UNIT 3: BASICS OF C

UNIT STRUCTURE

3.0 OBJECTIVE

After going through this unit, you will be able to:

Understand the structure of a C program
Write, Compile and Execute a C Program
Define identifiers, data types and keywords in C
Describe memory requirements for different types of variables
Write and evaluate various arithmetical and logical expressions
Write interactive programs using input/output functions

3.1 INTRODUCTION

Computer programs usually work with different types of data and need a way to store the values being used. These values can be numbers or characters. C language has two ways of storing number values—variables and constants—with many options for each. Constants and variables are the fundamental elements of each program. Simply speaking, a program is nothing else than defining them and manipulating them. A variable is a data storage location that has a value that can change

during program execution. In contrast. a constant has a fixed value that can't change.
The next step is to use those variables in expressions. For writing an expression we need operators along with variables. An operator performs an operation (evaluation) on one or more operands. An operand is a sub expression on which an operator acts.

C itself provides no input/output facilities; there are no READ or WRITE statements, and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly called functions. Most C implementations have included a reasonably standard collection of such functions.

This unit is concerned with the basic elements used to construct simple C program statements. These elements include the C character set, identifiers and keywords, data types, constants, variables and arrays, declaration and naming conventions of variables. It also focuses on different types of operators available in C including the syntax and use of each operator and how they are used in C. Lastly, various ways for input/output operations are discussed.

3.2 HISTORY OF 'C' LANGUAGE

The C programming language was devised in the early 1970s by Dennis M. Ritchie an employee from Bell Labs (AT&T).

In the 1960s Ritchie worked, with several other employees of Bell Labs (AT&T), on a project called Multics. The goal of the project was to develop an operating system for a large computer that could be used by a thousand users. In 1969 AT&T (Bell Labs) withdrew from the project, because the project could not produce an economically useful system. So the employees of Bell Labs (AT&T) had to search for another project to work on (mainly Dennis M. Ritchie and Ken Thompson).

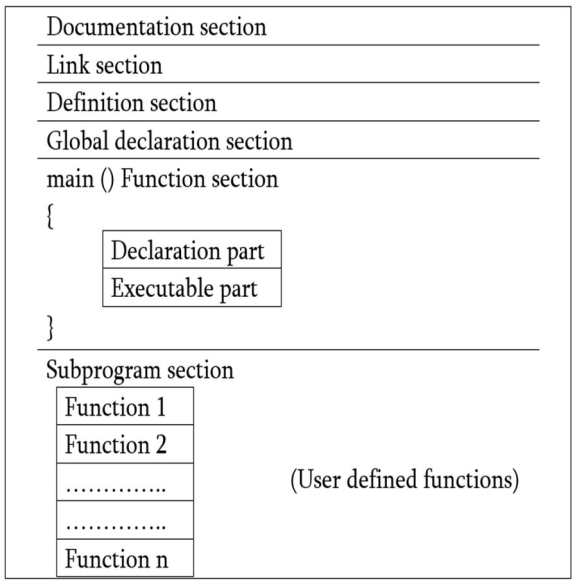
Ken Thompson began to work on the development of a new file system. He wrote, a version of the new file system for the DEC PDP-7, in assembler. (The new file system was also used for the game Space Travel). Soon they began to make improvements and add expansions. (They used their knowledge from the Multics project to add improvements). After a while a complete system was born. Brian W. Kernighan called the system UNIX, a sarcastic reference to Multics. The whole system was still written in assembly code.

Besides assembler and FORTRAN, UNIX also had an interpreter for the programming language B. (The B language is derived directly from Martin Richards BCPL). The language B was developed in 1969-70 by Ken Thompson. In the early days computer code was written in assembly code. To perform a specific task, you had to write many pages of code. A high-level language like B made it possible to write the same task in just a few lines of code. The language B was used for further development of the UNIX system. Because of the high-level of the B language, code could be produced much faster, then in assembly.

A drawback of the B language was that it did not know data-types. (Everything was expressed in machine words). Another functionality that the B language did not provide was the use of "structures". The lag of these things formed the reason for Dennis M. Ritchie to develop the programming language C. So in 1971-73 Dennis M. Ritchie turned the B language into the C language, keeping most of the language B syntax while adding data-types and many other changes. The C language had a powerful mix of high-level functionality and the detailed features required to program an operating system. Therefore many of the UNIX components were eventually rewritten in C (the UNIX kernel itself was rewritten in 1973 on a DEC PDP-11).

The programming language C was written down, by Kernighan and Ritchie. In 1983 a committee was formed by the American National Standards Institute (ANSI) to develop a modern definition for the programming language C (ANSI X3J11). In 1988 they delivered the final standard definition ANSI C.

3.3 STRUCTURE OF A 'C' PROGRAM



1. **Documentation section:**
   The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.
2. **Link section:** The link section provides instructions to the compiler to link functions from the system library such as using the *#include directive*.
3. **Definition section:** The definition section defines all symbolic constants such using the *#define directive*.
4. **Global declaration section:** There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the *user-defined functions*.
5. **main () function section:** Every C program must have one main function section. This section contains two parts; declaration part and executable part
   1. **Declaration part:** The declaration part declares all the *variables* used in the executable part.
   2. **Executable part:** There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The *program execution* begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.
6. **Subprogram section:**
   If the program is a multi-function program then the subprogram section contains all the user-defined functions that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

All section, except the main () function section may be absent when they are not required.

First C language Program

```
//Lets see how to write a simple c program
#include <stdio.h>
#include <conio.h>
int main()
{
 printf("Hello,World");
 getch();
 return 0;
}
```

3.4 C Tokens

C language syntax specify rules for sequence of characters to be written in C language. The rule specify how character sequence will be grouped together to form **tokens**. A smallest individual unit in c program is known as C Tokens.

Tokens are either keyword, identifier, constant, variable or any symbol which has some meaning in C language. A C program can also be called as collection of various tokens.
Example of C tokens,

int	curly braces { }	comments	semicolon ;

3.4.1 Comments

Comments are simple text in your C program that increases readability of programs. Compiler ignore anything written as comment in your program.

Example of comments :

Single line Comment
//This is a comment

Single line Comment
/*This is a comment*/

Multi line Comment
/*This is a long
and valid comment*/

Wrong Syntax
//this is not
a valid comment

3.5 Some basic syntax rule for C program

- C is a case sensitive language so all C instructions must be written in lower case letter.
- All C statement must be end with a semicolon.
- Whitespace is used in C to describe blanks and tabs.
- Whitespace is required between keywords and identifiers

3.6 Step By Step Execution Of C Program

Step 1 : Edit

1. This is First Step i.e Creating and Editing Program.
2. First Write C Program using Text Editor , such as [Notpad++, Notpad, vi editor]
3. Save Program by using [.C] Extension.
4. File Saved with [.C] extension is called "Source Program".

Step 2 : Compiling

1. *Compiling C Program :* C Source code with [.C] Extension is given as input to compiler and compiler convert it into Equivalent Machine Instruction.
2. *Compiler Checks for errors* . If source code is error-free then Code is converted into Object File [.Obj].

Step 3 : Checking Errors

1. During Compilation Compiler will check for error, If compiler finds any error then it will report it.
2. User have to re-edit the program.
3. After re-editing program , Compiler again check for any error.
4. If program is error-free then program is linked with appropriate libraries.

Step 4 : Linking Libraries

1. Program is linked with included header files.
2. Program is linked with other libraries.
3. This process is executed by Linker.

Step 5 : Error Checking

1. If run time error occurs then "Run-time" errors are reported to user.

2. Again programmer have to review code and check for the solution.

Assuming that you are using a Turbo C or Turbo C++ compiler here are the steps that you need to follow to compile and execute your first C program…

a) Start the compiler at **C>** prompt. The compiler (TC.EXE is usually present in **C:\TC\BIN** directory). Select **New** from the **File** menu.
b) Type the program.
c) Save the program using **F2** under a proper name (say Program1.c).
d) Use **Ctrl + F9** to compile and execute the program.
e) Use **Alt + F5** to view the output.

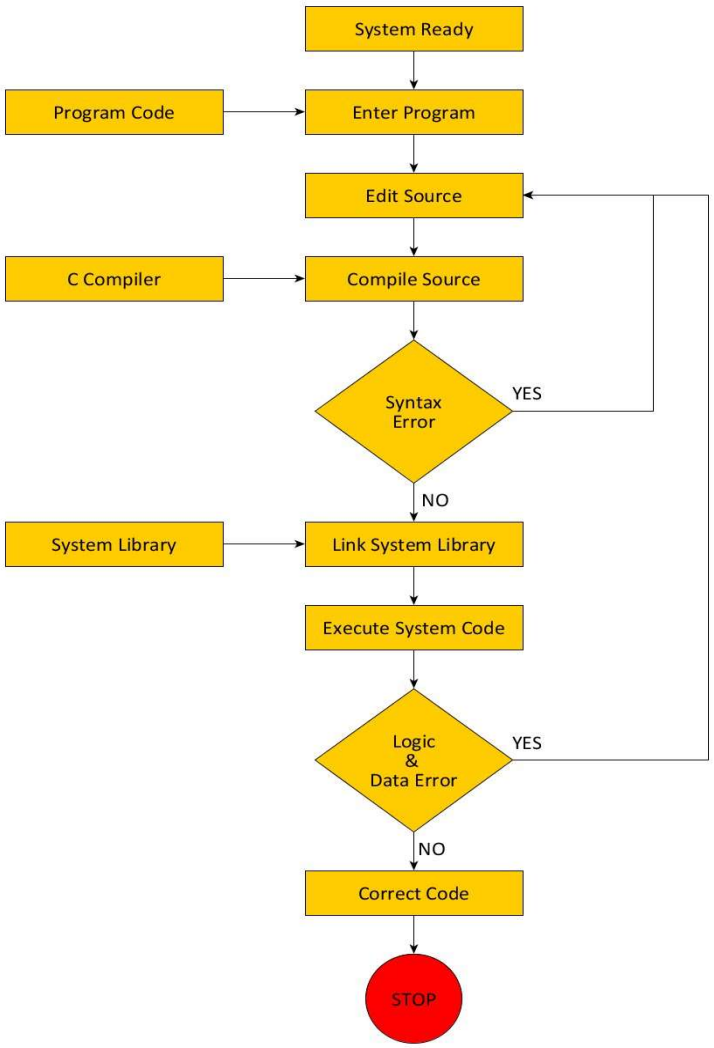


Fig 3.6.1 : Program Execution Process

3.7 Keywords

Keywords are preserved words that have special meaning in C language. The meaning has already been described. These meaning cannot be changed. There are total 32 keywords in C language.

Auto	double	int	struct
break	else	long	switch
case	enum	register	typedef

const	extern	return	union
char	float	short	unsigned
continue	for	signed	volatile
default	goto	sizeof	void
do	if	static	while

3.8 Identifiers

In C language identifiers are the names given to variables, constants, functions and user-define data. These identifier are defined against a set of rules.

3.8.1 Rules for an Identifier

1. An Indetifier can only have alphanumeric characters(a-z , A-Z , 0-9) and underscore(_).
2. The first character of an identifier can only contain alphabet(a-z , A-Z) or underscore (_).
3. Identifiers are also case sensitive in C. For example *name* and *Name* are two different identifier in C.
4. Keywords are not allowed to be used as Identifiers.
5. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

3.8.2 Types of Identifiers:-

There are two types of Identifiers in C. These are:

1. Standard Identifier
2. User defined Identifier

- **Standard Identifier:-**

Like keywords, there are also predefined identifiers in C. The predefined identifiers of the C language that are used for special purposes in the source program are called the standard Identifier. Each identifier has special meanings in C. The names used for standard library functions are standard identifiers.

For Example:-

In C program, the scanf() and printf() functions are mostly used for input or output operations. Therefore, the functions names ‘’scanf’’ or “printf” are examples of standard identifiers.

- **User Defined Identifiers:-**

The user (or programmer) can define its own identifiers in the C program such as variables, user defined functions, labels etc. The Identifiers defined by the user in the program are called user defined Identifiers. C is a case sensitive language.

For Example:-

The C compiler considers ‘Area’ and ‘area’ as two different identifiers.

Character set

In C language characters are grouped into the following catagories,

1. Letters(all alphabets a to z & A to Z).
2. Digits (all digits 0 to 9).
3. Special characters, (such as colon :, semicolon ;, period ., underscore _, ampersand & etc).
4. White spaces.

3.8.3 VARIABLES

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location. For example:

int playerScore = 95;

Here, playerScore is a variable of integer type. The variable is assigned value: 95.

The value of a variable can be changed, hence the name ‘variable’.

In C programming, you have to declare a variable before you can use it.

Rules for naming a variable in C

1. A variable name can have letters (both uppercase and lowercase letters), digits and underscore only.
2. The first letter of a variable should be either a letter or an underscore. However, it is discouraged to start variable name with an underscore. It is because variable name that starts with an underscore can conflict with system name and may cause error.
3. There is no rule on how long a variable can be. However, only the first 31 characters of a variable are checked by the compiler. So, the first 31 letters of two variables in a program should be different.

C is a strongly typed language.

3.9 CONSTANTS/LITERALS

A constant is a value or an identifier whose value cannot be altered in a program. For example: 1, 2.5, "C programming is easy", etc.

As mentioned, an identifier also can be defined as a constant.

Const double PI = 3.14

Here, PI is a constant. Basically what it means is that, PI and 3.14 is same for this program.

Below are the different types of constants you can use in C.

3.9.1. Integer constants

An integer constant is a numeric constant (associated with number) without any fractional or exponential part. There are three types of integer constants in C programming:

- decimal constant(base 10)
- octal constant(base 8)
- hexadecimal constant(base 16)

For example:

Decimal constants: 0, -9, 22 etc

Octal constants: 021, 077, 033 etc

Hexadecimal constants: 0x7f, 0x2a, 0x521 etc

In C programming, octal constant starts with a 0 and hexadecimal constant starts with a 0x.

3.9.2. Floating-point constants

A floating point constant is a numeric constant that has either a fractional form or an exponent form. For example:

-2.0

0.0000234

-0.22E-5

Note: E-5 = 10^{-5}

3.9.3. Character constants

A character constant is a constant which uses single quotation around characters. For example: 'a', 'l', 'm', 'F'

3.9.4. Escape Sequences

Sometimes, it is necessary to use characters which cannot be typed or has special meaning in C programming. For example: newline(enter), tab, question mark etc. In order to use these characters, escape sequence is used.

For example: \n is used for newline. The backslash (\ ) causes "escape" from the normal way the characters are interpreted by the compiler.

Escape Sequences	Character
\b	Backspace
\f	Form feed
\n	Newline
\r	Return

Escape Sequences	Character
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\’	Single quotation mark
\"	Double quotation mark
\?	Question mark
\0	Null character

3.9.5. String constants

String constants are the constants which are enclosed in a pair of double-quote marks. For example:

```
"good"              //string constant
""                  //null string constant
"     "             //string constant of six white space
"x"                 //string constant having single character.
"Earth is round\n"  //prints string with newline
```

3.9.6. Enumeration constants

Keyword enum is used to define enumeration types. For example:

enum color {yellow, green, black, white};

Here, color is a variable and yellow, green, black and white are the enumeration constants having value 0, 1, 2 and 3 respectively.

In C programming, variables or memory locations should be declared before it can be used. Similarly, a function also needs to be declared before use.

3.10 DATA TYPES IN C

Data types simply refers to the type and size of data associated with variables and functions.

1. Fundamental Data Types
    - Integer types
    - Floating type
    - Character type
2. Derived Data Types
    - Arrays
    - Pointers
    - Structures
    - Enumeration

int - Integer data types

Integers are whole numbers that can have both positive and negative values but no decimal values. Example: 0, -5, 10

In C programming, keyword int is used for declaring integer variable. For example:

int id;

Here, id is a variable of type integer.

You can declare multiple variable at once in C programming. For example:

int id, age;

The size of int is either 2 bytes(In older PC's) or 4 bytes. If you consider an integer having size of 4 byte(equal to 32 bits), it can take 2^{32} distinct states as: -2^{31},$-2^{31}+1$, ...,-2, -1, 0, 1, 2, ..., $2^{31}-2$, $2^{31}-1$

Similarly, int of 2 bytes, it can take 2^{16} distinct states from -2^{15} to $2^{15}-1$. If you try to store larger number than $2^{31}-1$, i.e,+2147483647 and smaller number than -2^{31}, i.e, -2147483648, program will not run correctly.

float - Floating types

Floating type variables can hold real numbers such as: 2.34, -9.382, 5.0 etc. You can declare a floating point variable in C by using either float or double keyword. For example:

float accountBalance;

double bookPrice;

Here, both accountBalance and bookPrice are floating type variables.

In C, floating values can be represented in exponential form as well. For example:

float normalizationFactor = 22.442e2;

Difference between float and double

The size of float (single precision float data type) is 4 bytes. And the size of double(double precision float data type) is 8 bytes. Floating point variables has a precision of 6 digits whereas the precision of double is 14 digits.

char - Character types

Keyword char is used for declaring character type variables. For example:

char test = 'h';

Here, test is a character variable. The value of test is 'h'.

The size of character variable is 1 byte.

A character variable holds ASCII value (an integer number between 0 and 127) rather than that character itself in C programming. That value is known as ASCII value.

For example, ASCII value of 'A' is 65.

What this means is that, if you assign 'A' to a character variable, 65 is stored in that variable rather than 'A' itself.

Example 3.10.1 : Program to Print ASCII Value

```
#include <stdio.h>
int main()
{
   char c;
   printf("Enter a character: ");
   // Reads character input from the user
   scanf("%c", &c);
   // %d displays the integer value of a character
   // %c displays the actual character
   printf("ASCII value of %c = %d", c, c);
   return 0;
}
```

3.11 C QUALIFIERS

Qualifiers alters the meaning of base data types to yield a new data type.

Size qualifiers

Size qualifiers alters the size of a basic type. There are two size qualifiers, long and short. For example:

long double i;

The size of double is 8 bytes. However, when long keyword is used, that variable becomes 10 bytes.

There is another keyword short which can be used if you previously know the value of a variable will always be a small number.

Sign qualifiers

Integers and floating point variables can hold both negative and positive values. However, if a variable needs to hold positive value only, unsigned data types are used. For example:

// unsigned variables cannot hold negative value

unsigned int positiveInteger;

There is another qualifier signed which can hold both negative and positive only. However, it is not necessary to define variable signed since a variable is signed by default.
An integer variable of 4 bytes can hold data from -2^{31} to $2^{31}-1$. However, if the variable is defined as unsigned, it can hold data from 0 to $2^{32}-1$.
It is important to note that, sign qualifiers can be applied to int and char types only.

Constant qualifiers
An identifier can be declared as a constant. To do so const keyword is used.
const int cost = 20;
The value of cost cannot be changed in the program.

Volatile qualifiers
A variable should be declared volatile whenever its value can be changed by some external sources outside the program. Keyword volatile is used for creating volatile variables.

3.12 INPUT / OUTPUT STATEMENTS

The C language provides a set of library functions to perform input and output (I/O) operations. Those functions can read or write any type of data to files.
In C, a *file* can refer to a disk file, a terminal, a printer, etc. That is, a file represents a concrete device with which you want to exchange information. The C language treats a file as a series of bytes (or characters). These series of bytes, which are what is really transferred between the file and a program, are called as a whole as a *stream*.
Before you can operate on the file, you have to open that file. In C there are 3 file streams that are pre-opened for you - that is, they are always available for use in the programs.

- stdin: The standard input for reading. Usually it links to your keyboard.
- stdout: The standard output for writing. Usually, it points to your terminal screen.
- stderr: The standard output for writing error messages. Usually, it also points to your terminal screen.

In the following sections we will see different ways to use stdin and stdout. In fact you've already used stdout; when you have used printf, this function sends the output to the default file stream, which points to your screen. The functions we are going to see need the stdio.h library to work.

Using the getc function
The getc function reads the next character from a file stream, and returns the character as an integer.
```
#include <stdio.h>
int getc(FILE *stream);
```
Here FILE *stream represents a file stream variable. If an end-of-file or error occurs, the function returns EOF.
Note
Don't worry about the FILE data structure; now we are gonna use the stdin and stdout file streams, which are predefined. On the other side, EOF is a constant defined in the header file stdio.h. Usually, its value is -1, but keep using EOF, instead of -1; that way, if you later use the program in other compile or operating system that uses a different value.

Using the getchar function
The getchar function is equivalent to getc(stdin).
```
#include <stdio.h>
int getchar(void);
```
Here void indicates that no argument is needed for calling the function, because it will read in from the standard input.
The next program reads two characters typed in by the user from the keyboard, in the two different ways seen so far, and then displays the characters on the screen:
Example 3.12.1
```
#include <stdio.h>
int main(void)
{
  int ch1;
  char ch2;
  printf("Please, type in two characters together:\n");
```

```
  ch1 = getc(stdin);
  ch2 = getchar();
  printf("The first character you have typed is: %c\n",ch1);
  printf("The second character you have typed is: %c\n",ch2);
  return 0;
}
```

As you can see in the program listing, even though the functions expect an integer variable, the ch2 variable can be declared as a char. This is correct because, internally, what is stored of a variable of char type is its numeric value, so they can be passed to the functions that expect an int type.

Using the putc function

The putc function prints out a character on the specified file stream.

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Here c is the character to be written, stored as an integer. The second argument is the file where the character is going to be printed. If an error occurs, putc returns EOF; otherwise, the function returns the written character.

Using the putchar function

The putchar is also used to put a character on the screen. The only difference between it and putc is that putchar needs only the first argument, because the standard output is set as the file stream for putchar.

```
#include <stdio.h>
int putchar(int c);
```

The next program puts the character 'A' on the screen through the two different functions we have seen so far.

Example 3.12.2

```
#include <stdio.h>
int main(void)
{
  int char1 = 65; /*Typically, the numeric value of A*/
  char char2 = 'A';
  printf("The character whose numeric value is 65 is:\n");
  putc(char1,stdout);
  printf("And char2 has the character:\n");
  putchar(char2);
  return 0;
}
```

Using the gets function

To read character by character from the input we have getc and getchar functions. We can also read line by line with several functions; one of them is gets:

```
#include <stdio.h>
char *gets(char *s);
```

Here the characters read from the standard input stream are stored in the character array identified by s.
The gets function stops reading, and appends a null character '\0' to the array, when a newline or end-of-file (EOF) character is encountered. The function returns s if it concludes successfully. Otherwise, a null pointer is returned.

Using the puts function

The puts function can be used to write a sequence of characters to the standard output stream:

```
#include <stdio.h>
int *puts(const char *s);
```

s refers to the character array that contains the string. If the function is successful, it returns 0. Otherwise, it returns a nonzero value.

The following program shows an example of using gets and puts.

Example 3.13.3

```
#include <stdio.h>
#define MAX_LENGTH 80
int main(void)
{
  char string[MAX_LENGTH];
  printf("Please, write a line of no more than 80 characters:\n");
  gets(string);
  printf("The entered line is:\n");
  puts(string);
  return 0;
}
```

Note

As you can see, the use of gets is quite dangerous. Since gets does not know the size of the character array you pass it, it simply reads data until a newline is encountered. This can be a trouble when user types more characters than your array can hold. As a result, your other variables would get overwritten and your program will crash or, at best, behave unpredictably. We will see more secure functions later on.

3.13 FORMATTED INPUT AND OUTPUT FUNCTIONS

Using the scanf function

The scanf function can be used to read various types of input data, such as integers, floats or strings.

```
#include <stdio.h>
int scanf(const char *format,...);
```

Here various format specifiers can be included inside the format string referenced by the char pointer variable format, as with printf. If the scanf function concludes successfully, it returns the number of data items read from stdin. If an error occurs, the scanf function returns EOF.

Using the string format specifier %s tells the scanf function to continue reading characters until a space, a newline, a tab, a vertical tab, or a form feed is encountered. Characters read are stored into an array that should be big enough to store the input characters. A null character is automatically appended to the array after the string is read.

Note

As with gets, it is very dangerous to use scanf to read strings entered by the user, because scanf does not pay attention to the length of the typed strings, and it will admit a string longer than the size defined for the array into which that string is going to be saved. As a result, scanf will write the remained characters in other memory slots which may have data being used by the program. This will eventually produce anomalous behaviours, memory leaks, etc. Later on we will see how to read user input in a secure way.

With scanf, unlike printf, you must ***pass pointers to your arguments (&)*** so that the scanf function can modify their values.

The following program shows how to use scanf with various format specifiers.

Example 3.13.1

```
#include <stdio.h>
#define MAX_LENGTH 80
int main(void)
{
  char string[MAX_LENGTH];
  int integer1, integer2;
  float float_number;
  printf("Enter two integers separated by a space: \n");
  scanf("%d %d", &integer1, &integer2);
  printf("Enter a floating-point number:\n");
  scanf("%f", &float_number);
  printf("Enter a string:\n");
  scanf("%s",string);
```

```
  printf("This is what you have entered:\n");
  printf("%d %d %f %s\n",integer1,integer2,float_numer,string);
  return 0;
}
```

Notice that line 15 uses the scanf function to read a series of characters, and then saves these characters (plus a null character as the terminator) into the array pointed to by string. However, the address-of operator & is not used here, since the name of an array (string in this example) points (is equivalent) to the starting address of the array.

Using the printf function

In C programming language, printf() function is used to print the "character, string, float, integer, octal and hexadecimal values" onto the output screen.

We use printf() function with %d format specifier to display the value of an integer variable.

Similarly %c is used to display character, %f for float variable, %s for string variable, %lf for double and %x for hexadecimal variable.

To generate a newline,we use "\n" in C printf() statement.

Example 3.13.2:

```
#include <stdio.h>
int main()
{
  char ch = 'A';
  char str[20] = "Patna is a nice city";
  float flt = 10.234;
  int no = 150;
  double dbl = 20.123456;
  printf("Character is %c \n", ch);
  printf("String is %s \n" , str);
  printf("Float value is %f \n", flt);
  printf("Integer value is %d\n" , no);
  printf("Double value is %lf \n", dbl);
  printf("Octal value is %o \n", no);
  printf("Hexadecimal value is %x \n", no);
  return 0;
}
```

OUTPUT:

```
Character is A
String is Patna is a nice city
Float value is 10.234000
Integer value is 150
Double value is 20.123456
Octal value is 226
Hexadecimal value is 96
```

You can see the output with the same data which are placed within the double quotes of printf statement in the program except

- %d got replaced by value of an integer variable (no),
- %c got replaced by value of a character variable (ch),
- %f got replaced by value of a float variable (flt),
- %lf got replaced by value of a double variable (dbl),
- %s got replaced by value of a string variable (str),
- %o got replaced by a octal value corresponding to integer variable (no),
- %x got replaced by a hexadecimal value corresponding to integer variable
- \n got replaced by a newline.

Revisiting printf function, you can:

- Specify the minimum field width: The C language allows you to add an integer between the percent sign (%) and the letter in a format specifier. This ensures that the output reaches the minimum width. For example, %10f ensures that the output is at least 10 character spaces wide. This is especially useful when printing out a column of values. For example, if we have the next piece of code:

```
int num = 12;
int num2 = 12345;
printf("%d\n",num2);
printf("%5d\n",num);
```

We will get the following output:

```
12345
 …12
```

- Align output: By default, all output is right-justified when using the minimum field width. You can change this and force output to be left- justified. To do so, you need to prefix the minimum field specifier with the minus sign (-). For example, %-12d specifies the minimum field width as 12, and justifies the output from the left edge of the field.
- Specify precision: You can put a period . and an integer right after the minimum field width specifier to have a precision specifier. You can use the precision specifier to determine the number of decimal places for floating-point numbers, or to specify the maximum field width (or length) for integers or strings.

Example 3.13.3:

```
/* Prints int and float values in various formats */
#include ,stdio.in>
void main( )
{ int i;
float x;
i = 40;
x = 839.21f;
 printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
 printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);
return 0; }
```

- **Output:**

```
 |40|…40|40…| ..040|
|…839.210|   8.392e+02|839.21     |
```

3.14 OPERATORS

C language supports a rich set of built-in operators. An operator is a symbol that tells the compiler to perform certain mathematical or logical manipulations. Operators are used in program to manipulate data and variables.

C operators can be classified into following types,

- Arithmetic operators
- Relation operators
- Logical operators
- Bitwise operators
- Assignment operators
- Conditional operators
- Special operators

3.14.1 Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction and multiplication on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division(modulo division)

Example 3.14.1

// C Program to demonstrate the working of arithmetic operators

```
#include <stdio.h>
int main()
{
   int a = 9,b = 4, c;
   c = a+b;
   printf("a+b = %d \n",c);
   c = a-b;
   printf("a-b = %d \n",c);
   c = a*b;
   printf("a*b = %d \n",c);
   c=a/b;
   printf("a/b = %d \n",c);
   c=a%b;
   printf("Remainder when a divided by b = %d \n",c);
   return 0;
}
```

Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

The operators +, - and * computes addition, subtraction and multiplication respectively as you might have expected. In normal calculation, 9/4 = 2.25. However, the output is 2 in the program.
It is because both variables a and b are integers. Hence, the output is also an integer. The compiler neglects the term after decimal point and shows answer 2 instead of 2.25.
The modulo operator % computes the remainder. When a = 9 is divided by b = 4, the remainder is 1. The % operator can only be used with integers.

Suppose a = 5.0, b = 2.0, c = 5 and d = 2. Then in C programming,

```
a/b = 2.5  // Because both operands are floating-point variables
a/d = 2.5  // Because one operand is floating-point variable
c/b = 2.5  // Because one operand is floating-point variable
c/d = 2    // Because both operands are integers
```

3.14.2 Increment and decrement operators

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example 3.14.2:

```
// C Program to demonstrate the working of increment and decrement operators
#include <stdio.h>
int main()
{
   int a = 10, b = 100;
   float c = 10.5, d = 100.5;
   printf("++a = %d \n", ++a);
   printf("--b = %d \n", --b);
   printf("++c = %f \n", ++c);
   printf("--d = %f \n", --d);
   return 0;
}
```

Output

```
++a = 11
--b = 99
++c = 11.500000
++d = 99.500000
```

Here, the operators ++ and -- are used as prefix. These two operators can also be used as postfix like a++ and a--. However, there is a slight but important difference you should know when these two operators are used as prefix and postfix.

Suppose you use ++ operator as prefix like: ++var. The value of var is incremented by 1 then, it returns the value.
Suppose you use ++ operator as postfix like: var++. The original value of var is returned first then, var is incremented by 1.

Example 3.14.3

```
#include <stdio.h>
int main()
{
 int var=5;
   // 5 is displayed then, var is increased to 6.
 printf("%d\n",var++);
   // Initially, var = 6. It is increased to 7 then, it is displayed.
 printf("%d",++var);
  return 0;
}
```

Output

```
5
7
```

3.14.3 C Assignment Operators

An assignment operator is used for assigning a value to a variable. The most common assignment operator is **'='.**

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example 3.14.4

```
// C Program to demonstrate the working of assignment operators
#include <stdio.h>
int main()
{
   int a = 5, c;
   c = a;
   printf("c = %d \n", c);
   c += a; // c = c+a
   printf("c = %d \n", c);
   c -= a; // c = c-a
   printf("c = %d \n", c);
   c *= a; // c = c*a
   printf("c = %d \n", c);
   c /= a; // c = c/a
   printf("c = %d \n", c);
   c %= a; // c = c%a
   printf("c = %d \n", c);
   return 0;
}
```

Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

3.14 C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 returns 0
>	Greater than	5 > 3 returns 1
<	Less than	5 < 3 returns 0
!=	Not equal to	5 != 3 returns 1
>=	Greater than or equal to	5 >= 3 returns 1
<=	Less than or equal to	5 <= 3 return 0

Example 3.14.4

```
// C Program to demonstrate the working of arithmetic operators
#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10;
   printf("%d == %d = %d \n", a, b, a == b); // true
   printf("%d == %d = %d \n", a, c, a == c); // false
   printf("%d > %d = %d \n", a, b, a > b); //false
   printf("%d > %d = %d \n", a, c, a > c); //false
   printf("%d < %d = %d \n", a, b, a < b); //false
   printf("%d < %d = %d \n", a, c, a < c); //true
   printf("%d != %d = %d \n", a, b, a != b); //false
   printf("%d != %d = %d \n", a, c, a != c); //true
   printf("%d >= %d = %d \n", a, b, a >= b); //true
   printf("%d >= %d = %d \n", a, c, a >= c); //false
   printf("%d <= %d = %d \n", a, b, a <= b); //true
   printf("%d <= %d = %d \n", a, c, a <= c); //true
   return 0;
}
```

Output

```
5 == 5 = 1
5 == 10 = 0
5 > 5 = 0
5 > 10 = 0
5 < 5 = 0
5 < 10 = 1
5 != 5 = 0
5 != 10 = 1
5 >= 5 = 1
5 >= 10 = 0
5 <= 5 = 1
5 <= 10 = 1
```

3.14.5 C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning of Operator	Example
&&	Logial AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c == 5) && (d > 5)) equals to 0.

||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c == 5) || (d > 5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression ! (c == 5) equals to 0.

Example 3.14.5

```
// C Program to demonstrate the working of logical operators
#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10, result;

   result = (a = b) && (c > b);
   printf("(a = b) && (c > b) equals to %d \n", result);

   result = (a = b) && (c < b);
   printf("(a = b) && (c < b) equals to %d \n", result);

   result = (a = b) || (c < b);
   printf("(a = b) || (c < b) equals to %d \n", result);

   result = (a != b) || (c < b);
   printf("(a != b) || (c < b) equals to %d \n", result);

   result = !(a != b);
   printf("!(a == b) equals to %d \n", result);

   result = !(a == b);
   printf("!(a == b) equals to %d \n", result);

   return 0;
}
```

Output

```
(a = b) && (c > b) equals to 1
(a = b) && (c < b) equals to 0
(a = b) || (c < b) equals to 1
(a != b) || (c < b) equals to 0
!(a != b) equals to 1
!(a == b) equals to 0
```

Explanation of logical operator program

- (a = b) && (c > 5) evaluates to 1 because both operands (a = b) and (c > b) is 1 (true).
- (a = b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
- (a = b) || (c < b) evaluates to 1 because (a = b) is 1 (true).
- (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b)are 0 (false).
- !(a != b) evaluates to 1 because operand (a != b) is 0 (false). Hence, !(a != b) is 1 (true).
- !(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b) is 0 (false).

3.14.6 Bitwise Operators

During computation, mathematical operations like: addition, subtraction, addition and division are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators
&	Bitwise AND

Operators	Meaning of operators
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Bitwise AND operator &
The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.
Let us suppose the bitwise AND operation of two integers 12 and 25.
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bit Operation of 12 and 25
 00001100
& 00011001

 ____
 00001000 = 8 (In decimal)

Example 3.14.6.1
```
#include <stdio.h>
int main()
{
   int a = 12, b = 25;
   printf("Output = %d", a&b);
   return 0;
}
```

Output
Output = 8

Bitwise OR operator |
The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. In C Programming, bitwise OR operator is denoted by |.
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
 00001100
| 00011001

 ____
 00011101 = 29 (In decimal)

Example 3.14.6.2
```
#include <stdio.h>
int main()
{
   int a = 12, b = 25;
   printf("Output = %d", a|b);
   return 0;
}
```
Output
Output = 29

Bitwise XOR (exclusive OR) operator ^
The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. It is denoted by ^.
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25
```
  00001100
| 00011001
  ________
  00010101  = 21 (In decimal)
```

Example 3.14.6.3

```
#include <stdio.h>
int main()
{
   int a = 12, b = 25;
   printf("Output = %d", a^b);
   return 0;
}
```
Output
Output = 21

Bitwise complement operator ~
Bitwise compliment operator is an unary operator (works on only one operand). It changes 1 to 0 and 0 to 1. It is denoted by ~.
35 = 00100011 (In Binary)

Bitwise complement Operation of 35
```
~ 00100011
  ________
  11011100  = 220 (In decimal)
```
Twist in bitwise complement operator in C Programming
The bitwise complement of 35 (~35) is -36 instead of 220, but why?
For any integer n, bitwise complement of n will be -(n+1). To understand this, you should have the knowledge of 2's complement.
2's Complement
Two's complement is an operation on binary numbers. The 2's complement of a number is equal to the complement of that number plus 1. For example:

Decimal	Binary	2's complement
0	00000000	-(11111111+1) = -00000000 = -0(decimal)
1	00000001	-(11111110+1) = -11111111 = -256(decimal)
12	00001100	-(11110011+1) = -11110100 = -244(decimal)
220	11011100	-(00100011+1) = -00100100 = -36(decimal)

Note: Overflow is ignored while computing 2's complement.
The bitwise complement of 35 is 220 (in decimal). The 2's complement of 200 is -36. Hence, the output is -36 instead of 220.
Bitwise complement of any number N is -(N+1). Here's how:
bitwise complement of N = ~N (represented in 2's complement form)
2'complement of ~N= -(~(~N)+1) = -(N+1)

Example 3.14.6.3
```
#include <stdio.h>
int main()
{
   printf("complement = %d\n",~35);
   printf("complement = %d\n",~-12);
   return 0;
}
```

Output
complement = -36
Output = 11

3.14.7 Shift Operators in C programming

There are two shift operators in C programming:

- Right shift operator
- Left shift operator.

Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.
212 = 11010100 (In binary)
212>>2 = 00110101 (In binary) [Right shift by two bits]
212>>7 = 00000001 (In binary)
212>>8 = 00000000
212>>0 = 11010100 (No Shift)

Left Shift Operator

Left shift operator shifts all bits towards left by certain number of specified bits. It is denoted by <<.
212 = 11010100 (In binary)
212<<1 = 110101000 (In binary) [Left shift by one bit]
212<<0 =11010100 (Shift by 0)
212<<4 = 110101000000 (In binary) =3392(In decimal)

Example: 3.14.6.4

```
#include <stdio.h>
int main()
{
   int num=212, i;
   for (i=0; i<=2; ++i)
      printf("Right shift by %d: %d\n", i, num>>i);

    printf("\n");

    for (i=0; i<=2; ++i)
      printf("Left shift by %d: %d\n", i, num<<i);

    return 0;
}
```

Output

```
Right Shift by 0: 212
Right Shift by 1: 106
Right Shift by 2: 53

Left Shift by 0: 212
Left Shift by 1: 424
Left Shift by 2: 848
```

3.14.8 Other Operators

Comma Operator

Comma operators are used to link related expressions together. For example:
int a, c = 5, d;

The sizeof operator

The sizeof is an unary operator which returns the size of data (constant, variables, array, structure etc).

Example 3.14.8.1: sizeof Operator

```
#include <stdio.h>
int main()
{
   int a, e[10];
```

```
    float b;
    double c;
    char d;
    printf("Size of int=%lu bytes\n",sizeof(a));
    printf("Size of float=%lu bytes\n",sizeof(b));
    printf("Size of double=%lu bytes\n",sizeof(c));
    printf("Size of char=%lu byte\n",sizeof(d));
    printf("Size of integer type array having 10 elements = %lu bytes\n", sizeof(e));
    return 0;
}
```

Output

Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
Size of integer type array having 10 elements = 40 bytes

C Ternary Operator (?:)

A conditional operator is a ternary operator, that is, it works on 3 operands.

Conditional Operator Syntax

Conditional Expression ? expression1 : expression2

The conditional operator works as follows:

- The first expression conditional Expression is evaluated at first. This expression evaluates to 1 if it's and evaluates to 0 if it's false.
- If conditional Expression is true, expression1 is evaluated.
- If conditional Expression is false, expression2 is evaluated.

Example 3.14.8.2: C conditional Operator

```
#include <stdio.h>
int main(){
   char February;
   int days;
   printf("If this year is leap year, enter 1. If not enter any integer: ");
   scanf("%c",&February);

   // If test condition (February == 'l') is true, days equal to 29.
   // If test condition (February ==='l') is false, days equal to 28.
   days = (February == '1') ? 29 : 28;

   printf("Number of days in February = %d",days);
   return 0;
}
```

Output

If this year is leap year, enter 1. If not enter any integer: 1
Number of days in February = 29

Other operators such as & (reference operator), * (dereference operator) and -> (member selection) operator will be discussed in C pointers.

3.14.9

C Programming operator precedence & associativity

C Programming supports wide range of operators. While Solving the Expression we must follow some rules.

While solving the expression [a + b *c] , we should first perform Multiplication Operation and then Addition, similarly in order to solve such complicated expression you should have hands on *Operator Precedence and Associativity of Operators*.

Operator precedence & associativity table

Operator precedence & associativity are listed in the following table and this table is summarized in decreasing Order of priority i.e topmost operator has highest priority and bottommost operator has Lowest Priority.

Operator	Description	Associativity
( ) [ ] . ->> ++ –	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ – + – ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment	right-to-left

^= |= <<= >>=	Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

Summary of operator precedence

1. **Comma** Operator Has **Lowest Precedence** .
2. **Unary Operators** are Operators having **Highest Precedence**.
3. **Sizeof** is Operator not Function .
4. Operators sharing Common Block in the Above Table have Equal Priority or Precedence .
5. While Solving Expression , Equal Priority Operators are handled on the basis of FIFO [First in First Out] i.e Operator Coming First is handled First.

Important definitions

Unary Operator	A unary operation is an operation with only one operand, i.e. an operation with a single input . A unary operator is one which has only one operand. e.g. post / pre increment operator
Binary Operator	A Binary operator is one which has two operand. e.g. plus , Minus .
Associativity	Their associativity indicates in what order operators of equal precedence in an expression are applied
Precedence	Priority Of Operator

Example 3.14.9.1: Simple use of precedence chart

```
#include<stdio.h>
 int main() {
 int num1 = 10, num2 = 20;
 int result;
 result = num1 * 2 + num2;
 printf("\nResult is : %d", result);
 return (0);
}
```

consider the simple expression used in above program and refer operator precedence & associativity table chart
result = num1 * 2 + num2;
In this case multiplication operator will have higher priority than addition and assignment operator so multiplication will be evaluated firstly.
result = num1 * 2 + num2;
result = 10 * 2 + 20;
result = 20 + 20;
result = 40;
Example 2 : Use of associativity
In above example none of the operator has equal priority. In the below example some operators are having same priority.
result = num1 * 2 + num2 * 2 ;
In the above expression we have overall 3 operators i.e. 2 multiplication, 1 addition and 1 assignment operator. Now refer operator precedence & associativity table (block 3) which clearly tells that associativity is from left to right so we will give priority to multiplication operator on the left side

result = num1 * 2 + num2 * 2 ;
result = 10 * 2 + 20 * 2 ;
result = 20 + 20 * 2 ;
result = 20 + 40 ;
result = 60 ;

3.15 C PROGRAMMING EXPRESSIONS

1. In programming, an expression is any legal combination of symbols that represents a value.
2. C Programming provides its own rules of Expression, whether it is legal expression or illegal expression. For example, in the C language x+5 is a legal expression.
3. Every expression consists of at least one operand and can have one or more operators.
4. Operands are values and Operators are symbols that represent particular actions.

Valid C Programming Expression:
C Programming code gets compiled firstly before execution. In the different phases of compiler, c programming expression is checked for its validity.

Expressions	Validity
a + b	Expression is valid since it contain + operator which is binary operator
+ + a + b	Invalid Expression

In order to solve any expression we should have knowledge of C Programming Operators and their priorities.

Types of Expression :
In Programming, different verities of expressions are given to the compiler. Expressions can be classified on the basis of Position of Operators in an expression –

Type	Explanation	Example
Infix	Expression in which Operator is in between Operands	a + b
Prefix	Expression in which Operator is written before Operands	+ a b
Postfix	Expression in which Operator is written after Operands	a b +

These expressions are solved using the stack.

Examples of Expression :
Now we will be looking into some of the C Programming Expressions, Expression can be created by combining the operators and operands. Each of the expression results into the some resultant output value. Consider few expressions as follows:

n1 + n2,
This is an expression which is going to add two numbers and we can assign the result of addition to another variable.

x = y,
This is an expression which assigns the value of right hand side operand to left side variable

v = u + a * t,
We are multiplying two numbers and result is added to 'u' and total result is assigned to v

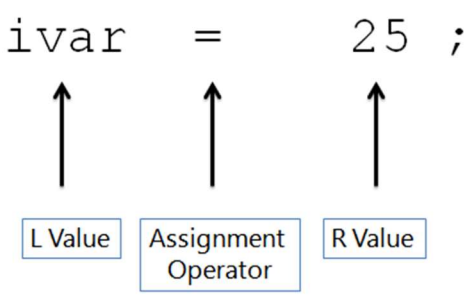
x <= y,
This expression will return Boolean value because comparison operator will give us output either true or false

++j,
This is expression having pre increment operator, it is used to increment the value of j before using it in expression

3.16 L-VALUES AND R_VALUES

1. L-Value stands for **left value**
2. L-Value of Expressions refer to a memory locations
3. In any assignment statement L-Value of Expression must be a container(i.e. must have ability to hold the data)
4. Variable is the only container in C programming thus L Value must be any Variable.
5. L Value Cannot be Constant,Function or any of the available data type in C

Diagram Showing L-Value of Expression :



Example 3.16.1 : L-Value of Expression :

```
#include<stdio.h>
int main()
{
  int num;
  num = 5;
return(0);
}
```

In the above expression, Constant value 5 is being assigned to a variable 'num'. Variable 'num' is called as storage region's , 'num' can considered as LValue of an expression.

Below are some of the tips which are useful to make your concept about L-Value of Expression more clear.

Lvalue cannot be a Constant

```
int main()
{
int num;

5 = num;  //Error

return(0);
}
```

You cannot assign the value or any constant value to another constant value because meaning of constant is already defined and it cannot be modified.

Lvalue cannot be a Constant Variable

Even we cannot assign a value to a constant variable. Constant variable should not be used as L Value.

```
int main()
{
const num;
num = 20;  //Error
return(0);
}
```

Lvalue cannot be a MACRO

We know that macros gets expanded before processing source code by compiler. All the macros will be replaced by defined value using pre-processor before compiling program.

```
#define MAX 20
int main()
{
MAX = 20;  //Error
return(0);
}
```

pre-processor will replace all the occurances of macros and hand over modified source code to compiler. following code will be given to compiler after doing pre-processor task

```
#define MAX 20

int main()
{
20 = 20;
return(0);
}
```

Lvalue cannot be a Enum Constant

```
enum {JAN,FEB,MARCH};
int main()
{
JAN = 20;  //Error
return(0);
}
```

Lvalue cannot be a Data Type

```
#define<stdio.h>
#define max 125
struct book{
  char *name;
  int  pages;
};
void main()
{
book = {"C Programming",100};
}
```

L-Value Require error

<u>Causes of Error :</u>

1. Whenever we are trying to **assign value to constant value**
2. Whenever we are **trying to increment or decrement Constant Expression**
3. Inside if statement , if we try to write comparison operator as **"= ="** instead of "==", then we will get this error.

R-Value of Expression

R Value stands for **<u>Right value</u>** of the expression.

1. In any **Assignment statement** R-Value of Expression must be anything which is capable of returning Constant Expression or Constant Value.
2. R Value Can be anything of following –

Examples of R-Value of Expression	

Examples of R-Value of Expression	
Variable	Constant
Function	Macro
Enum Constant	Any other data type

R value may be constant or constant expression

```
num = 20;      //Constant RValue
num = 20 + 20; //Constant Expression as RValue
```

R value may be MACRO

```
#define MAX 20
main() {
  int num = MAX;
}
```

If we have defined a MACRO then we can use MACRO as right value. In the above example we have assigned the 20 to the variable “num”.
In case if we defined the MACRO value of different data type then it may results into compile error.

R Value may be variable

```
#define MAX 20
main()
{
  int flag = 0;
  int num  = flag;
}
```

3.17 TYPE CONVERSION IN C

Type casting

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the **cast operator** as follows −

(type_name) expression

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation −

Example 3.17.1:

```
#include <stdio.h>
main() {
  int sum = 17, count = 5;
  double mean;
  mean = (double) sum / count;
  printf("Value of mean : %f\n", mean );
}
```

When the above code is compiled and executed, it produces the following result −

Value of mean : 3.400000

It should be noted here that the cast operator has precedence over division, so the value of **sum** is first converted to type **double** and finally it gets divided by count yielding a double value.

Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the **cast operator**. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

Integer Promotion

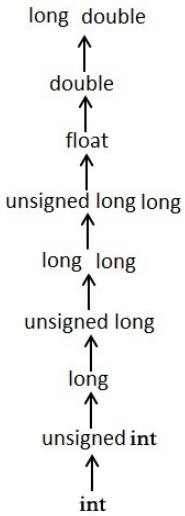
Integer promotion is the process by which values of integer type "smaller" than **int** or **unsigned int** are converted either to **int** or **unsigned int**. Consider an example of adding a character with an integer −

Example 3.17.2:

```
#include <stdio.h>
main() {
   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   int sum;
   sum = i + c;
   printf("Value of sum : %d\n", sum );
}
```

When the above code is compiled and executed, it produces the following result −
Value of sum : 116
Here, the value of sum is 116 because the compiler is doing integer promotion and converting the value of 'c' to ASCII before performing the actual addition operation.

Type Conversion
The *usual arithmetic conversions* are implicitly performed to cast their values to a common type. The compiler first performs *integer promotion*; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy −



The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && and ||. Let us take the following example to understand the concept −

Example 3.17.3:

```
#include <stdio.h>
main() {
   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   float sum;
   sum = i + c;
   printf("Value of sum : %f\n", sum );
}
```

When the above code is compiled and executed, it produces the following result −
Value of sum : 116.000000
Here, it is simple to understand that first c gets converted to integer, but as the final value is double, usual arithmetic conversion applies and the compiler converts i and c into 'float' and adds them yielding a 'float' result.

Difference between Type Conversion and Type Casting

Type Conversion is that which converts to data type into another for example converting a int into float converting a float into double The Type Conversion is that which automatically converts the one data type into another but remember we can store a large data type into the other for example we can't store a float into int because a float is greater than int.

When a user can convert the one data type into then it is called as the type casting Remember the type Conversion is performed by the compiler but a casting is done by the user for example converting a float into int Always Remember that when we use the type C onversion then it is called the promotion when we use the type casting means when we convert a large data type into another then it is called as the demotion when we use the type casting then we can loss some data.

3.18 Summary

- The C programming language was devised in the early 1970s by Dennis M. Ritchie an employee from Bell Labs (AT&T).
- The C language had a powerful mix of high-level functionality and the detailed features required to program an operating system. Hence it is called as a middle level language.
- A smallest individual unit in c program is known as C Tokens.
- Comments are simple text in your C program that increases readability of programs. Compiler ignore anything written as comment in the program.
- There are total 32 keywords in C language.
- In C language identifiers are the names given to variables, constants, functions and user-define data.
- In C programming, you have to declare a variable before you can use it.
- In C programming, octal constant starts with a 0 and hexadecimal constant starts with a 0x.
- Sometimes, it is necessary to use characters which cannot be typed or has special meaning in C programming. In order to use these characters, escape sequence is used.
- printf() and scanf() functions are used for formatted input and output operations.
- An operator is a symbol that tells the compiler to perform certain mathematical or logical manipulations.
- The modulo operator % computes the remainder.
- A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.
- Type casting is a way to convert a variable from one data type to another data type.