least significant bit) of `x` is set to 0. Therefore, if we have `x = 0001 1101`, then

```
x << 3 gives result = 1110 1000
```

On the contrary, when we apply a right shift, every bit in `x` is shifted to the right by one place. So, the LSB of `x` is lost, the MSB of `x` is set to 0. For example, if we have `x = 0001 1101`, then

```
x >> 1 gives result = 0000 1110
```

Similarly, if we have `x = 0001 1101`, then

```
x >> 4 gives result = 0000 0001
```

Note The expression `x << y` is equivalent to multiplication of x by 2^y. And the expression `x >> y` is equivalent to division of x by 2^y if x is unsigned or has a non-negative value.

Assignment Operators

In C language, the assignment operator is responsible for assigning values to the variables. While the equal sign (=) is the fundamental assignment operator, C also supports other assignment operators that provide shorthand 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`. The assignment operator has right-to-left associativity, so the expression