JEPPIAAR INSTITUTE OF TECHNOLOGY

“Self-Belief | Self Discipline | Self Respect”

DEPARTMENTOF

INFORMATION TECHNOLOGY

LECTURE NOTES

OCS752- INTRODUCTION IN C PROGRAMMING

Year/Semester:IV/07/ECE

2021 – 2022

Prepared by

MS. S. SCINTHIA CLARINDA

Assistant Professor/IT

SYLLABUS

UNIT I INTRODUCTION
Structure of C program – Basics: Data Types – Constants –Variables - Keywords – Operators: Precedence and Associativity - Expressions - Input/Output statements, Assignment statements – Decision-making statements - Switch statement - Looping statements – Pre-processor directives - Compilation process – Exercise Programs: Check whether the required amount can be withdrawn based on the available amount – Menu-driven program to find the area of different shapes – Find the sum of even numbers.

UNIT II ARRAYS
Introduction to Arrays – One dimensional arrays: Declaration – Initialization - Accessing elements – Operations: Traversal, Insertion, Deletion, Searching - Two dimensional arrays: Declaration – Initialization - Accessing elements – Operations: Read – Print – Sum – Transpose – Exercise Programs: Print the number of positive and negative values present in the array – Sort the numbers using bubble sort - Find whether the given is matrix is diagonal or not.

UNIT III STRINGS
Introduction to Strings - Reading and writing a string - String operations (without using built-in string functions): Length – Compare – Concatenate – Copy – Reverse – Substring – Insertion – Indexing – Deletion – Replacement – Array of strings – Introduction to Pointers – Pointer operators – Pointer arithmetic - Exercise programs: To find the frequency of a character in a string - To find the number of vowels, consonants and white spaces in a given text - Sorting the names.

UNIT IV FUNCTIONS
Introduction to Functions – Types: User-defined and built-in functions - Function prototype - Function definition - Function call - Parameter passing: Pass by value - Pass by reference - Built-in functions (string functions) – Recursive functions – Exercise programs: Calculate the total amount of power consumed by ‘n’ devices (passing an array to a function) – Menu-driven program to count the numbers which are divisible by 3, 5 and by both (passing an array to a function) – Replace the punctuations from a given sentence by the space character (passing an array to a function)

UNIT V STRUCTURES
Introduction to structures – Declaration – Initialization – Accessing the members – Nested Structures – Array of Structures – Structures and functions – Passing an entire structure – Exercise programs: Compute the age of a person using structure and functions (passing a structure to a function) – Compute the number of days an employee came late to the office by considering his arrival time for 30 days (Use array of structures and function

UNIT I INTRODUCTION

1.1 Program

1.1.1 Definition

A program is a set of ordered instructions written in a particular programming language to perform a specific task in a computer. The program is then translated into machine code by compiler and linker so that computer can execute it directly or run it line by line by interpreter program. Few of the programming languages include C, C++, FORTRAN, PASCAL, Java, Python etc. Programs can be classified as application or system programs.

Application program or application software is a computer software package that performs a specific function directly for an end user or, in some cases, for another application. Examples of application programs are word processors, spreadsheets, database systems, and web browsers etc., System program or system software are programs that allow the computer hardware to interact with the programmer and the user, leading to the effective execution of application software on the computer. System programs include operating system, compilers, linkers, interpreters as well as the run-time libraries.

1.1.2 Compilation and Execution

Computers are electronic devices that could understand only machine language, which contains only zeros and ones. If an instruction has to be given to computer, it should be in binary form that is ones and zeros. Writing programs in binary form those computers understand is very difficult for humans. So we create software, which will convert our known language (high level language) into machine level language to make computers work. This software is known as compilers and the process is known as compilation (Figure 1.1.). On compilation we a file known as object file which is machine language. Commonly used programming languages C, C++, FORTRAN, PASCAL, Java, Python etc., are high-level languages.

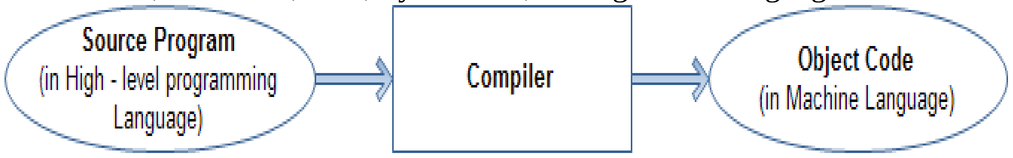


Figure 1.1 Program compilation

For some programming languages interpreters are used as an alternative for compilers. The major differences between interpreters and compilers are as follows (Table 1.1):

Interpreters	Compilers
Translation is done one statement at a time.	Scans and translates the entire code into machine code.

Lesser amount of time is taken for analyzing the source code but the overall execution time is of the program is slower.	Large amount of time is taken to analyze the source code but the overall execution time of the program is comparatively faster.
Memory efficient as no intermediate object code is generated.	Requires more memory as intermediate object code is generated and further requires linking.
Continues the translation of the program until the first error is met, then it stops. Hence debugging is easy.	The error message is generated only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby uses interpreters.	Programming language like C, C++ uses compilers

Table 1.1 Interpreters Vs Compilers

Object file can be directly loaded into computer memory and could be run. The process of loading object file and running that file is known as execution (Figure 1.2). The execution process in details is performed by linking the object code with the other necessary object code or libraries and loading programs into main memory. The processor then takes each instruction from main memory and executes those instructions.

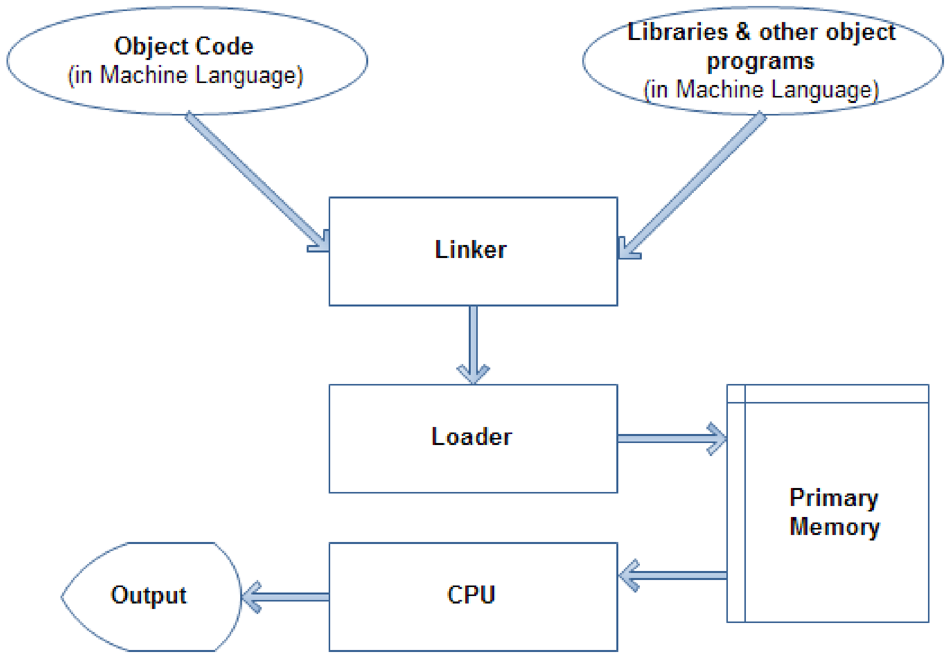


Figure 1.2 Program execution

1.2 Programming Paradigms

A Programming paradigm is the basic style of programming which defines how the structure and elements of a program will be organized to provide a solution to any particular problem statement. The capabilities and limitations of a programming language are based on the programming paradigm it follows. Some programming languages follow a single programming paradigm while few languages follow more than one paradigm. The evolution of high level programming languages resulted in shift in programming paradigms. The programming paradigms are classified as follows:

- Monolithic programming – emphasizes on providing solutions to a problem statement.
- Procedural programming – emphasizes on set of subroutines and is an algorithmic approach.
- Structured programming – focuses on creating and invoking modules.
- Object oriented programming – comprises of writing classes and instantiating real-time objects.
- Logic-oriented programming – focuses on goals expressed usually in terms of predicate calculus.
- Rule-oriented programming – uses 'if-then-else' rules for computation.
- Aspect oriented programming – focuses on global concerns like logging, authentication, security, performance etc.

Each of these programming paradigms has its own features and limitations and always no single programming paradigm could suit all types of applications. For example for designing computation intensive application, procedure oriented programming is suitable; for designing knowledge base, rule-oriented programming is the suitable option. In this section we will discuss on the first four programming paradigms.

1.2.1 Monolithic Programming

The programs written in BASIC or assembly language follows monolithic programming paradigm. The monolithic programming paradigm comprises of sequential code and global data. The sequential code is one in which the instructions are executed in a particular sequence. In case of changing the sequence 'goto' or 'jump' statements are used. The entire program is a single module with no concept of subroutines and the global data can be accessed from any part of program.

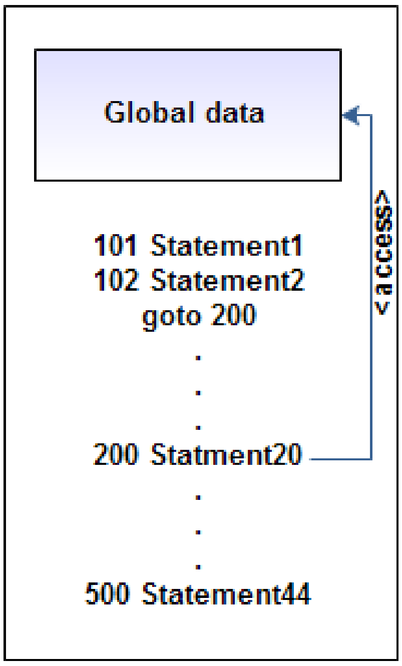


Figure 1.3 Monolithic programming

Advantages:

- Suitable for small and simple programs.
- Debugging and maintenance of code is easy since programs are smaller in size and executed in a sequence.

Disadvantages:

- No concept of reusability.
- The data is not fully protected as the global data is accessed from anywhere.
- Increase in program size makes coding, debugging and maintenance to be difficult.

1.2.2 Procedural Programming

Program is subdivided into subroutines that perform a well-defined task. This reduces repetition of code. A subroutine if required can call another subroutine. The sequence of execution is altered using 'jump', 'call' or 'goto' statements. FORTRAN and COBOL are two popular procedural programming languages.

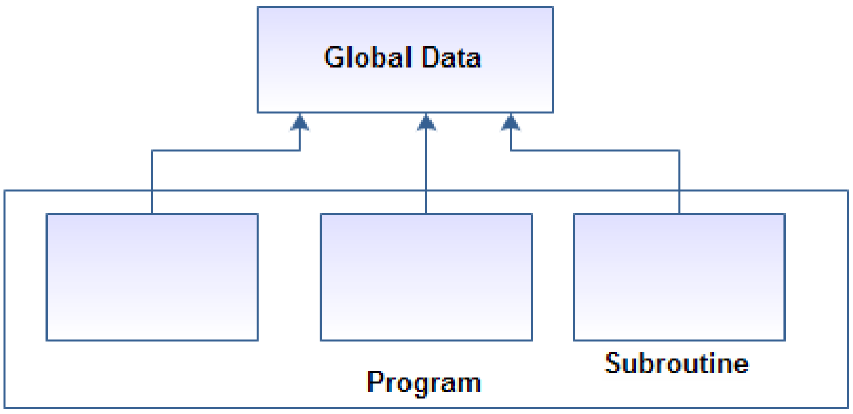


Figure 1.4 Procedural programming

Advantages:

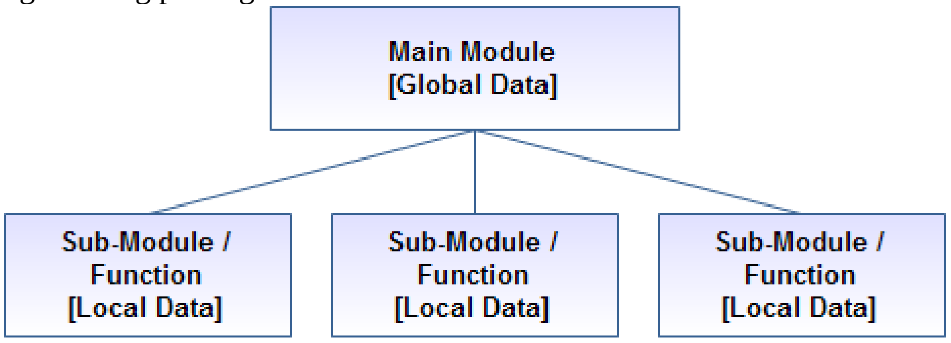
- Programs are easier to write when compared to monolithic programming.
- Code reusability is provided.

Disadvantages:

- The data is not fully protected as the global data is used.
- The programs are difficult to maintain.
- The more time and effort is required to write programs.

1.2.3 Structural Programming

Structured programming also referred to as modular programming was suggested in 1966 by mathematicians Corrado Bohm and Guiseppe Jacopini. The structured programming was defined aiming at large programs that require a team of developers to develop different parts of the system. Structured programming employs a top-down approach where the system has a main-module and further modularization is done till individual pieces of code can be written easily and could not be further sub-divided. The languages such as C, Pascal etc., support the concept of structured programming paradigm.


Main Module
[Global Data]
Sub-Module / Function [Local Data]
Sub-Module / Function [Local Data]
Sub-Module / Function [Local Data]
Figure 1.5 Structural programming

Advantages:

- It takes less time to write the programs and also modules written for one program can be reused in other programs as well.

- Each module is developed for a specific task and could be coded, debugged and updated individually.

Disadvantages:

- It is not data centric.
- More emphasis is given to the code.
- Global data is shared and may get inadvertently modified.

1.2.4 Object Oriented Programming

The object oriented programming paradigm consists of easily maintainable and reusable code encapsulated with necessary data within itself namely objects. The objects in the program interact with each other by message passing leading to necessary features of the application being developed. The programming languages such as C++,Java, Ada are object oriented programming languages.

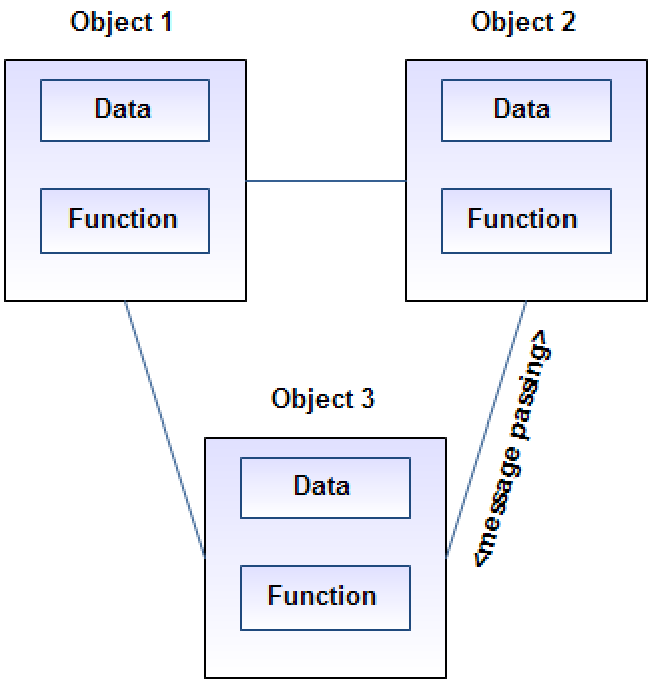


Figure 1.6 Object oriented programming

Advantages:

- Data is hidden and only accessible by functions in the object and not by external function.

- New data and function can be added whenever needed.

2.1 Structure of C Program

C is a structured programming language and hence comprises of modules or functions. A program written in C language is composed of Preprocessor directives, Global declarations and a set of functions as shown in the Figure 2.1

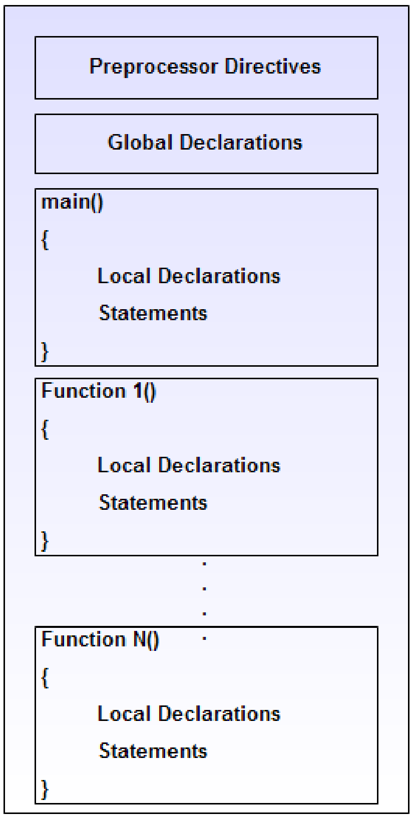


Figure 2.1 Structure of a C Program

- The Preprocessor directives are the special instructions that indicate how to prepare the program for compilation. One of the mostly used preprocessor directive is **include**, that instructs the compiler to include the specified header files needed for compilation.
- Global declarations include the variable and function declarations. The global variables are common to all functions including main function. Functions are by default global and function declaration is also termed as function prototype.

- All the functions including main() are divided into two parts- Local declarations and statements. Local declaration consists of variable declarations that are used within the function and not visible from outside the function. The statements contain the code that manipulates the data to perform a specific task.
- The main() function is most significant part of any C program and is mandatory since the execution of the program begins at the function.
- The comments could be included in the program anywhere in the program and is optional. The comments are statements that convey a message and to increase the readability of the program. They are not processed by the compiler. The comments could be a single-line comment or multi-line comment.
    - Single-line comment starts with // and terminates at the end of line.
    - Multi-line comment starts with /* and terminates with */ and is used to write comments of multiple lines.

Here is a sample program to print the message "First program in C" in the below example.

```
// C Program to print ""First program in C"
/* Preprocessor directive to include header file stdio.h that consists of function to
perform input and output on standard input and output device */
# include <stdio.h>
main()   // main function – beginning of the execution
{
        printf(" First program in C ");
}
```

On compiling and executing this program the message "First program in C" will be printed on the console.

2.2 Preprocessor Directives

The preprocessor is a program that processes the C program before it is passed through the compiler. The preprocessor looks for the presence of preprocessor directive; if present preprocessor expands the preprocessor directives and takes the corresponding action before any code is generated by the program statements.

- Preprocessor directives are always preceded by hash sign (#).
- No semicolon has to be placed at the end to specify end of preprocessor directive. Whereas newline character is the end of the directive.
- However the preprocessor directives may contain comments.
- To extend preprocessor directive to multiple lines, backslash (\)character is present as the last character of the line.
- Preprocessor directive is usually placed before the main(), but it can appear anywhere in the program; However written in the middle the directive will be applied only in the reminder of the source file.

Advantages:

The advantages of the preprocessor directives in the C program include:

- Program becomes readable and easy to understand.
- Modification of program becomes easy.
- Program becomes portable since preprocessor directives make it easy to compile the program in different execution environment.
- Due to all the above said advantages the program also becomes more efficient to use.

Types:

Preprocessor directives are broadly classified as unconditional and conditional preprocessor directives. The unconditional directives are defined to perform a well-defined task whereas conditional directives are used to instruct the preprocessor to select whether or not to include a chunk of code in final token stream passed to the compiler. #define, #line, #undef, #include, #error and #pragma are the set of unconditional directives. #if, #else, #elif, #ifdef, #ifndef and #endif are the conditional directives. There are also few predefined macro names in C such as _LINE_, _DATE_, _TIME_ etc that returns specified line number, date of compilation of the source file, time of starting the compilation correspondingly. We will look at each of the directives in detail.

2.2.1 #DEFINE:

#define statement is also known as macro definition or simply a macro. There are two types of macros.

- Object-like macro and
- Function-like macro

Object-like macro is a simple identifier which will be replaced by a code fragment. It does not take any argument. The preprocessor replaces every occurrence of the identifier in the source code by a string. The general syntax for defining a macro is given as

```
#define identifier string
```

The macro start with #define, should be followed by valid identifier and a string with at least one blank space between them. The line below defines a macro named PI as an abbreviation for the token 3.14.

```
#define PI 3.14
```

If there is a C statement

```
area=PI * radius * radius
```

then preprocessor will recognize and expand the macro PI. The C compiler will see the token as it would have written

```
area=3.14 * radius * radius
```

Function-like macro is used to simulate functions. While simulating function the name as macro serves as header and the macro body serves as the function body. In general function body is invoked at runtime by the function call whereas in macro the source code or macro body is inserted at the place of occurrence of macro header by replacing the corresponding arguments at compile time. Using macro instead of function is more efficient since they avoid the overhead involved in calling a function.

The syntax of a function-like macro can be given as

```
#define identifier(arg1,arg2,….argn) string
```

The identifier is function name, followed by the parameter list and replacement string. The macro name can be in lowercase, but as a convention you must write it in uppercase characters. The following line defines a macro ADD having two parameters a and b and the replacement string is (a+b).

```
#define ADD(a,b) (a+b)
```

If the C statements are as below

```
int a=5, b=3;
c=ADD(a,b);
```

then the output of preprocessor will be

```
c=a+b;
```

2.2.2 #INCLUDE:

The #include directive is used to inform the preprocessor to treat the contents of specified file as if present in the source program at the point where the directive appears. The #include directive are of two forms based on the way it is specified.

```
#include <filename>
```

variant means that the filename is system header file and search is made for such filename in the standard list of system directories.

```
#include “filename”
```

variant means that the header file is users own header file and hence the search starts from the directory of current file, then in the quote directories and them the same directories used for <filename>.

The header files are the files with extension .h which consists of function declarations and macro definitions that need to be shared between several source files. There are few standard header files that are used often such as stdio.h, stdlib.h, math.h, string.hetc. The stdio.h file defines core input and output functions, stdlib.h defines numeric conversion functions, pseudo-random numbers generation functions, memory allocation and process control functions. Whereas string.h defines string handling functions and math.h defines common mathematical functions. The #include directive is defined as follows:

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

2.2.3 #UNDEF:

The #undef directive undefined or removes a macro name previously created with #define. Like definition, undefinition also occurs at a specific point in the source file, and it applies starting from that point. If the macro is defined with parameters, while undefining there is no need to give the parameter list. The syntax of undef directive is as follows:

```
#undef identifier
```

Where identifier is a macro name that is already defined using #define. For example two macro defined before main() is undefined at the end of main().

```
#define MAX 10
        #define ADD(X,Y) (X+Y)
        main()
{
.
.
}
#undef MAX
#undef ADD
```

2.2.4 #LINE:

The #line directive supplies line number for compiler messages. It tells the preprocessor to change the compiler's internally stored line number and filename to a given line number and filename. When an error is generated during compilation process, compiler will show an error message with reference to the name of file assigned where error has happened and the assigned line number. This enables to easily detect the erroneous code and rectify it. The syntax for #line directive is

```
#line line_number filename
```

Here the line_number is the new line number that will be assigned to next line of code. The parameter filename is optional parameter that redefines the filename that will appear in case an error occurs.

Consider the following program

```
#include <stdio.h>
main()
{
        #line 10 “Error.c”
        int a=10:
        #line 20
        printf(“%d”,a);
}
```

The code will generate an error that will show error in file “Error.c”, line 10 and line 20.

2.2.5 #ERROR:

The error directive is used to produce compile-time error messages. The syntax of this directive is:

```
#error string
```

When #error directive is encountered the compilation process gets terminated and the message specified in the string is printed to stderr. The following code illustrates the usage of #error directive.

```
#ifndef MACRO
        #error MACRO not defined
#endif
```

2.2.6#PRAGMA:

A #pragma directive is an instruction to the compiler and is usually ignored during preprocessing. The syntax of #pragma directive is given as

#pragma string

Here the string is one of instruction given to the compiler with any required parameters. The effect of pragma will be applied from the point where it is included to the end of compilation unit or until another pragma changes its status.

Instruction	Description	Syntax	Purpose
COPYRIGHT	To specify a copyright string	#pragma COPYRIGHT string	String specifies the set of characters included in the copyright message of object file
COPYRIGHT_DATE	To specify a copyright date for a copyright string	#pragma COPYRIGHT string	String is the date used by the COPYRIGHT pragma
HP_SHLIB_VERSION	To create version of a shared library	#pragma HP_SHLBB_VERSION [“]date[“]	Date is month/year
VERSIONID	To specify a version string	#pragma VERSIONID “string”	String is a string of characters stored in object file

Table 2.1 Pragma directives

2.2.7 #IFDEF

#ifdefia simplest form of conditional preprocessor directive and is used to check for the existence of macro definition. The syntax is given as

```
#ifdef MACRO
        controlled text
```

```
#endif
```

The controlled text will be executed if MACRO is defined. The following example defines a stack array of MAX is defined for the preprocessor.

```
#ifdef MAX
        int STACK[MAX];
#endif
```

The stack array will not be created if MAX is not defined.

2.2.8#IFNDEF

#ifndefia simplest form of conditional preprocessor directive and is used to check for the absence of macro definition. The syntax is given as

```
#ifndef MACRO
        controlled text
#endif
```

The controlled text will be executed if MACRO is not defined. The following example defines a MAX value if not defined yet.

```
#ifndef MAX
        #define MAX 10
#endif
```

2.2.9 #IF, #ELIF AND #ELSE:

```
#if expression
conditional codes
#endif
```

Here, expression is a expression of integer type (can be integers, characters, arithmetic expression, macros and so on). The conditional codes are included in the program only if the expression is evaluated to a non-zero value.

The optional #else directive can used with #if directive.

```
#if expression
conditional codes if expression is non-zero
#else
conditional if expression is 0
#endif
```

You can also add nested conditional to your #if...#else using #elif

```
#if expression
conditional codes if expression is non-zero
#elif expression1
conditional codes if expression is non-zero
#elif expression2
conditional codes if expression is non-zero
... .. ...
else
```

```
        conditional if all expressions are 0
        #endif
```
Here is an example for usage of nested #if...#else using #elif for setting up of maximum value.
```
#define MAX 7
#if MAX > 200
        #undef MAX
#defineMAX 200
        #elif MAX < 50
        #undef MAX
        #defineMAX 50
#else
        #undef MAX
        #define MAX 100
#endif
```

2.2.10 #DEFINED

The special operator defined is used to test whether certain macro is defined or not. It's often used with #if directive. This is example for using defined operator.
```
        #if defined BUFFER_SIZE && BUFFER_SIZE >= 2048
                int  STACK[MAX];
```

2.2.11 PREDEFINED MACROS:

There are some predefined macros which are readily available for use in C programming. Here is a table of Pre-defined macros.

Predefined macro	Value
__DATE__	String containing the current date
__FILE__	String containing the file name
__LINE__	Integer representing the current line number
__STDC__	If follows ANSI standard C, then value is a nonzero integer
__TIME__	String containing the current date.

Table 2.2 Predefined macros

We can use these macros in the program to access the value of macros and here is an example to access the current date.

```
#include <stdio.h>
int main()
{
printf("Current time: %s",__TIME__);   //calculate the current time
}
```

The output will be

```
Current time: 19:54:39
```

2.3 Elements of C

The basic building block of C language is the smallest individual unit called token. The program is constructed using combination of tokens. The main types of token in C are

- Character Set
- Keywords
- Variables
- Constants
- Strings
- Special Characters
- Operators

2.3.1 Character Set:

The Fundamental unit to represent information in a language is character set. The character set of C can be given as:

- Alphabets: include both upper and lower case letters (a-z and A-Z).
- Digits: includes numerical digits (0-9).
- Special characters such as: !,@,#,$,%,^,&,*,(,),{,},[,],<,>,?,etc.
- White space characters used to print bank space on screen.
  - \b - blank space
  - \t - Horizontal tab
  - \v - vertical return
  - \r - Carriage return
  - \f - Form feed
  - \n - Newline
- Escape sequences: these are non-printing control characters that begin with the back slash (\). For example
  - \\ - Back slash
  - \? - Question mark
  - \’ - Single quote
  - \” - Double quote
  - \x - Hexadecimal Constant
  - \0 - Octal Constant

2.3.2 Keywords:

In every language there are few words which are reserved for specific purpose and could not be used as identifier are termed as Keywords. Table 2.3 shows the list of keywords in C.

Auto	break	Case	char	const	continue	default
Double	else	enum	extern	float	for	goto
Int	long	register	return	short	signed	sizeof
Struct	switch	typedef	union	unsigned	void	volatile
Do	if	static	while			

Table 2.3 Keywords in C

2.3.3 Identifiers:

Identifiers are the names given to program elements such as variables, arrays and functions. There are certain rules to be followed to form identifier names

- Identifiers cannot include special characters such as(#,$,?,etc.)except the _.
- There cannot be two successive underscores.
- Keywords could not be used as identifiers.
- Identifiers can be of any reasonable length, whereas compiler looks for only first 31 characters of the name hence it should not be more than 31 characters.
- Identifiers are case sensitive.
- Identifier should not start with digits and should start with a letter or underscore. The underscore is not usually preferred because the identifiers in the standard library start with _ and there may be duplicate names causing conflicts.

Few valid identifiers include:

student_name, marks, EMP_NO, RollNo, emp_name

Examples of invalid identifiers include:

21_studentname, $marks, auto, emp__name

2.3.4 Data Types:

Data types specify how and what type of data we use in our program. C language supports 2 different types of data

- Primary data types:

These are fundamental data types in C namely integer (int), floating point (float), character (char) and void.

- Derived data types:

Derived data types are combinations or grouping ofprimary datatypes like array, structure, union and pointer.

Data type determines the type of data a variable will hold. If a variable x is declared as int. it means x can hold only integer values. Every variable which is used in the program must be declared as what data-type it is.

Integer type:

Integers are used to store whole numbers. Size and range of Integer type on 16-bit machine is given in Table 2.4.

Type	**Size(bytes)**	**Range**
int or signed int	2	-32,768 to 32767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Table 2.4 Integer data types in C

Floating point type:

Floating types are used to store real numbers. Size and range of Integer type on 16-bit machine is given in Table 2.5.

Type	Size(bytes)	Range
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

Table 2.5 Float data types in C

Character type:

Character types are used to store characters value. Size and range of Integer type on 16-bit machine is given in Table 2.6.

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

Table 2.6 Character data types in C

void type:

void type means no value. This is usually used to specify the type of functions which returns nothing. A variable cannot of type void.

2.3.5 Variables:

Variable is the name given to storage location in computer memory. The value of a variable could be changed at any point during the execution. A programmer can choose a purposeful as well as valid identifier name for a variable. For example: emp_name, BASIC, RollNo etc. The variables can be declared by specifying the data type and variable name as below:

```
int emp_number;
float BASIC;
unsigned short intacc_no;
char gender;
```

The variable could also be assigned with values during declaration and is termed as initialization. For example:

```
int emp_number = 62789;
float BASIC   =  15000;
unsigned short intacc_no =  478798706;
char gender = ‘f’;
```

2.3.6 Constants:

Constants are the identifiers whose values do not change even by mistake. The value of constant will be known to compiler during compilation. Constants in a C program can be of integer type, float type, character type or of string type. Examples of constants are as follows:

- Integer constants: 1, 34, 123u, 1234, 012, 0X12 here suffix ‘u’ represents unsigned int, (could also be ‘U’) , ‘0’ represents octal integer, ‘0X’ represents hexadecimal constant.
- Float constants: 0.02f , 99.80, 0.5e2 b by default 0.02 will be treated as double hence suffixed with ‘f’ to make it as float and ‘e’ for exponentiation.
- Character constants: ‘a’, ‘F’ these are stored as ASCII values.

2.3.7 Enumeration Constants:

Enumeration constant or enum is a enumerated data type that is user defined. The syntax for defining aenum data type is

```
enum identifier{value1, value2,…..valuen};
```

here a user defined data type is created with the identifier name and set of values that could be assigned for the identifier. For example:

```
enum month {Sunday,Monday,Tuesday, Wednesday, Thursday,Friday,Saturday};
        enum month currday;
```

The variable currday can be assigned with any of the values amongst 7 days.

```
currday = Friday;
```

The default values will be assigned from 0 to 6 from Sunday to Saturday correspondingly. The default values could also be reassigned on need.

```
enum month {Sunday=07 ,Monday, Tuesday, Wednesday=3, Thursday, Friday,Saturday};
```

2.3.8 Storage Classes:

Each variable has a storage class which decides scope, default initial values and life time of a variable. The storage classes most often used in C programming are

1. Automatic variables
2. External variables
3. Static variables
4. Register variables

Automatic variables (auto):

Scope: Variable defined with auto storage class is local to the function block inside which they are defined.

Default Initial Value: Any random value i.e garbage value.

Lifetime: Till the end of the function/method block where the variable is defined.

A variable declared inside a function without any storage class specification, is by default an automatic variable. They are created when a function is called and are destroyed automatically when the function's execution is completed. Automatic variables can also be called local variables because they are local to a function. By default they are assigned garbage value by the compiler.

```
#include<stdio.h>
void main()
{
int detail;
   // or
autoint details;    //Both are same
}
```

External or Global variable:

Scope: Global i.e everywhere in the program. These variables are not bound by any function, they are available everywhere.

Default initial value: 0(zero).

Lifetime: Till the program doesn't finish its execution, you can access global variables.

A variable that is declared outside any function is a Global Variable. Global variables remain available throughout the program execution. By default, initial value of the Global variable is 0(zero). One important thing to remember about global variable is that their values can be changed by any function in the program.

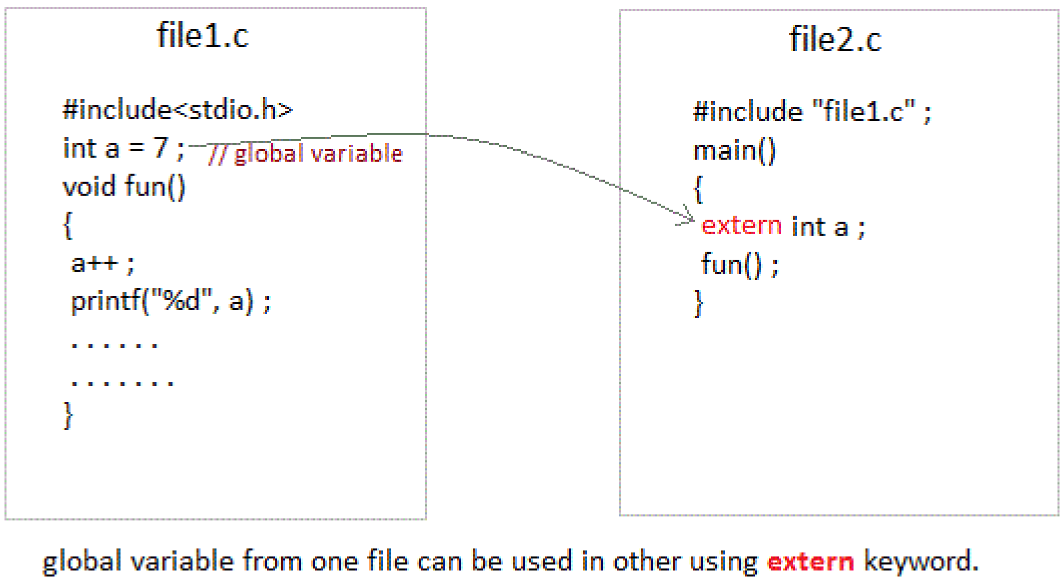
```
#include<stdio.h>
int number;// global variable
void main()
{
            number=10;
            printf("Value in main function is %d\n", number);
            fun1();//function calling
            fun2();//function calling
}
/* This is function 1 */
fun1()
{
number=20;
printf("Value in fun1 is %d", number);
}
/* This is function 1 */
fun2()
{
printf("\nValue in fun2 is %d", number);
}
```

The output will be as follows

```
Value in main function is 10
Value infun1 is 20
Value infun2 is 20
```

Here the global variable number is available to all three functions and thus, if one function changes the value of the variable, it gets changed in every function.

The extern keyword is used with a variable to inform the compiler that this variable is declared somewhere else. The extern declaration does not allocate storage for variables.



```
#include<stdio.h>
int a = 7 ; // global variable
void fun()
{
 a++ ;
 printf("%d", a) ;
 . . . . . .
 . . . . . . .
}
```

```
#include "file1.c" ;
main()
{
 extern int a ;
 fun() ;
}
```

global variable from one file can be used in other using **extern** keyword.

Figure 2.2 Use of extern keyword

Program without extern will be as below with error been generated.

```
int main()
{
    a =10;//Error: cannot find definition of variable 'a'
    printf("%d", a);
}
```

Program with extern variable will be as below.

```
int main()
{
    extern int x;//informs the compiler that it is defined
    //somewhere else
    x =10;
    printf("%d", x);
}
int x;//Global variable x
```

Static variables:

Scope: Local to the block in which the variable is defined

Default initial value: 0(Zero).

Lifetime: Till the whole program doesn't finish its execution.

A static variable tells the compiler to persist/save the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, static variable is initialized only once and remains into existence till the end of the program. A static variable can either be internal or external depending upon the place of declaration. The scope of internal static variable remains inside the function in which it is defined. External static variables remain restricted to scope of file in which they are declared.

They are assigned 0 (zero) as default value by the compiler.

```
#include<stdio.h>

void test();//Function declaration (discussed in next topic)

int main()
{
        test();
        test();
        test();
}

void test()
{
        static int a =0;//a static variable
         a = a +1;
        printf("%d\t",a);
}
```

The output will be

```
1 2 3
```

Register variable:

Scope: Local to the function in which it is declared.

Default initial value: Any random value i.e garbage value

Lifetime: Till the end of function/method block, in which the variable is defined.

Register variables inform the compiler to store the variable in CPU register instead of memory. Register variables have faster accessibility than a normal variable. Generally, the frequently used variables are kept in registers. But only a few variables can be placed inside registers. One application of register storage class can be in using loops, where the variable gets used a number of times in the program, in a very short span of time. We can never get the address of such variables.

Syntax :

```
register int number;
```

Even though we have declared the storage class of our variable number as register, we cannot surely say that the value of the variable would be stored in a register. This is because the number of registers in a CPU are limited. Also, CPU registers are meant to do a lot of important work. Thus, sometimes they may not be free. In such scenario, the variable works as if its storage class is auto.

Which storage class should be used and when it should be used?

To improve the speed of execution of the program and to carefully use the memory space occupied by the variables, following points should be kept in mind while using storage classes:

- We should use static storage class only when we want the value of the variable to remain same every time we call it using different function calls.
- We should use register storage class only for those variables that are used in our program very often. CPU registers are limited and thus should be used carefully.
- We should use external or global storage class only for those variables that are being used by almost all the functions in the program.
- If we do not have the purpose of any of the above mentioned storage classes, then we should use the automatic storage class.

2.3.9 Operators

An operator is a symbol that specifies the mathematical, logical or relational operation to be performed. The operators supported by C language could be group as

- Arithmetic operators

- Relational operators
- Logical operators
- Bitwise operators
- Conditional operators
- Assignment operators
- Special operators

Arithmetic operators:

C supports all the basic arithmetic operators. The following table 2.7 shows all the basic arithmetic operators with their purpose and usage. Consider the value of a and b as 9 and 3 respectively.

Operator	Purpose	Usage
+	adds two operands	c = a + b value of c will be 12
-	subtracts second operand from first	c = a – b value of c will be 3
*	multiplies two operands	c = a * b value of c will be 27
/	divides numerator by denominator	c = a / b value of c will be 6
%	reminder of division	c = a % b value of c will be 0
++	increment operator	increases integer value by one. a++; a values becomes 10 ++a; a value becomes 11
--	decrement operator	Decreases integer value by one. a--; a value becomes 10 --a; a value becomes 9

Table 2.7 Arithmetic operators

Relational operators:

Relational operators or comparison operator is used to compare two variables.

Given below in the table 2.8 set of relational operators and few samples of how they operate.

Operator	Purpose	Usage
==	Returns 1 if both operands are equal, 0 otherwise	a==b returns 0

!=	Returns 1 if both operands are not equal, 0 otherwise	a==b returns 1
>	Checks if operand on left side is greater than the right	a>b gives 1
<	Checks if operand on left side is lesser than the right	a<b gives 0
>=	Checks if operand on left side is greater than or equal to the right	a>b gives 1
<=	Checks if operand on left side is lesser than or equal to the right	a<b gives 0

Table 2.8 Relational Operators

Logical Operators:

C language supports three logical operators Logical AND (&&), Logical OR (||) and Logical NOT (!). Table 2.9 shows the usage of logical operators when the values of x and y are 1 and 0 respectively.

Operator	Purpose	Usage
&&	Logical AND	(a&&b) is false ; true if both the values are 1
||	Logical OR	(a||b) is true ; true if any one value is 1
!	Logical NOT	(!a) is false ; true if 0 and false if 1

Table 2.9 Logical Operators

Bitwise operators:

Bitwise operators perform manipulations of data at bit level. These operators also perform shifting of bits from right to left. Bitwise operators are not applied to float or double. The table 2.10 shows the purpose and usage of bitwise operators for a value 1100 and b is 1010.

Operator	Purpose	Usage
&	Bitwise AND	(a&b) is 1000 ; 1 if both the bit values are 1 else 0
|	Bitwise OR	(a|b) is 1110 ; 1 if any one bit value is 1 else 0
^	Bitwise exclusive OR	(a^b) is 0110 ;

		1 if change of bit value else 0
<<	Left shift	b << 2 is 1000 shifts b two bits left
>>	Right shift	b >>2 is 0010 shifts b two bits right

Table 2.10 Bitwise operators

Conditional Operator:

The conditional or ternary operator (? :) is just like if-else statement that can be used within an expression. The ternary operator is of the form

exp1 ?exp2 : exp2

where exp1 is evaluated and if the result is true exp2 is evaluated and the result is returned or exp3 is evaluated and the result is returned. For example

max = a > b ? a : b

if value of a and b is 3 and 5, max = 5

Assignment Operators:

In C assignment variables are responsible for assigning values to the variable. Table 2.11 shows various assignment operators, their purpose and usage if value of b is 1.

Operator	Purpose	Usage
=	assigns values from right side operands to left side operand	a = b a is assigned with value of b 1

+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
		a=a+1
		a=1+1 ; a=2

-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
		a=a-1
		a=2-1; a=1

*=	mutiply left operand with the right operand and assign the result to left operand	a*=b is same as a=a*b
		a=a*1
		a=1*1;a=1

/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
		a=a/1
		a=1/1;a=1

%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b
		a=a%1
		a=1%1;a=0

Table 2.11 Assignment operators

Special Operators:

There are few special operators in c language and the table 2.12 shows the operators their purpose and usage.

Operator	Purpose	Usage
sizeof	Returns the size of an variable	**sizeof(x)** return size of the variable **x**
&	Returns the address of an variable	**&x ;** return address of the variable **x**
*	Pointer to a variable	***x ;** will be pointer to a variable **x**
,	Evaluates expression in the order of left to tight	Int a=2,b=3,x=0; x= (++a,b+=a) the value of x is 6; ++a makes a=3 +=a evaluates to b=6 Finally the result of right side of the expression is assigned.

Table 2.12 Special operators

2.3.10 Expressions:

An expression in C is made up of one or more operands. An expression is a sequence of operators and operands that specifies the computation of a value. Operand is an entity on which the operation is to be performed; it could be a variable name, a constant, a function call or a macro name. Operators specify the operation to be performed on the operands. Based on the number of operators present in an expression the expression could be classified as simple or compound expression.

An expression with only one operator is called simple expression and expression with more than one operator is called compound expression. For example

c=a+b is a simple expression whereas
c=a+b*d/e is a compound expression.

In case of compound expression the precedence and associativity of the operators decided how the evaluation of expression is done and the result is generated.

2.3.11 Precedence and Associativity:

If more than one operator is involved in an expression, C language has a predefined rule of priority for the operators. This rule of priority of operators is called operator precedence. In C, precedence of arithmetic operators(*, %, /, +, -) is higher than relational operators(==, !=, >, <, >=, <=) and precedence of relational operator is higher than logical operators(&&, || and !).

For example consider the expression

(1 > 2 + 3 && 4)

This expression is equivalent to:

((1 > (2 + 3)) && 4)

i.e, (2 + 3) executes first resulting into 5

Then, first part of the expression (1 > 5) executes resulting into 0 (false)

Then, (0 && 4) executes resulting into 0 (false)

If two operators of same precedence (priority) are present in an expression, Associativity of operators indicates the order in which they execute.

For example consider the expression

1 == 2 != 3

Here, operators == and != have same precedence. The associativity of both == and != is left to right, i.e, the expression on the left is executed first and moves towards the right.

Thus, the expression above is equivalent to :

((1 == 2) != 3)
i.e, (1 == 2) executes first resulting into 0 (false)
then, (0 != 3) executes resulting into 1 (true)

The table 2.13 presents the precedence and associativity of the operators in the precedence order from top to bottom.

Operator	Meaning of operator	Associativity
() [] -> .	Functional call Array element reference Indirect member selection Direct member selection	Left to right
! ~ + - ++	Logical negation Bitwise(1 's) complement Unary plus Unary minus Increment	Right to left

Operator	Meaning of operator	Associativity
--	Decrement	
&	Dereference Operator(Address)	
*	Pointer reference	
sizeof	Returns the size of an object	
(type)	Type cast(conversion)	
*	Multiply	Left to right
/	Divide	
%	Remainder	
+	Binary plus(Addition)	Left to right
-	Binary minus(subtraction)	
<<	Left shift	Left to right
>>	Right shift	
<	Less than	Left to right
<=	Less than or equal	
>	Greater than	
>=	Greater than or equal	
==	Equal to	Left to right
!=	Not equal to	
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
?:	Conditional Operator	Right to left
=	Simple assignment	Right to left

Operator	Meaning of operator	Associativity
*= /= %= -= &= ^= |= <<= >>=	Assign product Assign quotient Assign remainder Assign sum Assign difference Assign bitwise AND Assign bitwise XOR Assign bitwise OR Assign left shift Assign right shift	
,	Separator of expressions	Left to right

Table 2.13 C operators with precedence and associativity

2.4 Statements in C

C programs are generally collection of Statements, statements is an executable part of the program to do some action. The statements are grouped based on the purpose and the flow of control.

2.4.1 Input/Output Statements:

Before performing input and output in C program. Let us have a small introduction on the concept of streams and standard input and output device. Stream is sequence of bytes of data flow in and out of the program. In C we have two types of streams

- Text Stream and
- Byte Stream

The text stream has sequence of characters divided into lines and each line is terminated by new line character (\n). On the other had a binary stream contains data values using their memory representation. Pre-defined stream in C are

- stdin – standard input stream (reads input from keyboard)
- stdout – standard output stream (prints the output on console or monitor)
- stderr – standard error stream also an output stream (prints the error messages on the console.

The following figure 2.3 shows the representation of input and output flow using streams. Here the input and output are from standard input/output devices and input can also be from a file or data could be stored in a file. The discussion of this could be done during file handling chapter.

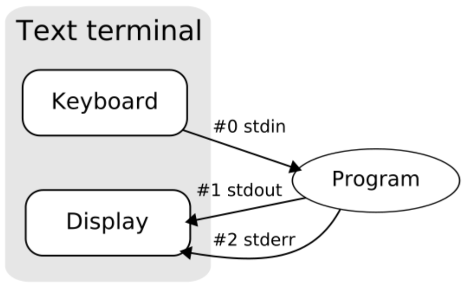


Figure 2.3 Input and output streams in C

The input and output statements can be broadly classified as Formatted and unformatted input and output statements. Formatted input/output statements include scanf() and printf() where as unformatted input/ output statement includes getch(), putch(), gets(), puts(), getche(), getchar() and putchar().

printf():

The standard input-output header file, named stdio.h contains the definition of the functions printf() which is used to display output on screen.

The syntax for printf() is as follows

printf(“control string”, variable list);

The control string in printf() consists of

- Conversion specification which has data value type, data value size and specific format information.
- Control characters
- Textual data

For example

```
main()
{
        int i=5;
        printf(“The square of %d is %d\n”,i,i*i);
}
```

Here control string “The square of %d is %d\n” we have textual data “The square of is “, control character “\n” and conversion specification “%d”. Here in control specification only data value type is given where %d represents integer.

Format String	Meaning
%d	Scan or print an integer as signed decimal number
%f	Scan or print a floating point number

%c	To scan or print a character
%s	To scan or print a character string. The scanning ends at whitespace.

Table 2.14 Format Specifiers for Input/Output

We can also limit the number of digits or characters that can be input or output, by adding a number with the format string specifier, like "%1d" or "%3s", the first one means a single numeric digit and the second one means 3 characters, hence if you try to input 42, while scanf() has "%1d", it will take only 4 as input. Same is the case for output.

In C Language, computer monitor, printer etc output devices are treated as files and the same process is followed to write output to these devices as would have been followed to write the output to a file. The printf() function returns the number of characters printed by it,

```
int i = printf("CProgram");
```

In this program printf("CProgram"); will return 8 as result, which will be stored in the variable i, because CProgram has 12 characters.

scanf():

The standard input-output header file, named stdio.h contains the definition of the functions scanf(), which is used to take input from user respectively.

The syntax for scanf() is as follows

```
scanf( " control string", argument list);
```

The control string in scanf() starts with % symbol and has format specifier to receive input from the user givenin figure 2.11. The scanf() returns the number of characters read by it.

For example,

```
main()
{
        int i=0;
        scanf("%1d", &i);
        printf("The square of %d is %d\n",i,i*i);
}
```

Here the control string "%1d" represents the input should be a single digit integer. &i represents the address of variable i where the input has to be stored.

getchar() & putchar() functions:

The getchar() function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in a loop in case you want to read more than one character. The putchar() function displays the character passed to it on the screen and returns the same character. This function too displays only a single character at a time. In case you want to display more than one characters, use putchar() method in a loop.

```
#include <stdio.h>
void main( )
{
        int c;
        printf("Enter a character");
        // Take a character as input and store it in variable c
        c = getchar();
        //  display the character storedin variable c
        putchar(c);
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

---

gets() & puts() functions:

The gets() function reads a line from stdin(standard input) into the buffer pointed to by str pointer, until either a terminating newline or EOF (end of file) occurs. The puts() function writes the string str and a trailing newline to stdout.

str → This is the pointer to an array of chars where the C string is stored.

```
#include<stdio.h>
void main()
{
   /* character array of length 100 */
        char str[100];
        printf("Enter a string");
        gets(str );
        puts(str );
        getch();
}
```

When you will compile the above code, it will ask you to enter a string. When you will enter the string, it will display the value you have entered.

2.4.2 Assignment statements:

C provides an **assignment operator** for this purpose, assigning the value to a variable using assignment operator is known as an assignment statement in C. The function of this operator is to assign the values or values in variables on right hand side of an expression to variables on the left hand side.

The syntax of the *assignment expression*

variable = constant / variable/ expression;

The data type of the variable on left hand side should match the data type of constant/variable/expression on right hand side with a few exceptions where automatic type conversions are possible.

Examples of assignment statements,

```
b = c ; /* b is assigned the value of c */
a = 9 ; /* a is assigned the value 9*/
b = c+5; /* b is assigned the value of expr c+5 */
```

The expression on the right hand side of the assignment statement can be:

- An arithmetic expression;
- A relational expression;
- A logical expression;
- A mixed expression.

2.4.3 Decision making statements

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C language handles decision-making by supporting the following statements,

- if statement
- switch statement
- conditional operator statement (? : operator)
- goto statement

Decision making with if statement:

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

- Simple if statement
- if....else statement
- Nested if....else statement
- Using else if statement

Simple if statement

The general form of a simple if statement is,

```
if(expression)
{
        statement inside;
}
        statement outside;
```

If the expression returns true, then the statement-inside will be executed, otherwise statement-inside is skipped and only the statement-outside is executed.

Consider the example to find whether given number is even:

```
#include <stdio.h>
void main( )
{
        int x=6;
```

```
                if (x %2==0 )
                {
                        printf("x is even");
                }
        }
```

The output will be

```
x is even
```

---

if...else statement

The general form of a simple if...else statement is,

```
if(expression)
{
statement block1;
}
else
{
statement block2;
}
```

If the expression is true, the statement-block1 is executed, else statement-block1 is skipped and statement-block2 is executed.

Consider the example to find whether given number is even or odd:

```
        #include <stdio.h>
        void main( )
        {
        int x=7;
        if (x %2==0 )
           {
        printf("x is even");
           }
        else
        {
        printf("x is odd");
        }
      }
```

The output will be

```
x is odd
```

Nested if....else statement

The general form of a nested if...else statement is,

```
        if( expression )
        {
```

```
if( expression1 )
   {
statement block1;
   }
else
   {
statement block2;
   }
}
else
{
statement block3;
}
```

if expression is false then statement-block3 will be executed, otherwise the execution continues and enters inside the first if to perform the check for the next if block, where if expression 1 is true the statement-block1 is executed otherwise statement-block2 is executed.

Here is an example to find greatest among three numbers using nested if-else statement.

```
#include <stdio.h>
void main( )
{
int a, b, c;
printf("Enter 3 numbers...");
scanf("%d%d%d",&a, &b, &c);
if(a > b)
   {
if(a > c)
      {
printf("a is the greatest");
      }
else
      {
printf("c is the greatest");
      }
   }
else
   {
if(b > c)
      {
printf("b is the greatest");
      }
else
```

```
        {
printf("c is the greatest");
        }
    }
}
```

---

else if ladder

The general form of else-if ladder is,

```
if(expression1)
{
statement block1;
}
else if(expression2)
{
statement block2;
}
else if(expression3 )
{
statement block3;
}
else
default statement;
```

The expression is tested from the top(of the ladder) downwards. As soon as a true condition is found, the statement associated with it is executed.

For example the below program checks whether the given number is divisible by 5 or divisible by 8 or divisible by 5 and 8 or not divisible by both 5 and 8.

```
#include <stdio.h>
void main( )
{
        int a;
        printf("Enter a number...");
        scanf("%d", &a);
        if(a%5 == 0 && a%8 == 0)
   {
        printf("Divisible by both 5 and 8");
   }
else if(a%8 == 0)
   {
        printf("Divisible by 8");
   }
else if(a%5 == 0)
```

```
   {
         printf("Divisible by 5");
   }
else
   {
         printf("Divisible by none");
   }
}
```

Switch Statements

Switch statement is a control statement that allows us to choose only one choice among the many given choices. The expression in switch evaluates to return an integral value, which is then compared to the values present in different cases. It executes that block of code which matches the case value. If there is no match, then default block is executed(if present). The general form of switch statement is,

```
switch(expression)
{
case value-1:
        block-1;
        break;
case value-2:
        block-2;
        break;
case value-3:
        block-3;
        break;
case value-4:
        block-4;
        break;
default:
        default-block;
        break;
}
```

Rules for using switch statement:

- The expression (after switch keyword) must yield an integer value i.e the expression should be an integer or a variable or an expression that evaluates to an integer.
- The case label values must be unique.
- The case label must end with a colon(:)
- The next line, after the case statement, can be any valid C statement.

We don't use those expressions to evaluate switch case, which may return floating point values or strings or characters.break statements are used to exit the switch block. It isn't necessary to use break after each block, but if you do not use it, then all the consecutive blocks of code will get executed after the matching block. In the example no break statement is written

```
int i = 1;
switch(i)
{
case 1:
printf("A");        // No break
case 2:
printf("B");        // No break
case 3:
printf("C");
break;
}
```

The output will be

```
A B C
```

---

whereas the output was supposed to be only A because only the first case matches, but as there is no break statement after that block, the next blocks are executed too, until it a break statement in encountered or the execution reaches the end of the switch block.default case is executed when none of the mentioned case matches the switch expression. The default case can be placed anywhere in the switch case. Even if we don't include the default case, switch statement works.

Nesting of switch statements are allowed, which means you can have switch statements inside another switch block. However, nested switch statements should be avoided as it makes the program more complex and less readable.

An example of calculator program which does addition and subtraction is

```
#include<stdio.h>
void main( )
{
int a, b, c, choice;
while(choice != 3)
   {
      /* Printing the available options */
printf("\n 1. Press 1 for addition");
printf("\n 2. Press 2 for subtraction");
printf("\n Enter your choice");
      /* Taking users input */
scanf("%d", &choice);

switch(choice)
      {
```

```
case 1:
printf("Enter 2 numbers");
scanf("%d%d", &a, &b);
          c = a + b;
printf("%d", c);
break;
case 2:
printf("Enter 2 numbers");
scanf("%d%d", &a, &b);
          c = a - b;
printf("%d", c);
break;
default:
printf("you have passed a wrong key");
printf("\n press any key to continue");
     }
   }
}
```

Difference between if..else and switch is

If..else	**switch**
can evaluate float conditions	cannot evaluate float conditions
can evaluate relational operators	cannot evaluate relational operators i.e they are not allowed in switch statement

Conditional operator statement (? : operator)

Conditional operator statement is replacement for if-else statement. The syntax of conditional operator statement is

Exp1 ?Exp2 : Exp3;

This is replacement of if-else

```
If (exp1)
{
        exp2;
}
else
{
        exp3;
}
```

Here is an example to print whether student is pass or fail using mark.

```
main()
{
        float mark=98;
        if (mark<50)
                printf(“ FAIL”);
        else
                prinft(“PASS”);
}
```

The output will be

```
PASS
```

goto statement

A goto statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function. The syntax for a goto statement in C is as follows

```
goto label;
..
.
label: statement;
```

where Here label can be any plain text except C keyword and it can be set anywhere in the C program above or below to goto statement. Here is an example for goto statement

```
        #include <stdio.h>
        int main()
        {
        int sum=0;
        for(int i = 0; i<=10; i++){
                sum = sum+i;
                if(i==5){
                goto addition;
                }
          }

addition:
        printf("%d", sum);

    return 0;
    }
```

The output is 15

in this example, we have a label addition and when the value of i (inside loop) is equal to 5 then we are jumping to this label using goto. This is reason the sum is displaying the sum of numbers till 5 even though the loop is set to run from 0 to 10.

2.4.4 Looping Statements

In any programming language including C, loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

The below diagram depicts a loop execution,

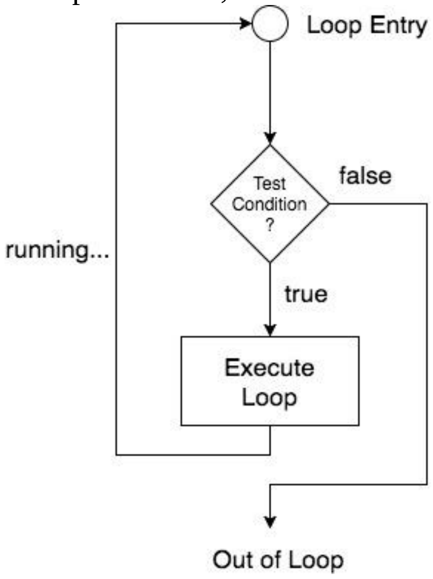


Figure 2.4 Loop execution

As per the above diagram, if the Test Condition is true, then the loop is executed, and if it is false then the execution breaks out of the loop. After the loop is successfully executed the execution again starts from the Loop entry and again checks for the Test condition, and this keeps on repeating.

The sequence of statements to be executed is kept inside the curly braces { } known as the Loop body. After every execution of the loop body, condition is verified, and if it is found to be true the loop body is executed again. When the condition check returns false, the loop body is not executed, and execution breaks out of the loop.

There are 3 types of Loop in C language, namely:

- while loop
- for loop
- do while loop

while loop

while loop can be addressed as an entry control loop. It is completed in 3 steps.

Variable initialization.(e.g int x = 0;)

condition(e.g while(x <= 10))

Variable increment or decrement (x++ or x-- or x = x + 2)

The syntax for while loop is:

```
variable initialization;
while(condition)
{
statements;
variable increment or decrement;
}
```

Here is a an example program to print first 10 natural numbers using while loop.

```
#include<stdio.h>
void main( )
{
int x;
   x = 1;
while(x <= 10)
   {
printf("%d\t", x);
      /* below statement means, do x = x+1, increment x by 1*/
x++;
   }
}
```

The output generated is as follows:

```
1 2 3 4 5 6 7 8 9 10
```

---

for loop

for loop is used to execute a set of statements repeatedly until a particular condition is satisfied. We can say it is an open ended loop. General format of for loop is,

```
for(initialization; condition; increment/decrement)
{
statement-block;
}
```

In for loop we have exactly two semicolons, one after initialization and second after the condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. But it can have only one condition.

The for loop is executed in the following order:

1. It first evaluates the initialization code.
2. Then it checks the condition expression.
3. If it is true, it executes the for-loop body.
4. Then it evaluates the increment/decrement condition and again follows from step 2.
5. When the condition expression becomes false, it exits the loop.

Here is an example program to print first 10 natural numbers using for loop.

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

```
void main( )
{
int x;
for(x = 1; x <= 10; x++)
    {
printf("%d\t", x);
    }
}
```

The output generated is as follows

```
1 2 3 4 5 6 7 8 9 10
```

Nested for loop

We can also have nested for loops, i.e one for loop inside another for loop. The syntax for nested for loop is

```
for(initialization; condition; increment/decrement)
{
for(initialization; condition; increment/decrement)
    {
statement ;
    }
}
```

Here is an example program to print half Pyramid of numbers using nested for loop

```
#include<stdio.h>
void main( )
{
int i, j;
    /* first for loop */
for(i = 1; i < 5; i++)
    {
printf("\n");
        /* second for loop inside the first */
for(j = i; j > 0; j--)
        {
printf("%d", j);
        }
    }
}
```

The output of the above program will be

```
1
```

```
21
321
4321
54321
```

---

do while loop

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of do-while loop. do statement evaluates the body of the loop first and at the end, the condition is checked using while statement. It means that the body of the loop will be executed at least once, even though the starting condition inside while is initialized to be false. General syntax is,

```
do
{
   .....
   .....
}
while(condition)
```

Here is a sample program to print first 10 multiples of 5 using do while loop.

```
#include<stdio.h>
void main()
{
int a, i;
   a = 5;
   i = 1;
do
   {
printf("%d\t", a*i);
i++;
   }
while(i <= 10);
}
```

The output of the above program will be

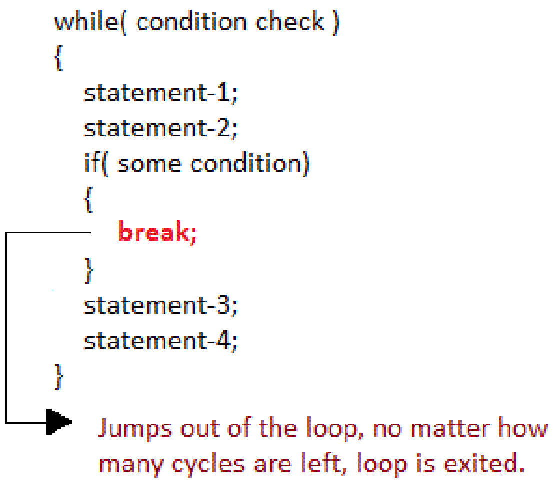
```
5 10 15 20 25 30 35 40 45 50
```

Jumping Out of Loops

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becomes true. This is known as jumping out of loop.

1) break statement

When break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.



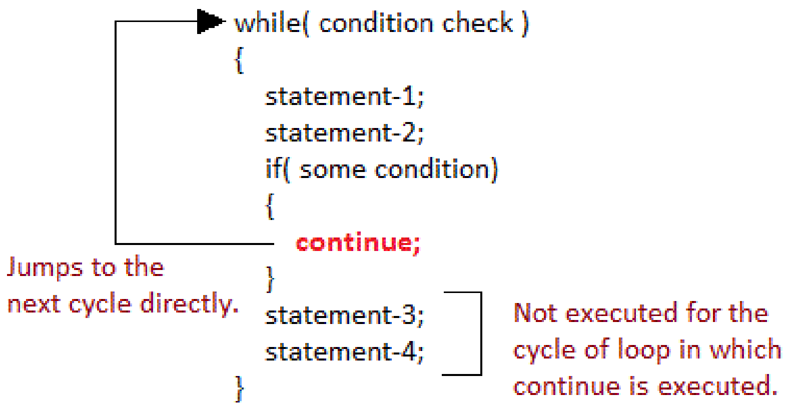
```
while( condition check )
{
   statement-1;
   statement-2;
   if( some condition)
   {
     break;
   }
   statement-3;
   statement-4;
}
```

Jumps out of the loop, no matter how many cycles are left, loop is exited.

Figure 2.5 break statement

2) continue statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering continue, cursor leave the current cycle of loop, and starts with the next cycle.



```
while( condition check )
{
   statement-1;
   statement-2;
   if( some condition)
   {
     continue;
   }
   statement-3;
   statement-4;
}
```

Jumps to the next cycle directly.
Not executed for the cycle of loop in which continue is executed.

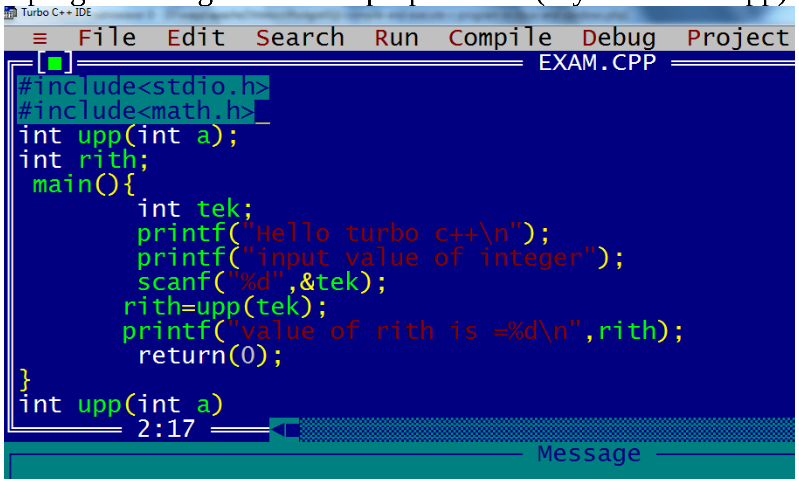
Figure 2.6 continue statement

2.5 Compilation and Execution Process

On making a clear analysis of the problem statement we can write the algorithm and pseudo code for the program under implementation. The program need to be written or typed and can be used for instructing the machine to execute it. To type your C program the program called Editor could be used. Once the program has been typed it needs to be converted to machine language (0's and 1's) before the machine can execute it. To carry out this conversion we need another program called Compiler. Compiler vendors provide an Integrated Development Environment (IDE) which consists of an Editor as well as the Compiler. There are several such IDEs available in the market targeted towards different operating systems. For example, Turbo C, Turbo C++ and Microsoft C are some of the popular compilers that work under MS-DOS; Visual C++ and Borland C++ are the compilers that work under Windows, whereas gcc compiler works under Linux.

In case of Turbo C or Turbo C++ compiler here are the steps that need to be followed to compile and execute the first C program…

i. Start the compiler at C> prompt. The compiler (TC.EXE is usually present in C:\TCBIN directory),a editor will be displayed.
ii. Select New from the File menu.
iii. Type the program.
iv. Save the program using F2 under a proper name (say xxx.c/xxx.cpp).

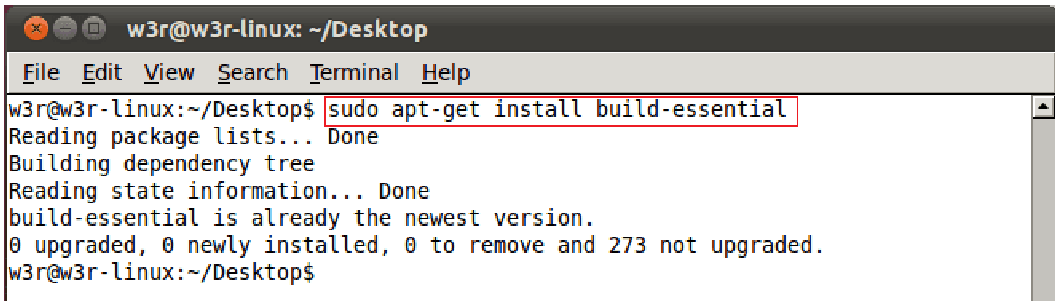


```
#include<stdio.h>
#include<math.h>_
int upp(int a);
int rith;
 main(){
        int tek;
        printf("Hello turbo c++\n");
        printf("input value of integer");
        scanf("%d",&tek);
       rith=upp(tek);
       printf("value of rith is =%d\n",rith);
        return(0);
}
int upp(int a)
```

Figure 2.6 TurboC Editor

v. Use Ctrl + F9 to compile and execute the program.
vi. Use Alt + F5 to view the output.

In case of Linux, most of the time, when installing Linux, GNU Gcc compiler will be installed by default. If not, run the following command (the system here is Ubuntu Linux):



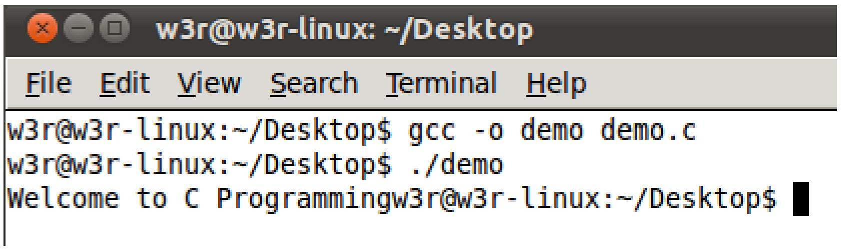
```
w3r@w3r-linux:~/Desktop$ sudo apt-get install build-essential
Reading package lists... Done
Building dependency tree
Reading state information... Done
build-essential is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 273 not upgraded.
w3r@w3r-linux:~/Desktop$
```

Figure 2.7 Linux Command Line Installation

If C compiler is already installed, it will show you a message like above. If not, it will Install all the necessary packages. Now open a text editor and write a small C program like following and save it as demo.c :

```
#include <stdio.h>
main()
{
        printf("Welcome to C Programming");
}
```

Now run the command as shown below to compile and execute the file:



```
w3r@w3r-linux:~/Desktop$ gcc -o demo demo.c
w3r@w3r-linux:~/Desktop$ ./demo
Welcome to C Programmingw3r@w3r-linux:~/Desktop$
```

Figure 2.7 Linux Command Line Execution

This how you can install GNU Gcc compiler, write a C program and run it under Linux.

Summary

- A c program is composed of preprocessor directives, global declarations and one or more functions.
- There are conditional and unconditional preprocessor directives present for aiding the compilation process.
- The execution of c program starts with main() function composed of various function calls.
- The basic elements needed to write a c program comprises of identifiers, variables, constants, character set etc.
- The basic data types in C are int, float and character.
- The operators of different forms such as arithmetic operators, bitwise operators, and logical operators are basic elements used for the construction an expression.
- The expressions are evaluated based on the precedence and associativity of the operators.
- The statements in C programs are the constructs which directs the flow of execution.

Sample Programs

1. *Multiplication table upto 10*

```
#include <stdio.h>
int main()
{
int n, i;
printf("Enter an integer: ");
scanf("%d",&n);
for(i=1; i<=10; ++i)
{
printf("%d * %d = %d \n", n, i, n*i);
}
return 0;
}
```

2. *Print whether a number is positive or negative.*

```
#include <stdio.h>
int main()
{
double number;
printf("Enter a number: ");
scanf("%lf", &number);
if (number <= 0.0)
   {
if (number == 0.0)
printf("You entered 0.");
else
printf("You entered a negative number.");
    }
else
printf("You entered a positive number.");
return 0;
}
```

3. *Check whether given number is prime or not.*

```
#include <stdio.h>
int main()
{
int n, i, flag = 0;

printf("Enter a positive integer: ");
scanf("%d", &n);
for(i = 2; i <= n/2; ++i)
    {
        // condition for nonprime number
if(n%i == 0)
        {
flag = 1;
break;
        }
    }
if (n == 1)
    {
printf("1 is neither a prime nor a composite number.");
    }
else
    {
if (flag == 0)
printf("%d is a prime number.", n);
else
printf("%d is not a prime number.", n);
    }
return 0;
}
```

4. *Check whether given year is a leap year or not.*

```
#include <stdio.h>
int main()
{
int year;
printf("Enter a year: ");
scanf("%d",&year);
if(year%4 == 0)
   {
if( year%100 == 0)
      {
         // year is divisible by 400, hence the year is a leap year
if ( year%400 == 0)
printf("%d is a leap year.", year);
else
printf("%d is not a leap year.", year);
      }
else
printf("%d is a leap year.", year );
   }
else
printf("%d is not a leap year.", year);
return 0;
}
```

5. *Simple calculator using swich case.*

```
// Program to create a simple calculator
#include <stdio.h>
int main() {
char operator;
doublefirstNumber,secondNumber;
printf("Enter an operator (+, -, *, /): ");
scanf("%c", &operator);
printf("Enter two operands: ");
scanf("%lf %lf",&firstNumber, &secondNumber);
switch(operator)
   {
case '+':
         printf("%.1lf + %.1lf = %.1lf",firstNumber, secondNumber,
         firstNumber+secondNumber);
break;
case '-':
         printf("%.1lf - %.1lf = %.1lf",firstNumber, secondNumber, firstNumber-
         secondNumber);
break;
case '*':
```

```
            printf("%.1lf * %.1lf = %.1lf",firstNumber, secondNumber,
            firstNumber*secondNumber);
    break;
    case '/':
            printf("%.1lf / %.1lf = %.1lf",firstNumber, secondNumber,
            firstNumber/secondNumber);
    break;
        // operator is doesn't match any case constant (+, -, *, /)
    default:
    printf("Error! operator is not correct");
       }
    return 0;
    }
```

Code Snippets

Find the output of the following code snippets:

1.
```
#include <stdio.h>
main()
{
        int a=4;
        printf("\n %d",10 + a++);
        printf("\n %d",10 + ++a);
}
```

2.
```
#include <stdio.h>
int main()
{
        int a=4,b=6;
        printf("\n %d",(a>b)?a:b);
}
```

3.
```
#include <stdio.h>
main()
{
        int a=4,b=6,res;
        res=a+++b;
        printf("\n %d",res);
}
```

4.
```
#include <stdio.h>
main()
{
        int a;
        float b;
        printf("\n enter a four digit number");
        scanf("%2d",&a);
        printf("\n enter a floating point number");
        scanf("%f",&b);
        printf("\n The numbers are %d and %f ",a,b);
}
```

5.
```
#include <stdio.h>
main()
{
        intch;
        printf("enter a character");
        scanf("%c",&ch);
        printf("\n The character is %d",ch);
}
```

Programming Exercises

1. Write a program that prints floating point values with the following specifications:
    a. Correct to two decimal places and
    b. Correct to four decimal places
2. Write a program to read 10 integers and print the sum of odd and even numbers separately.
3. Write a program to count vowels in a text.
4. Write a program to print factorial of a given number.
5. Write a program to print employee pay slip with the following information:
    a. Name
    b. Employee id
    c. Date of birth
    d. Basic
    e. Salary (given HRA=10% and TA=5% of the basic)