PES UNIVERSITY

Department of Computer Science and Engineering

Problem Solving With C - UE20CS151 Lecture Notes

Acknowledgement

I would like to sincerely thank the previous anchors of “**Problem Solving with C**” course, **Prof. N S Kumar, Dr. Shylaja S S and Prof. Nitin V Pujari** in 2017, 2016 and 2015 respectively for sharing all the C materials and guiding me in making these Lecture notes. This material would not have been possible without their guidance.

I would also like to **express my special thanks to present C team** for their valuable suggestions and advice. Thanks to **each and every one** who has directly or indirectly helped me while making this material.

Sindhu R Pai
Asst. Prof
Dept. of CSE
Anchor of PSWC
UE20CS151
April-June,
2021
PES University

Unit – 1: Counting

Objectives:

- To make students understand the given counting problem and apply theconstructs of C to solve the problem.
- To make students knowledgeable on C standards, Life cycle of a C Program andintroduce them to Program structure of C Language.
- To make students learn and appreciate the concept of operators and control structures

Outcome: At the end of the Unit, students will be able

- To solve the given counting problem using C Language
- To apply C standards to compile the code and recreate the phases involved in C program development Life Cycle.
- To apply control structures and operators in their application and execute the C code.

Let us answer few questions before we start with Problem Solving with C.

Q1. What is a Programming Language?

- Formal computer language or constructed language designed to communicate instructions to a machine, particularly a computer (wiki definition)
- A way of telling a machine what operations to perform
- Can be used to create programs to control the behaviour of a machine or to express algorithms

Q2. Why so many Programming Languages?

To choose the right language for a given problem. Example Domains are listed – Web Browsers, Social Networks, Image Viewer, Facebook.com, Bing, Google.com, Games, Various Operating Systems

Q3. What are the different Levels of Programming Language?

- Low Level
  - Binary codes which CPU executes
  - Programmer's responsibility is more
  - Machine Language
- Middle Level
  - Offers basic data structure and array definition, but the programmers should take care of the operations
  - C and C++
- High Level
  - Programmer concentrates on the algorithm and programming itself
  - Java , Python, Pascal

Q4. What is the meaning of Paradigm? – Style of Programming

Brief info on each of them.

- Imperative
  - First do this and next do that

  - FORTRAN, Algol, COBOL, Pascal, …
- Structured programming
  - Single entry and single exit. Avoid GOTO
  - C, …
- Procedural
  - Subset of imperative
  - Use of subroutines
  - C, …
- Declarative
  - Declares a set of rules about what outputs should result from which inputs
  - Lisp, SQL, …
- Functional
  - Function should be the first-class citizen. No side effects. Assign Function name to another, pass function name as an argument, return function itself
  - If and recursion. No loops
  - Scheme, Haskell, Miranda and JavaScript, …
- Logical
  - Uses predicates
  - Extraction of knowledge from basic facts and relations
  - ASP, Prolog and Datalog, …
- Object Oriented
  - Data needs protection
  - Java, C++, Python , …

Q5: Which Languages are used while Developing Whatsapp? Think.. !

Erlang, JqGrid, Libphonenum, LightOpenId, PHP5, Yaws and many more…

Q6. Why should one learn C? What are the advantages of 'C'? Is not 'C' an outdated language?

We have to fill our stomach every day 3 or 4 times so that our brain and body get enough energy to function. How about eating Vidyarthi Bhavan Dosa every day? What about Fridays when the eatery is closed? Why not buy Dosa batter from some nearby shop? Or do you prefer to make the batter yourself? Would you have time to do that? Would that depend on how deep your pockets are? Would you like to decrease your medical bills?

Every language has a philosophy. The language used by poets may not be suitable for conversation. Poets use ambiguity in meaning to their advantage, and some verses in Samskrita have more than one meaning. But that will not be suitable for writing a technical report.

The goal of 'C' is efficiency. The safety is in the hands of the programmer. 'C' does very little apart from what the programmer has asked for.

Example: When we index outside the bounds of a list in Python, we get an "index error" at runtime. To support this feature, Python runtime should know the current size of a list and should also check whether the index is valid each time we index on a list. You are all very good programmers and I am sure you never get an index error. You get what you deserve. If you are lucky, the program crashes. Otherwise something subtle may happen, which later may lead to catastrophic failures.

So,

- C gives importance to efficiency
- C is not very safe; you can make your program safe
- C is not very strongly typed; mixing of types may not result in errors
- C is the language of choice for all hardware related softwares – system software

C is the language of choice for softwares like operating system, compilers, linkers, loaders, device drivers.

Q7. Is 'C' not an old language?

Yes and No.

It was designed by Dennis Ritchie – my prostrations to him –most of us working in the field of computer science owe our life to him. We use 'C' like languages and unix like operating systems both have his contribution – in 70s. But the language has evolved over a period of time. The latest ‘C’ was revised in 2011.

There is one more reason to learn ‘C’. ‘C’ is the second most popular language as of now. according to Tiobe ratings. https://www.tiobe.com/tiobe-index/. This website gives the popularity ratings of programming languages.

Chapter 1: Introduction

C is a general-purpose, high-level language that was originally developed by Dennis M. Ritchie to develop the UNIX operating system at Bell Labs. C was originally first implemented on the DEC PDP-11 computer in 1972.

1.1 History of Programming Languages

Links for reference:

https://cs.brown.edu/~adf/programming_languages.html

https://en.wikipedia.org/wiki/History_of_programming_languages

1.2 C Standards

A language should be understood the same way by compiler writers and programmers all over the world. To facilitate this, the languages are standardized by international organizations like the ISO. C has been standardized by the American National Standards Institute (ANSI) since 1989 and subsequently by the International Organization for Standardization (ISO).

The language 'C' has 4 historical land marks.

- K & R 'C' : defined by the seminal book on 'C' by Kernighan and Ritchie
- C 89
- C 99
- C 11

1.3 Facts about C

- C was invented to write an operating system called UNIX.
- C is a successor of B language which was introduced around the early 1970s.
- Applications include
  - Adobe Systems: Includes Photoshop and Illustrator
  - Mozilla: Internet Browser Firefox Uses C++.
  - Bloomberg: Provides real time financial information to investors

  - Callas Software: Supports PDF creation, optimization, updation tools andplugins

- Today C is the most widely used and popular System Programming Language.
- Most of the state-of-the-art software have been implemented using C.
- Today's most popular Linux OS and RDBMS MySQL have been written in C

1.4 Program Structure

Let us summarize the characteristics of a program in ‘C’.

- The language is case sensitive. The uppercase and lowercase are distinguished.
- The program is immune to indentation. ‘C’ follows the free format source code concept. The way of presentation of the program to the compiler has no effect on the logic of the program.

  However, we should properly indent the program so that the program becomes readable for humans. When you prepare a new dish, I am sure you will also consider how you present it to your kith and kin.
- The program has a well-defined entry point. Every program has to have an entry point called main, which is the starting point of execution.

```
int main()              // execution begins here
{
      printf(“Hello Friends”);
      return 0;
}
```

- We add comments for documentation. Two types in C

  Line comment : starts with // and ends at the end of the line

  Block comment : start with * and ends with */ and can span multiple lines.

  Comments are ignored by the pre-processor.
- A program in ‘C’ normally has one or more pre-processor directives. These start with the symbol #. One of the directives is “include”.
- #include <stdio.h>

  The above line asks the translator to find the file whose name is stdio.h at some location in our file system and read and expand it at that point in the ‘C’ code.

- A program in ‘C’ has to have a function called main. The main function returns an int. The function may take a couple of arguments – will discuss this later. A function definition has a return type, then the function name and declaration of parameters with parentheses and a block.
- The main function is invoked(called) from some external agency – like our shell. It is the duty of this function to tell the caller whether it succeeded or failed. So, by convention, main returns 0 on success and non-zero on failure.
- printf is a library function to display. The first argument is always a string. In ‘C’, string constants or literals are always enclosed in double quotes.
- A block contains a number of statements. Each statement ends in a semicolon. This symbol ; plays the role of statement terminator.

1.5 Program Development Life Cycle (PDLC)

Phases involved in PDLC are Editing, Pre-processing, Compilation, Linking, Loading and execution.

Program to be executed is first.c

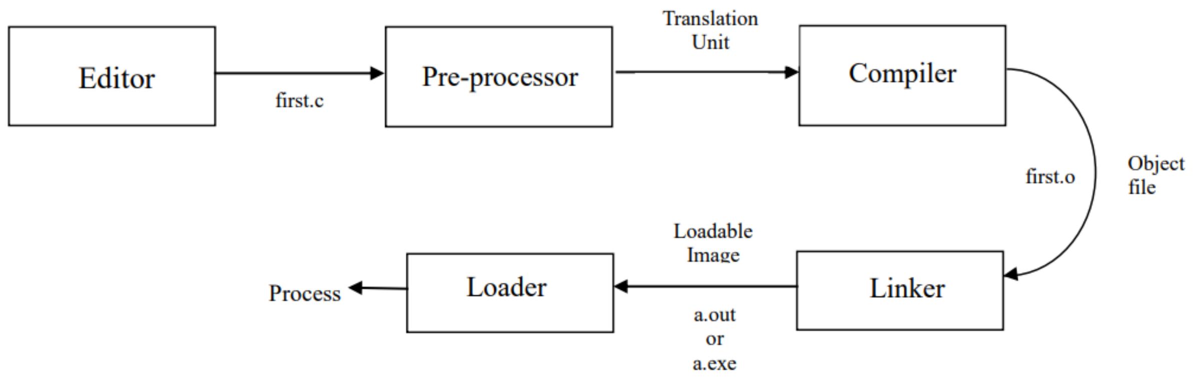


Fig 1: Phases involved in PDLC

1.6 C Program Execution in detail

Compiler used to execute the C program is gcc[GNU Compiler Collections] through Command Line Interface in Windows or Terminal in Linux based Systems.

GCC is available in Linux based systems by default.

One of the Link for GCC installation on Windows through MinGW package:

https://www.youtube.com/watch?v=sXW2VLrQ3Bs

Using C99: gcc -std=c99 program.c

Using C11: gcc -std=c11 program.c

Using __STDC_VERSION__ , we can display the standard of C in code.

The stages for a C program to become an executable are the following:

1. **Editing**:

We enter the program into the computer. This is called editing. We save the **source program**.

```
$ gedit first.c   // press enter key

#include <stdio.h>

int main()

{        // This is a comment

         //Helps to understand  the code Used for readability

         printf("Hello Friends");

         return 0;

}// Save this code
```

2. Pre-Processing:

In this stage, the following tasks are done:

a. Macro substitution–To be discussed in detail in Unit-5

b. Comments are stripped off

c. Expansion of the included files

The output is called a **translation unit or translation.**

To understand Pre-processing better, compile the above 'first.c' program using flag -E, which will print the pre-processed output to stdout.

```
$ gcc  -E first.c// No file is created. Outputof pre–processing on the standard output-terminal

......

# 846 "/usr/include/stdio.h" 3 4

# 886 "/usr/include/stdio.h" 3 4

extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__));

extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__)) ;

extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__));

......

# 916 "/usr/include/stdio.h" 3 4

# 2 "first.c" 2

int main()

{

printf("HelloFriends");

return0;

}

......
```

In the above output, you can see that the source file is now filled with lots of information, but still at the end of it we can see the lines of code written by us.

Some of the observations are as below.

- The first observation is that the comment that we wrote in our original code is not there. This proves that all the comments are stripped off.
- The second observation is that beside the line '#include' is missing and instead of that we see whole lot of code in its place. So it is safe to conclude that stdio.h has been expanded and literally included in our source file.

3. Compiling

Compiler processes statements written in a particular programming language and converts them into machine language or "code" that a computer's processor uses. The translation is **compiled**. The output is called an **object file**. There may be more than one translation. So, we mayget multiple object files.

```
gcc  -c first.c
```

This command does pre-processing and compiling. Output of this command is first.o -> **Object file.**

4. Linking

It is a computer program that takes one or more object files generated by a compiler and combines them into one, executable program. Computer programs are usually made up of multiple modules that span separate object files, each being a compiled computer program. This is the stage at which all the linking of function calls with their definitions aredone. Till stage, gcc doesn't know about the definition of functions like printf(). Until the compiler knows exactly where all of these functions are implemented, it simply uses a place-holder for the function call. The definition of printf() is resolved and the actual address of the function printf() is pluggedin. We put together all these object files along with the predefined library routines by **linking**. The output is called as **image or a loadable image**.

5. Executing the loadable image

Loader loads the image into the memory of the computer. This creates a **process** when command is issued to execute the code. We **execute or run** the process. We get some results. In 'C', we get what we deserve!

Below code is self explanatory.

```
#include<stdio.h>
int main()
{       printf("C standard is %d\n",__STDC_VERSION__);// It is a macro, will discuss in Unit -5
        return 0;
}
/*Use below commands to compile and execute the code. It doesn't work for C89
C:\Users\sindhu>gcc program.c        // By default it uses some standard.
C:\Users\sindhu>a
C standard is 201112                 // In my system, it is C11
C:\Users\sindhu>gcc -std=c99 program.c
C:\Users\sindhu>a
C standard is 199901
C:\Users\sindhu>
```

1.7 Commands to execute a C Program using gcc

Step1: gcc -c 1_intro.c

outputs object file---> intro.o

Step2: gcc 1_intro.o

outputs loadable image ---> a.out or a.exe

Step3: ./a.out and press enter in ubutnu. you get what you deserve!

a.exe and press enter in windows, you get what you deserve!

Note: You can use -o option to create the loadable image with your own name.

```
gcc 1_intro.o  -o  myName           // press enter
./myName      OR  myName            // press enter to see the output. Based on which OS
```

1.8 Errors in C

Error is an illegal operation performed by the programmer which results in abnormal working of the program. Error is also known as bugs in the world of programming that may occur unwillingly which prevent the program to compile and run correctly as per the expectation of the programmer.

Compile Time Error

Link Time Error

Run time Error

Logical Error

Compile time Error

Deviation from the rules of the Language or violatingthe rules of the language results in CT Error. Whenever there is Compile Time Error, object file(.o file) will not be created. Hence linking is not possible.

```
#include<stdio.h>
int main()
{
        printf("hello friends);//" missing at the end
        printf("%d",12/2)// ; missing at the end of statement
        return 0;
}
```

Link time Error

Errors that occur during the linking stage of a C program and this usually happens due to mismatch between Function call and function definitions. When you compile the code, object file gets created. Sometimes with a warning. During linking, definition of Printf is not available in included file. Hence LinkTime Error. No loadable object will be created due to this error.

```
#include<stdio.h>
int main()
{       Printf("hello friends");// Pis capital
```

```
        printf("%d",12/2);
        return 0;
}
```

Run time Error

Errors that occur during the execution of a C program and generally occur due to some illegal operation performed in the program. When the code is compiled, object file(.o file) will be created. Then during linking, executable will be created. When you run a loadable image, Code terminates throwing an error.

Examples of some illegal operations that may produce runtime errors are:

Dividing a number by zero

Trying to open afile whichis not created

Lack of free memory space

```
#include<stdio.h>
int main()
{
        printf("hello friends");
        printf("%d",12/0);              // Observe 12/0
        printf("bye friends");
        return 0;
}
```

Logical Error:

Occurs when the code runs completely but the expected output is not same as actual output.

Chapter 2: Basic Constructs in C

Few counting problems are listed as below.

Find the number of times a word is repeated in a given file

Count the number of different words that can be created using the given characters

Count the number of hosts in a network

Count the number of red pixels in a given image

What is the ratio of red:green:blue pixels in a given image

A={a,b,c} and B={y,z}. How many waysare there to create a list of length 3 where the first and second value in the list come from A and the third value in the list comes from B.

For the sets A={a,b,c} and B={x,y} above, we want to count the number of lists of length 3 where the first two elements come from A, the third element comes from B, and where no two elements in the list are the same.

How many 7-digit numbers have alternating odd-even digits and no digit repeated.

…

Problem to be solved at the end of Unit_1: Count and display the number of characters, words and lines in a user input and in a given file.

C program consists of various tokens and a **token can be any identifier -> a keyword, variable, constant, a string literal, or a symbol.** For example, the following C statement consists of five tokens.

```
printf("HelloFriends");
```

The individual tokens are −

printf

(

"HelloFriends"

)

;

2.1. Identifiers

An identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9). C does not allow few punctuation characters such as #, @ and % within identifiers. As C is a case-sensitive programming language, Manpower and manpower are two different identifiers in C.

Here are some examples of acceptable identifiers –

mohd	zara	abc	move_name	a_123
myname50	_temp	j	a23b9	retVal

2.2. Keywords

The following list shows some of the reserved words in C. The keywords are identifiers which have special meaning in C and hence cannotbe used as constants or variables.

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

2.3. Variables

This is the most important concept in all languages. Variable is a name given to a storage area that our programs can manipulate. It has **a name, a value, a location and a type**. It also has something called **life, scope, qualifiers** etc. We will discuss the second part later in the course.

Sometimes, the runtime makes variables with no names. These are called temporary variables. We cannot access them by name as we have no name in our vocabulary. A variable has a type. In ‘C’, type should be specified before a variable is ever used. We declare the variable with respect to type before using it. We cannot declare the variable more than once.

Example:

int a; double b;

The type of a variable can never change during the program execution. The type decides what sort of values this variable can take and what operations we can perform. Type is a compile time mechanism. The size of a value of a type is implementation dependent and is fixed for a particular implementation.

Variable is associated with three important terms: Declaration, Definition and Initialization.

- Declaration of a variable is for informing to the compiler the following information: name of the variable, type of value it holds and the initial value if any it takes. declaration gives details about the properties of a variable.
- Definition of a variable says where the variable gets stored. i.e., memory for the variable is allocated during the definition of the variable.

In C language definition and declaration for a variable takes place at the same time. i.e. there is no difference between declaration and definition.

If we want to only declare variables and not to define it i.e. we do not want to allocate memory, then the following declaration can be used. extern int a; // We will discuss this in later units

- Initialization: We can initialize a variable at the point of declaration. An uninitialized variable within a block has some undefined value. 'C' does not initialize any default value.

        int c = 100;

```
int d; // undefined value !!   variable can be assigned a value later in the code.
int c = 200;
int c;          // Declaration again throws an error
d = 300;
```

Assignment is considered as an expression in 'C' .

2.4. DataTypes

The amount of storage to be reserved for the specified variable.

Significance of data types are given below:

1) Memory allocation
2) Range of values allowed
3) Operations that are bound to this type
4) Type of data to be stored

Data types are categorized into **primary and non-primary (secondary) types**.

Primary ---> int,float,double,char

Secondary--> Derived and User-defined types

Derived-->Arrays

User-defined--> struct,union,enum,typedef

Data types can be extended using qualifiers – No much discussion

---> Size Qualifiers

---> Sign Qualifiers

Size of a type : Size required to represent a value of that type. Can be found using an operator called sizeof. **The size of a type depends on the implementation.** We should never conclude that the size of an int is 4 bytes. Can you answer this question –How many pages a book has? What is the radius of Dosa we get in different eateries?

The size of a type is decided on an implementation based on efficiency. C standards follow the below Rules.

sizeof(short int)<=sizeof(int)<=sizeof(long int)<=sizeof(long long int)<= sizeof(float)<=sizeof(double)<=sizeof(long double)

Range of values: If we have to know the range of values for each type of data, it is possible in C using some of the constants defined.

Max and min values of integral constants are available in limis.h. Max and min for float are available in float.h. Refer to the below code.

```
#include<stdio.h>
#include<limits.h>
#include<float.h>
int main()
{
        printf("%d %d\n",INT_MAX,INT_MIN);
        printf("%d %d\n",FLT_MAX,FLT_MIN);
        printf("%d %d\n",CHAR_MAX,CHAR_MIN);
        // Can include SHRT_MAX, UINT_MAX, LONG_MIN
        return 0;
}
```

2.5 Formatted Output and Input

For interactive output and input, printf()[formatted output] and scanf()[formatted input] functions are used.

2.5.1 Formatted Output function in C

Let us understand the output function printf in detail. The C library function printf() sends formatted output to stdout. A predefined function in "stdio.h" header file. By using this function, we can print the data or user defined message on console or monitor. **On success, it returns the number of characters successfully written on the output. On failure, a negative number is returned.**

int printf(const char *format, ...)

- printf("hello Friends");
  The first argument to printf is a string. The output depends on this string. String literals in ‘C’ are enclosed in double quotes.

- printf("hello ", "friends"); // hello

  Output depends on the first string. As there is no interpretation of thesecond parameter, it would not appear in the output.

- printf("hello %s\n", "friends\n"); // hello world

  The presence of %s in the first string makes thefunction printf look for the next parameter and interpret it as a string.

- printf("%s %s %s %s\n", "one", "two", "three", "four");

  The function **printf takes variable number of arguments**. All arguments are interpreted as the number of %s matches the number of strings following the firststring.

- printf("%s %s%s %s\n", "one", "two", "three");

  NO! we are in for trouble. There is no argument for the 4th%s in the format string. So, printf tries to interpret as a string. If we are lucky, the program will crash. We have an "**undefined behaviour**". C does no checking at runtime!

- printf("%5d and %5d is %6d\n", 20, 30, 20 + 30); // Discussed in detail:Format String

  Arguments to printf can be expressions – Then the expressions are evaluated and their values are passed as arguments.

  %d : indicates to printf to interpret the next parameter as an integer.

  %5d : tells printf to display the integer using 5 character width.

- int a = 20; int n = printf("%d\n",a);// printf executes and **returns # of characters successfully printed on the terminal**

  printf("n is %d",n); //2

- printf("what : %d\n", 2.5);

  GOD if any should help the programmer who writes such code!! undefined behaviour.

Let us digress to discuss the last statement in detail.

A few points to note about totally compiled languages like ‘C’.

- **There is no translator at runtime in ‘C’**.
- In ‘C’, type is a compile mechanism and value is a runtime mechanism. As the goal of ‘C’ is efficiency, only values are stored at runtime. There is no way to infer the type by looking at the bit pattern.
- All the code thatexecutes at runtime should be compiled and cannot be added at runtime.

Let us look at our present example.

- printf("what : %d\n", 2.5);

The function printftakes varying number of arguments. In such cases, the compiler cannot match arguments and parameters for type. So, the compiler might emit some warning (if it knows what printf expects) –but cannot throw errors.

As the compiler does not know that 2.5 should be converted to an integer, it does not do any conversion. So, the compiler puts the value of 2.5 the way it is expected to be stored –as per the standard IEEE 754. This standard uses what is called mantissa exponent format to store fractional (floating point) values.

At runtime, printf tries to interpret the bit pattern as an integer when it encounters the format %d. At runtime, no conversion can occur –we do not even know that what is stored as a bit pattern is a floating point value –Even if we know, there is no way to effect a conversion as we do not have a compiler at that point. We end up getting some undefined value.

So, in ‘C’, **you get what you deserve.**

Format string

Used for the beautification of the output. The format string is of the form

% [flags] [field_width] [.precision] conversion_character

where components in brackets [ ] are optional. The minimum requirement is % and a conversion character (e.g. %d). Below conversion characters for each type of data.

%d : dec int

%x : hex int

%o : oct int

%f : float

%c : char

%p : address

%lf : double

%s : string–[Will be discussed in detail in Unit-2]

```
float c = 2.5;
printf("%4.2f\n",c);// width.precission ,  f is type
printf("%2.1f\n",c);// data willnot be lost
printf("%-5d %d\n",16,b);//-for left justification
```

Escape sequences

Represented by two key strokes and represents one character.

\n	newline--move cursor to next line
\t	tab space
\r	carriage return --Move cursor to the beginning of that line
\a	alarm or bell
\"	double quote
\'	single quote
\\	black slash
\b	back space

```
printf("rama\nkrishna\bheema\n");          // rama
                                           krishna
                                           bheema
printf("rama\rkri\rbh");        // bhia
```

\r works differently in different platforms. We will not discuss more about this

```
printf("rama\tbheema\n");          // ramabheema
printf("\"mahabharatha\"");        // "mahabharatha"
printf("rama\b\bbheema");          // rabheema
```

2.5.2 Formatted Input function in C

scanf like printf takes a format string as the first argument. **The input should match this string.**

int scanf(const char *format, ...);

where

int (integer) is the return type

format is a stringthatcontains the type specifier(s).

"..." (ellipsis) indicates that thefunction accepts a variable number of arguments; each argument must be amemory address where the converted result is written to. On success, the function writesthe result into the arguments passed.

- scanf("%d%d", &var1,&var2);// & -address operator is compulsory in scanf for all primary types

  When the user types, 23 11, 23 is written to var1 and 11 is to var2.

- scanf("%d,%d", &a, &b);

  scanf has a comma between two format specifiers, the input should have a comma between a pair of integers.

- scanf("%d%d\n", &a, &b);

It is not a good practice to have any kind of escape sequence in scanf. In the above code, it expects two integers. Ex: 20 30. Then if you press enter key, scanf does not terminate. You need to give some character other than white space.

The function returns the following value:

>0 —The number of items converted and assigned successfully.

0 —No item was assigned.

<0 —Read error encountered orend-of-file (EOF) reached before any assignment was made.

- n = scanf("%d",&a);// If user enters 20, a becomes 20 and 1 is returned by the function.
- n = scanf("%d,%d",&a,&b); //If user enters 20 30, a becomes 20, value of b is undefined and 1 is returned by the function.

2.6 Character Inputand Output–Unformatted functions

A character in programming refers to a single symbol. The characters in ‘C’ have no relationship with characters in the plays of Shakespeare even though those characters utter characters we could process in programming! A character in ‘C’ is like a very small integer having one of the 256 possible values. It occupies a single byte. We code English alphabets A as 65, B as 66, Z as 90, a as 97 and so on in a coding scheme called ASCII.

To read a character from the keyboard, we could usescanf(“%c”, &x);

We could also use x = getchar();

We prefer the second as it is developed only for handling a single char and therefore more efficient even though it is not generic.

Similarly we prefer putchar(x) over printf(“%c”, x);

This code shows howto read two characters and display them.

```
int main()
{
        char ch = 'p'; // ch is a variable of type char and it can store only one character at a time.
        //Value for a character variable must be within single quote. 'p'
        printf("Ch is %c\n",ch);// p
        //char ch1 = 'pqrs';// TERRIBLE CODE
        printf("Ch is %c\n",ch);// s
        /*
        char x; char y;
        scanf("%c", &x);
        scanf("%c", &y);
        printf("x:%c y:%c\n", x, y);
        */
        x = getchar(); y = getchar();
        putchar(x);putchar(y); printf("%d", y);
        return 0;
}
// If I enter P<newline>, x becomes P and y becomes newline
```

// scanf and printf : generic; not simple

// getchar and putchar : not generic; read /write a char; simple

// if I enter p<space>q, q will not be stored in y. Only space is stored. This can be avoided using fflush(stdin) functionbetween two getcharfunction calls. This function clears the key board buffer. fflush(stdin) -windows

//__fpurge(stdin) -ubuntu. include <stdio_ext.h> in Linux based systems.

Let us try to read a line and display.

```
char ch;
while(ch != '\n')
{
        ch = getchar();
        putchar(ch);
}
```

This is a terrible code. The first time we enter the loop, ch is not initialized. If you are lucky, the loop is never entered!

In these cases, we require reading before the loop and reading atthe end of the body of the loop.

```
ch = getchar();
while(ch != '\n')
{
        putchar(ch);
}
```

Infinite loop unless the first character is new line!!

```
ch = getchar();
while(ch != '\n')
{
        putchar(ch);
        ch = getchar();
}
```

This is fine. But we can as well use assignment expression in the while.This is ‘C’ code!!

```
while( (ch = getchar()) != '\n')
{
        putchar(ch);
}
putchar('\n');
```

A few points to observe. The input from the file or the keyboard is stored in something called the buffer. That is transferred to getchar only after enter key is pressed.

Chapter 3: Operators

An operator is a symbol that tellsthe compiler to perform specific mathematical or logical functions. Operator is a symbol used for calculations or evaluations

3.1. Classification of operators

Based on the number of operands.

1)Unary	+, -, ++, --, sizeof ,&, *, …
2) Binary	+, -, *, /, %
3) Ternary	?:

Based on the operation on operands. Few ofthese are self-explanatory.We will see few others.

1) Arithmetic Operators

    +, -, *, /,%

2) Increment and Decrement Operators

    ++, --

3) Relational

    ==, !=, >=, >, <=, <

4) Logical

    &&, ||, !

5) Bitwise

    &, |, ^, >>, <<

6) Address Operatorand Dereferencing Operator

    &, *

7) Assignment

    =

8) Short-hand

    +=, -=, *=, /=, %=

9) miscoperators

    sizeof

1) Arithmetic operators

There is no exponentiation operator in C.

All are binary operands –requires two operands.

+ and –can also be used as unary operators –in such case, the operator is prefixed –appears to the left of the operand.

If the operands are int for /, the result is int quotientobtained truncation of the result. The operator % is defined on integral types only.

If the operands are mixed types(int and double) in operations + -* /, the result is of type double.

2) Increment and Decrement Operators

There are two types of Increment (++) and Decrement (--) Operators.

1. **Pre -increment and Pre-decrement operators:**First incrementor decrement the value and then use it in the expression.
2. **Post -increment and Post -decrement operators:**First use the value in the expression and then incrementor decrement.

When ++ and --are used in stand-alone expression, pre and post increment and decrement doesn't make difference. Refer the below code which demonstrate stand-alone expressions

```
#include<stdio.h>
int main()
{       int i=5;
        int j=5;
        printf("Beginning i value is %d\n",i);          //5
        printf("Beginning j value is %d\n",i);          //5
        ++i;// i++;
        printf("Later i value is %d\n",i);                      //6
        j--;// --j;
        printf("Later j value is %d\n",j);                      //4
        return 0;
}
```

Refer the below code which demonstrates the usage of ++ and --in an expression.

```
#include<stdio.h>
int main()
{
        int a=34; int b;
        printf("Beginning a value is %d\n",a);// 34
        b=a++; // b is 34 and a is incrementedto 35
        //b=++a;// First increment and then use the value in the expression
        // b is 35 and a is 35
        printf("b is %d and a is %d\n",b,a);
        // same works for pre and post decrement operators
}
```

3) Relational operators

<> <= >= == !=

The result of relational operation is 1or 0 –1stands for true and 0 for false.

The relational operators should not be cascaded –not logically supportedin C.

5 == 5 == 5 is false !!

5 == 5 == 5 becomes 1 == 5 ; this becomes0 !!

printf(“%d”, 5 == 5 == 5); // results in 0

4) Logical operators

! stands for not

&& standsfor and

|| stands for or.

In ‘C’, 0 is false and any non-zero value is true.

C follows short circuit evaluation.Evaluation takes place left to right and evaluation stops as soon as thetruth or falsehood of the expression is determined.

&& and || are sequence points–Explained in detailin the later part

5) Bitwise operators

<< >> & | ^ ~

The first 5 are binaryoperators and the last one is unary operator.

These operators are used on unsigned int only.

Common bitwise operations:

* multiply variable n by 2

n << 1

* check whether n is odd

n & 1

* swap two variables a b

a = a ^ b;

b = a ^ b;

a = a ^ b;

* check whether ith bit in variable n is 1

n & (1 << I) : 0 impliesnot set and non 0 implies set

* set the ith bit in n

n = n | 1 << i;

* clear the ith bit in n

n = n & ~(1 << I)

6) Address Operator and Dereferencing Operator

Address Operator - &

Dereferencing Operator - *

This block of code also describe Lvalue and Rvalue concept:

```
int a = 10;
int *p= &a;
printf("pointer to a : %p value \n", p, *p); // hex number 10
*p = 30;
printf("a : %d\n", *p); // a: 30
int b = *p;
printf("b: %d\n",b);// b: 30
```

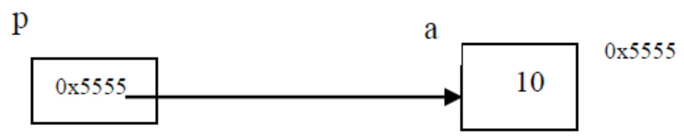
```
int b = 20;
p = &b;
printf("pointer to b : %p value \n", p, *p); // hex number 20
```

The variable a is of int type and (int *p;) p is a pointer to int. We always read a declaration right to left. A pointer variable of some particular type can hold the address of a variable of the same type.

& : in unary form –gives the address of the variable.

p = &a; assigns the address of a to p.

If we assume the address of a as0x5555, then &a is stored in p as p is a pointer to integer variable. Refer the below diagram.



*p ---> value at p ---> value at 0x5555 ----> 10 ----> a

We can get back a through p by using * : in unary form -dereferencing operator.

*p is same as a.

We can change a by assigning to *p.

*p = 30;

We can also change p itself by assigning address of b to p.

p = &b;

We talk about l-value and r-value with respect to assignment operator =.

r-value refers to whatever thatcould occur to the right of assignment operator.

L-value refers to whatever that could occur to the left of assignment operator.

A constant has only r-value. An initialized variable has both l and r value. An expression of the form a + b is only a r-value. The only operator which gives a value back is the dereferencing operator *.

3.2 Evaluation of an expression

An expression consists of operands and of operators. How is an expression evaluated? There are two parts in it.

1. Evaluation of operands:

This is fetching the operands from the memory of the computer to the registers of the CPU. All the evaluations happen within the CPU. This order is not defined.

The idea is to allow the compiler to optimize fetching of the operands. The compiler may generate code such a way that if an operand is used more than once, it may get fetched only once.

2. Evaluation of operators:

This follows the rules of precedence and if more than one operator has the same level of precedence, follows the rules of association.

Link for reference:http://web.cse.ohio-state.edu/~babic.1/COperatorPrecedenceTable.pdf

Does the first rule have any effect on our programs? Yes. If affects our programs if the expressions weuse are not pure and have side effects.

All assignments have side effects.

Let us consider an example.

```
a = 10;
b = a * a++;
```

The value of a++ is the old value of a and therefore 10. What is the value of the leftmost a? Is the value evaluated before incrementing or after incrementing? It depends on the compiler and the compiler options. So value of leftmost a is undefined. So, value of b is undefined.

```
//     10 * 10 => 100
//     11 * 10 => 110
//undefined; depends on the order of evaluation of operands
// bad code
```

3.3 Sequence Point Operation

How do we write safe and proper code in 'C' if operators have side effects and the variable values are not definedin such cases? The language 'C' also specifies points in thecode beyond which all these effects will definitely be complete. These are called sequence points.

```
// a = 10;
// a + a++ undefined
```

// to support short ckt evaluation, && becomes a sequence point

// expression before &&, || will be completely evaluated.

//a++ == 10 && a == 11 // ok; will be true 1

a + a++ is undefined as the value of left a is not defined.

a++ == 10 && a == 11 will be true.

The value a++ is 10 and a becomes 11 before the evaluation moves across the sequence point &&.

3.4. Ternary/Conditional operator

?:   Conditional Operator

Requires three operands.

E1 ? E2 : E3

where E1, E2 and E3 are expressions.

The expression E1 is first evaluated. If it is true, then the value of the expression is E2 else it is E3.

? of this expression acts like a sequence point.

Below code is self-explanatory.

```
int a= 10; int b = 20;
printf("%d\n",(a>b)?a:b);       //20
(a>b)? printf("%d",a):printf("%d",b);//20
```

Chapter 4: Control Structures

C programming language supports a few looping and a few selection structures. It may also support some unconventional control structures.

1)**Looping** structure:

-while statement

-for statement

-do … while statement

2) **Selection** structure:

-if statement

-switch statement

4.1. Looping

Let us start the discussion with the while statement.The following statement captures the structure of the while statement.

The while statement starts with the keyword while and is followed by an expression in parentheses and is followed by a statement or a block.

The expression of the while is evaluated. It could be non-zero or zero. If it is non-zero, the body of the while is executed and again the control comes backto the expression of the while. If the expression of the while is false, the loop is exited.

It is a good programming practice to always use a block even if only one statement need be in the loop.

The body of the loop is executed 0 or more times as the condition is checked top tested.

```
/<while stat>::= while (<expr>) [<stat>|<block>]
// expr : 0 is false; not 0 is true
//      no data structure
//      no indentation
//      body: single statement;
//              # of statements : grouped under { }
```

//**top testing;** execute body 0 or more times; conditional looping structure

```
// version 1
        int n = 5;
        while(n)
                printf("%d ", n);
                n= n –1;
```

This is an infinite loop as n remains to be 5 –is always true!

```
// version 2
// terrible
        int n = 5;
        while(n){printf("%d ", n);
                n = n –1;}
```

This would display 5 4 3 2 1. But the program layout does not indicate the logic structure of the program. Always indent the code to indicate the logic structure.

```
// version 3
// indent your program
// always use a block
        int n = 5;
        while(n)
        {
                printf("%d ", n);
                n = n -1;
        }
// output : 5 4 3 2 1This is nice.
```

```
// version 4
int n = 5;
while(n)
{         printf("%d ", n--); // 5 4 3 2 1
}
```

Observe the value of n--is the old value of n. When n is 1, the value of n becomes 0but the decrement operator returns 1.

```
// version 5
        int n = 5;
        while(n)
        {
                printf("%d ", --n); //  4 3 2 1 0
        }
```

Here, pre-decrement operator returns the changed value.

```
// version 6
        int n = 5;
        while(n--)
        {
                printf("%d ", n); //  4 3 2 1 0
        }
```

Observe that the old value of n is used to check whether the conditionis true or false and by the time, the body of the loop is entered, n would have the decrementedvalue. The expression of while acts like a sequence point.

```
//version 7
        int n = 5;
        while(--n)
        {
                printf("%d ", n); //  4 3 2 1
        }
```

In this case, the decremented value is checked for the truth of the while expression.

Comparethe last two cases. Inthe first case, the loop is executed n times and in the secondthe loop is executed n –1 times. In the first case, the value of n is -1 and in the second case, it is 0.

// version 8

```
int n = 5;
int f = 1;
while(n--)
{
        f *= n;
}
printf("what : %d\n", f);
```

The loop is executed n times. Each time, we multiply f with n. Would this find n power n? Or is it n!? You find out it yourself.

Our next requirement is to find the greatest common divisor(GCD) of given two numbers. How do we proceed to solve this problem?
Let us make our first attempt. Clearly the greatest common divisor of two numbers cannot be greater than the smaller of the two numbers.

Let us start our search with the smaller of the two numbers.

```
factor =(a < b) ? a :b;
```

We shall check if the factor divides both. If it does, our search is over. Otherwise, we shall decrease factor by 1 and try again.

```
while (! (a % factor == 0 && b % factor == 0))
{
        --factor;
}
```

A few questions to think.

- Can we rewrite the expression of the while using DeMorgan Theorem?
- Is this loop guaranteed to terminate?
- How many times will the loop execute if the numbers are relatively prime?
- Can we make the program more efficient?

Instead of decreasing factor by 1, can we conceptually decrease by a bigger size. This algorithm is called the Euclid's algorithm –supposed to be the oldest algorithm ever known.

Let us have a look at this program.

```
while (a != b)
{
        if(a > b)
        {
                a -= b;
        }
        else
        {
                b -= a;
        }
}
```

This algorithm states that if the two numbers are equal, that number itself is the GCD. Otherwise subtract the smaller from the bigger and repeat the exercise. This would definitely converge tothe GCD faster than the earlier algorithm.

In this program, we are also using if statement. If is followed by an expression within parentheses and then a statement or a block and then optionally followed by else and then a statement or a block.

If the expression of if is true we execute the block following if, otherwise execute the else part ifany. It is always preferred to use a block even if there is a single statement in theif and else portions.

```
// selection:
//<if stat> ::= if (<expr>) <stat>|<block>
//<if stat>::= if (<expr>) <stat>|<block> else <stat>|<block>
```

Division is repeated subtraction. Can we make this algorithm faster by using division in stead of subtraction. Here is the next version.

```
        rem = a % b;
        while(rem != 0)
```

```
{
        a = b;
        b = rem;
        rem = a % b;
}
```

Divide a by b. If the remainder is 0, then b is the GCD. Otherwise, replace a by b and b by the remainder and repeat.

A point for you to think. Would this work if b is greater than a?

Observe one other point in this code. We have the statement rem =a % b; repeated before the loop and at the end of the body of the loop. This acts like a guard.Can we avoid repeating the code? If for some reason, we changethe code, we should remember to change at both the places.

Here the 'C' way of writing the code.

```
while(rem = a % b)
{
        a = b;
        b = rem;
        rem = a % b;
}
```

We can have assignment expression as the expression of the loop. The loop is exited when rem becomes 0.

Our next requirement is to find the sum of numbers from 1 to n.

We can use the followingwhile loop.

// n is a variable given small value.

```
int i = 1;
int sum = 0;
while(i <= n)
{
        sum += i++;
}
```

This looping construct has clearly some initialization, some processing and some modification at the end of the loop. In such cases, we may wantto use the for statement.

```
// <for stmt>:: for(e1; e2; e3) <block>|<stmt>
// e1, e2, e3 : expressions
// e1 : initialization
// e2 : condition
// e3 : modification
```

For loop in Detail:

e1 is executed once. e2 is checked for condition. If it is non-zero, execute the body of the loop and thenexecute e3. Again,check for e2and thus the process continues till e2 results in Zero Value. Once e2 isZero, come out of the loop and executethe statement outside the body of the loop.

The semantics of the for statement is same as the while statement given below.

```
e1;
while(e2)
{
        <block>|<stmt>
        e3;
}
```

Few questions toAnswer:

- Can you have initialization outside the for loop?
- Can you skip condition in for loop?
- Can you have only ;; in for loop?

What this code does? Why it is wrong?

```
        int sum = 0;
        // wrong code
        for(int i = 1; i <= n; sum += i)
        {
                ++i;
        }
```

Observe closely. You will find that the summation starts from 2 and not from 1. We will also end up adding n + 1. Check the way a for statement executes.

This is the corrected code.

```
        int sum = 0;
        for(int i = 1; i <= n; ++i)
        {
                sum += i;
        }
```

‘C’ for statement is same as the ‘C’ while statement –we say they are isomorphic. It is a question of choice and style while writing programs in ‘C’.

Next our requirement is to find what is called the digital root of a given number. The idea is to add the digits of a given number. Ifthat sum exceeds a single digit, repeat the operation untilthe sum becomes a single digit. Thisis useful in what is called as parity checking.

```
        for(s = 0; n; n /= 10)
        {
                s += n % 10;
        }
```

This loop finds the sum of digits of n. The result is in s and n would have become 0. If s exceeds 9, we would like to copy s to n, repeat this operation.

Can we put this code under a while(s > 9) { <this code> }?
The answer is a clear NO as the value of s is not initialized until we enter the inner loop? Shall we initialize to 0? Then the loop itself is not entered. How about making s 10? Looks very unnatural.
Can you realize here that the sum becomes available only after doing the summation once? We require in this case, a bottom tested looping structure which executes the code at least once.
Here is the solution.

```
        do
        {
                for(s = 0; n; n /= 10)
                {
                        s +=n % 10;
                }
                n = s;
}while(s > 9);// ; at the end of do-while is compulsory
```

4.2 Selection

Let us turn our attention to the selection structure. We would to classify triangles given the 3 sides (which do form a triangle) as equilateral, isosceles or scalene. This is one possible solution.

```
int count = 0;
scanf("%d%d %d", &a, &b, &c);
if(a == b) ++count;
if(b == c) ++count;
if(c == a) ++count;
if(count == 0) printf("scalene\n");
if(count == 3) printf("equi\n");
if(count == 1) printf("iso\n");
```

Compare every pair of sides and increment the count each time –initialized to 0 on start.

Can the count be 2?

Observe a few points here.

- We are comparing integers (integral values)
- We are comparing a variable(an expression) with constants
- We are comparing for equality (no > or <)
- In all comparisons, the same variable is used.

In such cases, we may use switch statement.

```
switch(count)
{
        case 0: printf("scalene\n"); break;
        case 3: printf("equi\n"); break;
        case 1: printf("iso\n"); break;
}
```

The value of count is compared with case clauses. This comparison decides the entry into the switch and not exit from there. To avoid the code fall through,we use break statement. The rule of ‘C’ is “break if switch”.

We can also use default to capture whenall other comparisons fail.

```
    switch(count)
    {
        case 0: printf("scalene\n"); break;
        case 3: printf("equi\n"); break;
        default : printf("iso\n"); break;
    }
```

These switch statements are more efficient compared normal if statements. Note that not all nested if statements can become switch statements.

4.3 Nested control structures

We may have loops and selection nested. Here is an example to generate all Armstrong numbers between 0 and 999. The sum of cubes of digits is same as the number.

In this example,we have a loop on hundredth digit, a loop on tenth digit and a loop on unit digit. We cube the digits, find the sum, form the number using these digits and compare the sum with the number formed.

This program shows how not to write programs.

```
for(h = 0; h < 10; ++h)
{
    for(t = 0; t < 10; ++t)
    {
        for(u = 0; u < 10; ++u)
```

```
                {
                        hc = h * h * h;
                        tc = t * t * t;
                        uc = u * u * u;
                        n = h * 100 +t * 10 + u;
                        s = hc + tc + uc;
                        if(n == s)
                        {
                                printf("%d\n", n);
                        }
                }
        }
}
```

Think how many multiplications do we do for cubing? Should we repeatedly evaluate cube of t in the inner loop when it is not changing. Rearrange the code. Put the statements in the right blocks. You will be able to reduce the number of multiplications for cubing to approximately 1/3rd of what it is now.

Chapter 5: Language Specifications/ Behaviors

A Language specification is a documentation that defines a programming language so that users and implementers can agree on what programs in that language mean. Typically detailed and formal, and primarily used by implementers referring to them in case of ambiguity. Specification Can take several forms:

An explicit definition of the syntax and semantics of the language.

A description of the behavior of a "translator" for the language

"Model implementation" is a program that implements all requirements from a corresponding specification

5.1 Standardization of a Language

Type of standards

1. **de Facto:** Practices that are legally recognized, regardless of whether the practice exists in reality.
2. **de Jure:** Describes situations that exist in reality, even if not legally recognized

Language may have one or more implementations which acts as deFacto Language may be implemented and then specified, or vice versa or together.

- Languages can exist and be popular for decades without a specification - Perl
- After 20 years of usage, specification for PHP in 2014
- ALGOL 68 : First (and possibly one of the last) major language for which a full formal definition was made before it was implemented

5.2 Behaviors defined by C Standards

There are four specific behaviours defined.

1. Locale-specific behavior - Not discussed here
2. Unspecified behavior
3. Implementation-defined behavior
4. Undefined behavior

Undefined Behavior

- The result of executing a program whose behavior is prescribed to be unpredictable in the language specification to which the computer code adheres.
- It is the name of a list of conditions that the program must not meet.
- Examples: Memory access outside of array bounds, Signed integer overflow
- Standard imposes no requirements: May fail to compile, may crash, may generate incorrect results, may fortunately do what exactly programmer intended

Below code is self explanatory.

```
#include<stdio.h>
int main()
{
        //printf("%d\n",5/0);   // Undefined behavior
        float a = 23.5;
        //int b;
        //b = a/0;
        //printf("a is %d\n",a);// Undefined behavior
        //printf("b is %d\n",b);
        //printf("value is %s\n","50"); // proper output
        //printf("value is %s\n",50);
        //printf("%u %u %u\n",sizeof(int),sizeof(short int),sizeof(float));    //          Implementation
defined behavior
        int d = 10;
        printf("%d %d %d\n", d++, d++ - --d,--d);    // Unspecified behavior
        return 0;
}
```

Chapter N: Solution for the Problem

We are required to find the number of characters, number of words and number of lines in an user input and in a given file. We want to simulate a program in Unix command called wc – word count .

We have not learnt yet how to play with files in our programs. It is a bit too early in this course. We do know how toread from the keyboard. Can the operating system open a file for me and make it logically available on the keyboard? It can. This concept is called input redirection. When we run our command cmd from the unix shell and specify filename on the same line, our program cmd reads from the file whenever it reads from the keyboard.

input redirection:

```
$ cmd <filename
```

It is similarly possible to collect the output of the program which would appear on the screen in a file using output redirection.

output redirection:

```
$ cmd >filename
```

To solve this problem stated, we should know

a) how to read and display a character?-Discussed in Character I/O

b) how to read a line?–Discussed in Character I/O

c) how to make out when we reach the end of file?

d) how to break a given sequence of characters into words?

Our hurdle is to read the whole file. How do we know we have reached the end of file. ‘C’ handles this in a very interesting way. When the end of file is reached –which the operating system can make out –a particular value gets is returned by getchar. This value is decided by ‘C’ and not by the operating system. This value is associated with the name EOF. So, we keep reading until getchar returns EOF.

```
while( (ch = getchar()) != EOF)
{
        putchar(ch);
}
```

Few points to think before we start with the solution.

- It is always preferred to read at only one place.
- If you read input at two places, code could work. But what if we reach EOF in the inner loop?
- It It is always preferred to have a while with an if over while within a while.
- Can we use a boolean variable to indicate whether we are in a word or not. If we are in a word and we reach a space, the word ends and we count. If we are not in a word and we encounter a space, we ignore it. We set in_word when we encounter a non-white-space.

```
#include<stdio.h>
int main()
{
        int nw = 0;
        int nc = 0;
        int nl = 0;
        char ch;
        int inword = 0; // not in a word so far
        while( (ch = getchar()) != EOF)
        {
                ++nc;
                if(ch == '\n')
                {
                        ++nl;
                }
                if(inword && (ch == ' '|| ch == '\t' || ch == '\n'))
                {
                        inword = 0; ++nw;
                }
                else if (!(ch == ' '|| ch == '\t' || ch == '\n'))
                { //avoid recomputation of white space concept
                        inword = 1;
                }
        }
        printf("number of words %d number of lines %d number of characters %d\n",nw,nl,nc);
        return 0;
}
```

You may compile this and run this against any text file say myfile.txt.

```
# ./a.out < myfile.txt
```

Compare with the below command output

```
# wc myfile.txt                    // Not in windows
```

Both should give the same result.

We have tried on the collection of Shakespeare's works.

```
$ wc s.txt
124210  899680 5447737 s.txt
$ gcc program.c
$ ./a.out <s.txt
# of char : 5447737
# of words : 899680
# of lines : 124210
```

END OF UNIT–1.

HappyLearning "C".

Questions for you to think!

1. How do you print as below?

    "HELLO HOW ARE YOU?"

    %s and %d are are known as format specifiers.

    ‘MAGIC!’

2. What is the output of below code?

    printf("%d",123++);

    printf("%d",23.5%2);

3. What is the output of below code?

    printf("%f\n",(a>b)? a=a-b: b=b-a);

    printf("%x\n",17);

4. If I pre-process a non-c text file does the pre-processor give error ?
5. If I miss Pre-processor directive, does pre-processor give error ?
6. If I make Syntax Error, does pre-processor give error?
7. If I use #include and the file to beincluded is missing, does pre-processor give error ?
8. Can there be nested Comments in C-Language?
9. How to print the maximum and minimum value thatadouble variable can hold?
10. Is there a way to get Information about the sizeof the basic arithmetic types ? If Yes How ?
11. Where does thedeclarationof printfpresent ?
12. Can printf be redefined as another identifier in C ?
13. How does printf handle in case of mismatched place holders?
14. Does the printf know the type of variables in runtime ?
15. Is True and False are keywords in C?
16. What does -o option in gcc do ?
17. Can comments be inserted any where in the program?
18. Once the execution of C code is done, how to print the status of the recent process?
19. Will abc@bea valid identifier?
20. Can there be two a.out in the same directory ?