C

by Examples

A Complete Material on C-Language Programming

Since 1999

C-Family Computer Education

Flyover Pillar No:16, Service Road, Benz Circle,
Vijayawada, AP, India, pincode-520010,
9440-030405, 8500-117118.

This book softcopy is available in pdfhost.io website, to download the file, search through keyword 'cfamily', along with this book, you get 500 exercise programs list on C,C++, Java programming.
This pdf file(s) designed for 2-sided print-copy (Xerox copy).
I request you to rate the **C-Family** on Google review.

C by Examples

A complete reference material on C-language programming

About C

C-language is an ideal programming language and a perfect combination of power and flexibility. Though it is a high-level language, it has low level features as well as easy interaction with hardware and OS. Even assembly language code can be mixed in it. It is accepted as a good system programming language. Powerful OS like UNIX was developed using C.

'C' is widely used in the software industry for both system and application side development. C is the primary subject in computer science. It lays good foundation in programming and eases the learning of any advanced courses that are to follow. So, it has been added in every branch of arts, science, and engineering as part of curriculum. Every graduate student is going to face C, C++, DS, Java, Python and Oracle in the first & second academic years.

About C-Family Computers

C-Family is a premier institute in Vijayawada training the students in C, C++, DS, Java, Python, Oracle, WebDesign courses **since 1999**. C-family was not setup by any business men instead it was setup and run by a group of eminent programmers. We provide hands on training to ambitious students in right proportions to their passion for knowledge. Needless to say, taking individual care of each student is the hallmark of our institution. Our courses are designed in such a way that student get best foundation in logic and programming skills. [The languages C, C++, VC, VC++, JAVA are called C-Family Languages, as we teach all these courses, we named this institute **C-Family Computer Education**]

I am grateful to all the students, friends, staff who helped me making this material. This book is advisable to learn the logic and programming skills for beginners. For any suggestions contact me on **srihari.vijayawada@gmail.com**

by **Srihari Bezawada**

since 1999
C-Family Computer Education,
#40-9/1-26, Vasayva Complex,
Dr. Samaram Hospital Lane,
Benz Circle, Vijayawada-10,
9440-030405, 8500-117118.

Number systems and its conversion

We have mainly 4 number systems in the computer science.

1. Decimal Number System
2. Binary Number System
3. Octal Number System
4. Hexadecimal Number System

Decimal to Binary Conversion

The word decimal means 10, the base of decimal system is 10, here we have symbols 0, 1, 2, …9.
The word binary means 2, the base of binary system is 2, here we have symbols 0,1.
We know, computers follow binary number system, where, it manages the data and instructions in the form of binary numbers (0/1), and this binary numbers managed as two voltages, for example 5volts for 0, and 12volts for 1. So, computers follow binary system for storing the data and also for processing the data.
We know, humans follow decimal system, and this system is also used by the computer while showing data in visible form on the screen and printer. It uses graphical conversion to show binary values in decimal/text format (pixels/dots).

Following example shows how to convert decimal number 13 into its binary form.
Continuously divide 13 with 2 and collect the quotients and remainders, the collection of remainder forms a binary number, this is as given below.

13				Remainders
↓				↓
13	/	2	→	1
6	/	2	→	0
3	/	2	→	1
1	/	2	→	1
0				

Thus collection of these remainders from bottom-to-top forms a binary number (13_{10} → 1101_2)

Binary to Decimal Conversion

Multiply all digits(bits) of N with 2^0,2^1,2^2,2^3,2^4… from right-to-left, the sum of all such products forms a decimal number. Following example explains how to convert binary number 1101 to its decimal 13.

1	1	0	1	→	$1*2^3$	+	$1*2^2$	+	$0*2^1$	+	$1*2^0$	→ 13
2^3	2^2	2^1	2^0		8	+	4	+	0	+	1	

Example2: converting binary number 11001 to its decimal 25.

1	1	0	0	1	→	$1*2^4$	+	$1*2^3$	+	$0*2^2$	+	$0*2^2$	+	$1*2^0$	→ 25
2^4	2^3	2^2	2^1	2^0		16	+	8	+	0	+	0	+	1	

Hexa decimal System: The base of this system is 16, and we have 16 symbols from 0 to 15, the symbols are 0,1,2,3,….9,A,B,C,D,E,F (10→A, 11→B, 12→C, 13→D, 14→E, 15→F).

This system mainly used to represent binary number in simple readable form of digits & alphabets.

As we know, binary system is very difficult to follow, because everything is in 1 or 0. So we represent binary numbers in hexadecimal system for simplicity. Actually, Hexadecimal system resembles the binary values.

Here we divide a binary number into 4-bit groups, where each group contains exactly 4bits and each group is represented in hexadecimal value. Let us take a sample binary number: 1101100111000011

This binary value is divided in 4-bit as 1101-1001-1100-0011, and is taken in hexadecimal system as D9C3.

In Binary	1101	1001	1100	0011
	↓	↓	↓	↓
	13	9	12	3
	↓	↓	↓	↓
In Hexadecimal	D	9	C	3

Memory Measurements

Memory constituted in the form of bytes, where each byte consists of 8bits and the measurements are

1 bit = 1 cell → the single bit or cell can hold either 1/0
8 bits = 1 byte
1024 bytes = 1 kilo byte (1kb)
1024 kilo bytes = 1 mega byte (1mb)
1024 mega bytes = 1 giga byte (1gb)
1024 giga bytes = 1 tera byte (1tb)
1024 tera bytes = 1 peta byte (1pb)

eg1) The binary value 1101 is stored in a byte as (remember, 1 byte = 8 bits)

0	0	0	0	1	1	0	1
8th bit	7th bit	6th bit	5th bit	4th bit	3rd bit	2nd bit	1st bit

The computer automatically adds zeros in front of 1101, this is like 00001101.

eg2) The binary value 1001101 is stored in a byte as

0	1	0	0	1	1	0	1
8th bit	7th bit	6th bit	5th bit	4th bit	3rd bit	2nd bit	1st bit

Overflow error: we can't store more than 8bit value in 1byte memory, it causes over-flow error.

Here leftmost bits are omitted (truncated). eg: If we store 12bit value like 1100-1101-1001 in 1byte, then it holds only right most 8-bits 1101-1001 and the left most 4-bits 1100 are truncated.

Memory and its Min and Max value capacity

9	9	9	9
4th bit	3rd bit	2nd bit	1st bit

This value 9999 is equal to 10^4-1 in decimal system
This is maximum value that can be stored in 4bits of decimal system

1	1	1	1
4th bit	3rd bit	2nd bit	1st bit

This value 1111 is equal to 2^4-1 in binary system
This is maximum value that can be stored in 4bits of binary system

Some people raise a doubt about how the binary value 1111 is equal to 2^4-1, we already know how to convert binary value to decimal, so by converting 1111 into decimal, we get 2^4-1.

1	1	1	1
2^3	2^2	2^1	2^0

→

$1*2^3$	+	$1*2^2$	+	$1*2^1$	+	$1*2^0$
8	+	4	+	1	+	1

→ 2^4 -1 →15

Let's see $1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 \rightarrow 2^3 + 2^2 +2^1 + 2^0 \rightarrow 2^4$-1

Again we may get a doubt, how $2^3 + 2^2 +2^1 + 2^0$ is equal to 2^4-1. Following math analysis clears it

Let $X = 2^0 +2^1+2^2+2^3$

$X = 2^0 +2^1+2^2+2^3 +2^4 - 2^4$ //after adding and subtracting 2^4 to this X

$X = 2^0 + (2^1+2^2+2^3 +2^4) - 2^4$ // taking 2 as common

$X = 2^0 + 2(2^0+2^1+2^2+2^3) - 2^4$

$X = 2^0 + 2(X) - 2^4$ // since $X = 2^0 +2^1+2^2+2^3$

$X = 1+2X-2^4 \rightarrow X=2^4-1$

So in 8 bits (1 byte) memory, we can store a value in range 0 to 2^8-1 (0 to 255),
in 16 bits (2 byte) memory, we can store value 0 in range to 2^{16}-1 (0 to 65,535).

Representing –ve values

For signed types, the leftmost bit is used to represent sign (+ve/-ve). This bit is called sign bit, If this bit contained 0 then it is said to be +ve sign value, or else –ve sign value, For example,

the value +13(1101) is stored in one byte as

+ve sign							
0	0	0	0	1	1	0	1
8th bit	7th bit	6th bit	5th bit	4th bit	3rd bit	2nd bit	1st bit

The value -13 is stored as

-ve sign							
1	0	0	0	1	1	0	1
8th bit	7thbit	6th bit	5th bit	4th bit	3rd bit	2nd bit	1st bit

Note: This kind of value representation is called "signed magnitude representation".
So in 8bits (1+7 bits) memory, we can store value in the range of -2^7 to $+2^7$-1 (-128 to 127).
Actually, for negative values 2's complement method is used, here extra +1 is added to the given number, so we get -128 instead 127 in negative side. (in next topic, we will see what 2's is Complement)

Binary addition

Addition of binary values

X →	**0**	**0**	**1**	**1**
Y →	**0**	**1**	**0**	**1**
X+Y →	**0**	**1**	**1**	**10**

When we add 1+1 → we get sum as 0, and carry as 1, so result is 10
don't call this result as ten, call it as one - zero.

X→	**1**
Y→	**1**
Z→	**1**
X+Y+Z→	**1 1**

Carry	**1**	**1**	**1**	**1**	
X→		**1**	**1**	**0**	**1**
Y→		**1**	**1**	**1**	**1**
X+Y→	**1**	**1**	**1**	**1**	**0**

carry	**1**	**1**			
X→		**1**	**1**	**0**	**1**
Y→		**1**	**1**	**1**	**0**
X+Y→	**1**	**1**	**0**	**1**	**1**

Binary subtraction

In computer, subtraction is done like addition using 2's complement method, this is truly simple and fast. The normal subtraction of two values takes much time; here we may need to check & barrow from next digits. For example, subtraction of 3456 - 0999. Here first, we try to subtract last digits 6-9, but here 6<9, so we need to barrow from next digits. In this way, we have to look-up and barrow from next position digits. This process takes much time and complex. The best solution is, 2's complement method.

2's complement is → 1's complement + 1
1's complement is → reverse of a given number (replace 1 by 0, and 0 by 1)
that is, 1's complement of 1 is 0, 1's complement of 0 is 1
If X=1001, then 1's complement of X is 0110
If X=0001, then 1's complement of X is 1110

2's complement is → 1's complement + 1
If X=1001, then 2's complement of X is 0110+1 → 0111

Procedure for subtraction using 2's complement method.

1. Let X,Y are two values
2. Let Y is small
3. find 2's Complement of Y
4. Now add X + 2's Complement of Y
5. In the result of addition, remove leftmost bit
6. Hence we get subtracted value. (**note:** if X<Y then add –ve sign to the result)

1. Let X=1101 (13), Y=1001(9)
2. The 2's complement of Y is 0110+1 → 0111
3. Addition of 1101 + 0111 → 10100 (this is addition of X + 2's Complement of Y)
4. Remove left most bit from 10100, then it is → 0100 (this is the output)
5. The result of X-Y is 0100 (4)
6. This logic works for all binary values.
7. This is truly faster than traditional method.

* people often say, 2's complement values are -ve form values of given values.

What is instruction, statement and program?

Instruction is a command, that is given to the computer processor to perform certain elementary task.

For example, consider a simple addition instruction 10+20, when this instruction executes, we get 30.

Statement is a meaningful combination of one or more instructions that solves a complex task.

For example, consider calculating area of a circle **"area=22/7*radius*radius**"

Program is an organized collection of such related instructions/statements to solve a specified problem.

Software can be defined as collection of one or more programs.

The programs look like the following example; this calculates the sum of two numbers

```
step1. input(a,b)
step2. c=a+b
step3. print(c)
```

What is Programming Language?

Language means communication between two people; likewise, programming language is a communication between user and the computer. The user communicates with the computer by giving instructions to it, the instructions are written based on the language's syntax structure provided by the language manufactures. We have several languages such as Fortran, Pascal, COBOL, C, etc; out of all, C is a rich, simple, elegant, powerful, and structured language. Now a day's C became primer language for advanced languages like C++, Java, C#, VC, VC++, as they all adopted C syntax. All these languages are also called **C-Family** languages.

Evolution of languages

We have three types of languages 1.machine language, 2.Assembly language, 3.high level language. Programs are written using these languages.

A) Machine Language (binary/low level language)

It is treated as first generation language, used during development of computer in 19th century. In this language, the data and instruction of programs were written in terms of binary symbols (1&0's), even input and output were given in binary codes, as the computer understands the instructions only in ones and zeros. For example, the instruction 9+13 is in binary form like **0010** 1001 1101. Here, the first 4 bits represents the code of '+' and remaining 8 bits are 9 & 13. This language is also called binary or low-level language. Actually, this language is difficult to understand, remember and writing programs in binary codes, it leads to a lot of typing mistakes, because everything is 1&0s. This problem led to the invention of assembly language.

B) Assembly Language

this is somewhat better than binary language. Here, symbolic names were provided for every binary codes of machine language. The names such as add, sub, mul, div, cmp, etc. So, programmers can easily understand and write instructions using these names instead of binary 1&0's. For example adding 4 & 6 in 8088 assembly language as

```
mov  ax,4;
mov  bx,6;
add  ax,bx
```

However, the main drawback of assembly language is, a large set of instructions are needed to be written for simple tasks like addition of two numbers. It supports only small programs if the code is <1000 lines, otherwise difficult to manage the code. Moreover, assembly differs from one to another machine. Some people consider Assembly language also a machine language since the assembly codes resembles the machine codes.

C) High Level Language

this is English and mathematical oriented language. Here data and instructions composed in terms of English words and mathematical expressions. For example c=a+b, c=a-b , if(a<b), …etc. So one can easily learn and apply high level language code. One can write the programs without the knowledge of computer hardware and operating system, i.e, it hides(abstract) the underlying details about the instructions how they are being processed in the machine. So, programmers can easily code without bothering about hardware and O.S. The 8085, 8086 are known as Assembly languages whereas C,C++, JAVA are high level languages.

What is Assembler & Compiler?

Neither assembly nor high-level language codes can be understood by the computer directly. Before executing them on the processor, they have to be converted into processor understandable codes. The assembler is a translation-software, which translates assembly language program codes into equivalent processor understandable codes. Similarly, compiler translates high-level language code into equivalent machine(processor) understandable codes. We have several operating systems and several C compilers. All most all compilers follow the same rules except for a few modifications.

Brief History of C

In 1960's, there was a need for general purpose programming language and the available ones were of specific-purpose only. For example, COBOL was meant for business applications; similarly FORTRAN was extremely small and suitable only for scientific applications. Instead of being small, simple and specific, the CPL was intended to support wide range of applications in all sectors; thus, the demand for a general-purpose language led to the invention of CPL (Combined Programming Language). However, its heavy specific features made it difficult to program. Its drawbacks were eliminated, simplified and further developed by Martin Richards and named as BCPL (Basic CPL). Still it had some complex & difficult features of CPL and these difficulties were simplified by ken Thompson and named it as 'B'.

Later while developing UNIX operating system in 1969, ken Thomson faced several problems with the B language; here B was used for programming while making UNIX software. Dennis Ritchie modified the B language to overcome its limitations to suit all needs and renamed it as C. Of course, the development of UNIX using C language made it uniquely portable and improvable; Finally, C was released in 1972. Dennis Ritchie is a research scientist at Bell Telephone Laboratories in U.S.A. Brian Kernighan also participated while making the definitive description of the language. Therefore, the C is also referred as 'K&R-C'.

Features of C-Language

A) C is a small & compact language: It has small number of keywords and symbols. Therefore, it provides easy & compact programming. The size of software is also less compared to other language and it has good library functions.

B) C is a high-level language with low-level features: Although C is technically a high-level language, it has good syntax, features (pointer) and library functions to interact directly with hardware of the computer. C linked closely with the machine, so one can directly access the components like hard disk drive, optical drives, printers, operating system…etc; besides, it supports assembly language instructions within the C program. That is, one can mix up assembly language instructions within the C program; hence it is well suited for writing both system software and business packages.

C) C is a structured programming language: Previously, the languages were influenced by 'if', 'goto', 'repeat', etc; writing code using such statements makes the program unreadable and complex. If there are no ideal structural tools to force the programmer to write the code in uniform style then everybody writes one's own style and makes the program complex and un-readable to others. For example, if there are no predefined traffic rules in city roads, everyone travels as per one's convenience and makes traffic disorders and cause inconvenience to others; so the traffic structures such as signals, dividers, zebra lines and indicators force the traveler to follow rules in order not to cause confusion. Now, the word structured means a predefined logical model (uniform syntax) and structured programming means, writing instructions in a predefined structured manner, thereby every programmer writes instruction in uniform style and made easy to understand by others.

Structured programming was first suggested by Corrado Bohm and Guiseppe Jacopini. The two mathematicians demonstrated that any computer program can be written with just three structures: sequence, decision, and looping. ie, the logic of any program can be expressed in terms of these structures. A program of any complexity can be solved by the appropriate combinations (mixture) of these three basic constructs. These structures are constructed with one entry and one exit style, so that one can easily understand the execution flow of a program. Thus structured programming greatly reduces the complexity of programs. So every structured language supports the following constructs

- decision controls statements (if statement)
- Case selection. (switch case)
- Looping. (while , for loops)
- Subroutine (functions/sub programs)

Subroutine: this is also called function or sub-program or procedure or method. This feature is to divide a big program into several reusable segments called sub-programs, each with the all necessary data and instructions to perform a certain task. Thus, this collection of sub-programs makes the entire program.

D) C is a portable language: The word portable means easy to carry or transfer, here the portability refers to the ability of a program to run on different environments (hardware or operating systems).

As 'C' became powerful, it had provided different version of C-compilers for different operating systems. A C-program written in one platform can be portable to any other platform with few/negligible modifications. For example, a program written in UNIX operating system can be easily converted to run in WINDOWS or DOS and vice versa.

E) C has flexible coding style: Unlike other languages (COBOL, FORTRAN), C provides a freedom to the programmer while coding the program. We can write the code without bothering about the alignment of instructions. The C compiler can recognize the code even when the program is not aligned or typed properly. In other words, we can use more spaces, empty lines in between instructions (tokens), or we can type several instructions in one line.

F) Widely Acceptable: It is suitable for both system and application side programming. It frees the programmer from traditional programming limitations. It empowers the programmer to develop any kind of applications. Thus, accepted by almost all users and became the most popular language in the world. In fact, many of the software available in the market are written in C.

G) C is a case sensitive language: In C, an upper-case alphabet is never treated equal to the lower-case alphabet and vice versa. All most all keywords and predefined routines of C languages use only lower-case alphabets. Therefore, it is very simple to type in only one case. Of course, some user defined symbols (identifiers) can be typed in upper case to identify them uniquely.

Operating System

The term O.S refers to a set of programs that manages the resources of a computer. The resources of computer include processor, main-memory, disks, and other devices such as keyboard, monitor, printer that are connected to it. It also provides a good interface to the user. The interface provided by the O.S enables the user to use the computer without knowing the details of the hardware. So, interface hides the underlying working of a computer. The main jobs of O.S are memory management, process management, disk management, I/O management, security, and providing interface to the user …etc.

Currently, the widely used operating systems are MS-DOS, UNIX, and WINDOWS. DOS is a simple operating system largely used in PCs. Unix, Linux, Mac, Windows on the other hand used variety of computers such as mainframes, servers, graphics workstations, supercomputers, and also in PCs.

When a user runs a program (app) in the computer, the operating system loads the program into main memory (RAM) from the hard-disk, and then executes the instruction by instruction with the help of processor. The processor can take & execute only one instruction at a time. So, this loading and executing instructions is done under the control of OS. Thus, OS executes our programs with the help of hardware. OS is the main responsible for all these things and also makes the computer in working for other tasks.

The relation between hardware, operating system, user-program, and user can simulate with a banking-system such as bank, employee, transaction and customer. The hardware as a bank and O.S as a bank employee, who works dedicatedly to organize all transactions of a bank, whereas the user as a customer and user-program as a transaction, the user submits his program just by giving a command to the computer; the responder, the O.S, takes up the program into main memory and process the instructions on the hardware under its control. The following picture shows the layers of interaction between user and the computer.

User
User program
Operating system
Computer hardware

Customer
Transaction
Bank employee
Bank

C Character set

It is a set of symbols called characters, which are supported by the C-language. C supports all most all symbols which are provided in the keyboard

- Lower case alphabets (a-z), Upper case alphabets (A-Z)
- Digits (0-9)
- White space (‘ ‘)
- Math symbols (+ , - , * , / , % < > = …etc)
- Special symbols like   {} [] ()   ! & , “   \ …etc)

C Keywords

Like English language vocabulary, C has its own vocabulary called keywords; thus, Keyword is a reserved word, which has specific meaning in C, and it cannot be used for other purpose, using these keywords, the programs are constructed. C has the following standard set of 32 keywords.
if, else, switch, for, while, break, continue, goto, auto, register, extern, static, volatile, return, enum, void, char, int, long, short, float, double, signed, unsigned, case, const, default, do, union, sizeof, typedef, struct;
Note: All keywords should be written in lower case. Depending on the compiler/vendor, few additional keywords may also exist.

Tokens

Let the expression ‘x+y’. It has three tokens ‘x’, ’y’ and ‘+’. Here x, y are called operands, and ‘+’ is called an operator. The compiler splits all the instructions into individual tokens for checking syntax errors. Splitting expression into tokens and checking is known as parsing.

Now token can be defined as an elementary item in the program, which is parsed by the compiler.

Token can be a keyword, operator, identifier, constant, or any other symbol; For example,

```
a+b        //  this expression has 3 tokens: a , b , +
a<=b       // this has 3 tokens ‘a’, ‘b’, ‘<=’   ( here ‘<=’ is a single token )
c++        //  this has 2 tokens  c , ++          ( here ‘++’ is a single token )
if(a<b)    //  it has 6 tokens
```

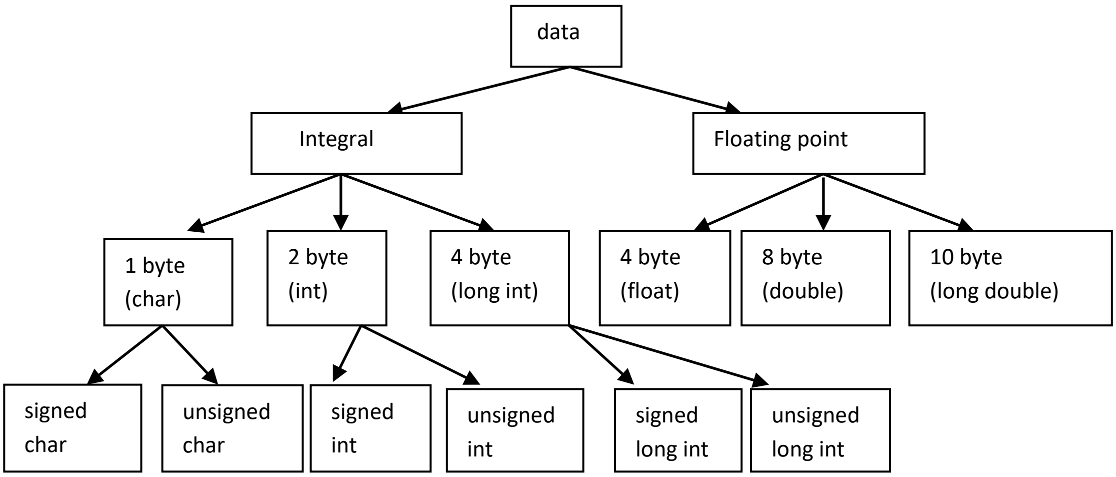
note: in C, ‘+’ is different from ‘++’ , both are different operators.

Classification of Data (Data types)

Data means a value or set of values that represent attributes like age, experience, address, salary, …etc.
for example, the data of employee can have like idno, name, age, experience, address, salary…etc.
The computer hardware (processor) designed to process only two types of data

1.**integral** (also called integer values)
2.**floating point** (also called real numbers or precisions)

The integral data are rounded values like 10, 20, -343, etc, whereas floating points are 15.44, -45.56, etc;
So, in the real world, any kind of data comes under these two types, for example, the employee number, age are integral types. Whereas the basic salary & net salary are stored in floating point format.
The employee name, address are collection of alphabets and other symbols, these are also integer type values. Because in computer, the alphabets are stored in its ASCII codes, for eg: 'A' as 65, 'B' as 66, 'a' as 97.
In C, these data are classified into nine types based on the application's requirement and hardware support; these are known as built-in or primitive or basic types.



Type	Bytes occupied	Range
singed char	1	-128 to 127
unsigned char	1	0 to 255
signed int	2	-32768 to 32767
unsigned int	2	0 to 65,535
signed short int	2	-32768 to 32767
unsigned short int	2	0 to 65,535
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
Float	4	3.4 e-38 to 3.4 e+38
Double	8	1.7e-308 to 1.7e+308
long double	10	3.4 e-4932 to 3.4 e+4932

signed int or **unsigned int** are system dependents, for example, in 16-bit DOS, it occupies 16 bits of memory, whereas in 32-bit Unix/Windows, it occupies 32 bits of memory. So, they occupy 2/4 bytes based on system.

The keyword **signed** is an optional word for signed types. That means, even if we do not mention the keyword 'signed', the compiler by default takes as signed types.
For example, the 'int' is equal to 'signed int' in the C-language.
int a, b, c; → is equal to → signed int a, b , c;

Operators

Operator is a symbol or keyword used to compute mathematical or logical calculations in a program.
C provides rich set of operators for making flexible and simple expressions. Operators are classified primarily into four categories: arithmetic, relational, logical, and bitwise. Assignment, referencing, de-referencing are called special operators. Each operator comes under one of the following types.

Unary Operators: These operators are associated with only one operand.

Binary Operators: These operators are associated with two operands.

Ternary Operators: These take three operands to perform an operation.

Note: just go through these operators briefly, we can learn in detailed in rest of the chapters.

	Unary	Binary	Ternary
Sign	+ ve sign , -ve sign		
Arithmetic	++ (increment) -- (decrement)	+ Addition - Subtraction * Multiplication / Division % Modulo division	
Relational		< Less than > Greater than >= greater than equal to <= less than or equal to == equal to != not equal to	?: Conditional operator
Logical	! (NOT)	&& AND || OR	
Bitwise	~ (1's compliment)	& Bitwise AND | Bitwise OR ^ Bitwise exclusive OR >> Right shifting << Left shifting	
Assignment		= Simple Assignment += Addition Assignment -= Subtraction Assignment /= Division Assignment *= %= …etc	
miscellaneous	sizeof * (dereferencing) & (Referencing)	. direct selection -> Indirect selection , expression separator	

A) Arithmetic operators (+ - * / %)

These operators are used to calculate arithmetical sums such as addition(+), subtraction(-), multiplication (*), division(/) for quotient and modulus division(%) for remainder.
Let us see some of arithmetic expressions. Here a & b are operands

a + b, a - b, a * b, a / b, a % b

Integer division gives only the integer part of the result, that is, the fraction part is truncated (ignored). For example, the result of 15/2 is 7 (not 7.5). The floating point division gives fractions also (15.0/2→7.5)

The modulus-division gives the remainder of a division, for example 17%3→2. Notice that, we cannot apply modulus-division on floating point values as the division goes on and on till the remainder becomes zero, therefore, it is meaningless applying '%' on float values. For example, the instruction 15.0%2.0 is an error.

Observe the result of following expressions

```
15/2      →  7     // decimal part is truncated, because 15 & 2 are integers
15.0/2    →  7.5   // because 15.0 is floating point
15.0/2.0  →  7.5
4/5       →  0
15%4      →  3     // remainder of division
4%6       →  4     // it goes  zero times and gives remainder 4
-15%6     → -3
10%-3     →  1     // in modulo division, sign of result is, sign of numerator
2.5%5     →  error
```

B) Relational operators

These operators are used to find the relation between two values. If relation is found to be true then the result is 1, otherwise it is 0. (The result of this 1 & 0 values is said to be BOOLEAN values).

The operators such as: <   >   <=   >=   ==   !=

```
a > b     //  greater than comparison
a < b     //  less than comparison
a <= b    //  less than or equal comparison
a >= b    //  greater than or equal comparison
a == b    //  equality comparison
a != b    //  non equality comparison
Observe the result of relational operators
2 == 5   →  0
2 != 5   →  1
5 < 10   →  1
10 < 5   →  0
```

C) Logical Operators

These operators are used to combine two or more relations to form a single compound relation.

These operators also gives the result either 1 or 0 (Boolean value)

&& → Logical AND operator
|| → Logical OR operator
! → Logical NOT operator

The operator '&&' works just like the word 'AND' in English language.

The operator '||' works just like the word 'OR' in English language.

syntax1**: relation1 && relation2** → eg: if marks1>50 && marks2>50 then student is passed
syntax2**: relation1 || relation2** → eg: if marks1<50 || marks2<50) then student is failed
syntax3: **! relation** → !true → false

1) When we want two or more relations to be true for one action then we use **AND** operator(&&),

for example if **A>B and A>C** are true, then 'A' is said to be bigger than B,C.

In C-lang, it is written as: **if(A>B && A>C)**

here if both conditions **A>B and A>C** are satisfied, then the operator && gives the result 1 (true).

If anyone of them is false, the result is 0 (false).

2) When we want either of one relation to be true for one action, then we use OR operator(||),
for example, If any **marks1<35 or marks2<35 or marks3<35** are true, then student is "failed".
in C program, it is written as → **if(marks1<35 || marks2<35 || marks3<35)**
if at least one relation is satisfied, the operator **||** gives the result true. If all are false, then result is false

The following truth table shows the result of logical operators

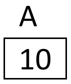
A	B	A &&B	A||B	!A
1	1	1	1	0
1	0	0	1	0
0	1	0	1	1
0	0	0	0	1

D) Assignment operator (=)

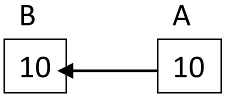
If you are new to the programming, you definitely misunderstand it as an "equal" comparison operator which we use in mathematics. But in C, it is used differently to copy one location value to another location. It is used to assign(copy) right hand side expression's value to left hand side variable; i.e, it copies RHS expression's value to LHS variable;

Syntax 1: variable=expression;
Syntax 2: variable1 = variable2 = variable3 =... variableN = expression;

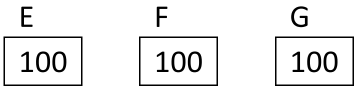
A = 10; //assigns 10 to 'A' (puts 10 in A's memory)



B = A; //assigns 'A' value to 'B', after this instruction both A , B contains same 10



E=F=G=100; // assigns 100 to all E, F, G



C=A+B; // the sum of A+B value assigns to C

A+B=C; // error, not following syntax rules, c value cannot be assigned to A+B

10=A; // error, not following syntax rules, A value cannot be assigned 10

E) Arithmetic assignment operators (shortcut operators)

These shortcut operators are used when left hand side operand is repeated in right hand side at the assignment. For example **'salary = salary + 1000'**, this can be taken in short form as **'salary += 1000'**

The operators are: **+=, -=, *=, /=, %= …… etc.**
A = A + 10; → A += 10
B = B - 10; → A -= 10
A = A * 200; → A *= 200
A = A / 5 → A /= 5

note: these short operators somewhat confusion, as a result discouraged in modern programming.

F) Increment or decrement operators (++ , --)

in programming, it is often needed to increment/decrement variable values by 1.
for this, C provides a very important and versatile operators called '++' and '--'.
these are very famous operators and almost all languages following in computer science.

```
++   is an increment by one operator
--   is a decrement by one operator
```

Let us take one example

```
k=5;   // sets 5 to k
k++;   // this instruction increments 'k' value by 1, so it becomes 6
```

so this 'k++' is a short form of k=k+1

These two operators are again classified into pre & post increment/decrements

```
1)  ++k;    // pre increment
2)  --k;    // pre decrement
3)  k++;    // post increment
4)  k--;    // post decrement
```

Pre increment/decrement has highest priority than any other operator. Hence, this operator is evaluated before any other operator, whereas post increment/decrement has lowest priority than any other operator. Hence, this is evaluated after all other operators are evaluated; (for clarity sees the operator priority table)

Note: the parenthesis ( ) has no effect on these operators. For example: 2*(A++) is equal to 2*A++;

Let us take some examples, Let A=5, B=10;

```
1)   A++;            // this is equal to A=A+1; here 'A' becomes 6
     B--;            // this is equal to B=B-1;  here 'B' becomes 9

2)   B = ++A;        // this is equal to given below, first pre-increment and then assignment
     ++A;
     B=A;
     result is:  A gains 6, B also gains 6

3)   B = A++;  → this is equal to given below, first assignment and then post-increment
     B=A;
     A++;
     result is:  B gains 5, A gains 6

4)   B = ++A  +  ++A  +  A++;  → this is equal to given below, we have two pre-increments, and one post-increment)
     ++A;            // pre increment, so do first  ( here 'A' becomes 6)
     ++A;            // pre increment, so do first  (here 'A' becomes 7 , this is increment of previous value  6 )
     B=A + A + A;    // add all values   (B=7+7+7)
     A++;            // post increment, so do last (here 'A' becomes 8)
     Result of A is 8 ,  B is 21
```

Note: So first do all pre increments, next process normal expressions, at last do all post increments.

```
5)   B  =  A++*3 +  ++A  +  5*++A;  → this is equal to given below
     ++A;
     ++A;
     B=A*3+A+5*A;
     A++;
```

Note: in modern compilers, this ++ and – – has changed their working style.

* pre increment means: increment & substitute its value

* post increment means: substitute & increment its value

let A=1; the observe following substitutions.

```
B  =  ++A*2    +    ++A*3     +    A++*4     +    A++*5  +  ++A;
          2*2    +       3*3      +     3*4       +      4*5     +     6
```

G) sizeof():operator gives the size of memory which is occupied by the variable or constant or data type

```
sizeof(int)          →  2
sizeof(long int)  →  4
sizeof(k)            →  2     //  let  k is int type variable
sizeof(10)          →  2     //  10 is int type value
```

H) Comma operator(,): It is used to separate two or more instructions in the program. It is used in several contexts in the programming. The actual behavior of comma operator is that, it returns the right hand side operand value as the result of the expression.

```
a=2, b=3, c=10;  //  Here comma works as separator
c=a , b;             //  it is combination of two instructions "c=a  and b",  but 'b' does nothing
a = (b, c);          //  it is also a combination of two instructions,   "b and a=c"
```

Operator precedence (like BODMAS rules in maths)

In general, complex expressions are formed with the combination of mathematical or logical operators. While evaluating such expressions, initially the sub expressions are evaluated according to the precedence. For example, the expression 5+2*4 evaluated as 5+8→13. Observe the following table showing the relative precedence of operators in c. Just look once about this precedence, we can learn in detailed in next chapters.

Priority 1	( ) [ ] -> .
Priority 2	! ~ +(sign) -(sign) ++ -- (pre incr/decr)
Priority 3	sizeof &(reference) *(de-reference)
Priority 4	* / %
Priority 5	+ -
Priority 6	<< >>
Priority 7	< <= > >=
Priority 8	== !=
Priority 9	^ | & (ORing, ANDing)
Priority 10	&& ||
Priority 11	?: (conditional operator)
Priority 12	= *= /= %= += -= &=
Priority 13	^= |= <<= >>=
Priority 14	Comma operator (,)
Priority 15	++ -- (post increment/ decrement)

In the above table, Operators in the same row have the same precedence. Among same precedence operators, the evaluation takes place from left-to-right side in the expression. Only the assignment & unary operators are performed from right-to-left. For example a=b=c=d, here first c=d, and b=c , finally a=b;

How to write expressions in C

An expression is a systematic combination of operands and operators to specify the relation among the values. However, any single constant, variable is also called an expression. For example,

1+2	4*5	22/7	10%3	10>5	a	a+b
1==0	4<<1	55<=100	-10	10	b	a<c

C language allows us to use complex expressions, wherein such expressions should be cautiously handled; otherwise it leads to logical errors. Let us see, how an expression should be typed in correct syntax.
For example:

1. $ax^3 + bx^2 + c x + d$: Answer → a*x*x*x + b*x*x + c*x + d

2. $\frac{-b+\sqrt{b^2-4ac}}{2*a}$: Answer → (-b+sqrt(b*b-4*a*c))/(2*a)

3. $\frac{ax^{45} + 6x^5}{cx+7}$: Answer →(a*pow(x,45)+6*pow(x,5)/(c*x+7))

Here pow() and sqrt () are functions, finds power and square root of a given value.

Variable and its Naming rules

Program means collection of data and instructions, the data is represented in terms of names such as age, experience, salary …etc; these names are called variables which refers/holds a value in the program.
Thus, variable is a name given to the memory location in computer main memory, where some values are stored and retrieved; these values may vary during execution of program.

Rules for naming variables, while naming a variable, we should follow rules as explained below

- ✓First letter must be an alphabet
- ✓Only alphabets, digits and underscore are allowed in variable names
- ✓We cannot use any other symbols like comma, hyphen, period…etc.
- ✓Spaces are strictly not allowed.
- ✓Should not be a keyword like ‘if’, ‘else’, ‘while’ …etc.
- ✓The variable name can contain a maximum length of 32 characters.

some valid names	**Some invalid names**
basic_salary	123pq // digits not allowed as first
net_ salary	ab,cd // comma not allowed
age	first-person // hyphen not allowed
weight	i and j // spaces not allowed
name	int // keyword not allowed
marks1	units/month // other symbols not allowed
basicSalary	basic salary // space not allowed
for1	

Declaration of variables

The declaration specifies name, size, type and other characteristics of a variable. Before using any variable, it must be introduced to the compiler about the name, size and type of data it is going to hold.

This introduction is called variable declaration; it involves creation of memory in the RAM and substitution of logical address (binding) where ever it is used in the program.

Syntax: <data type> <variable1>, [<variable2>, <variable3> … <variable n>];

Here in the syntax, the symbols "< >" represent user-defined and are compulsory, whereas square brackets [] represent optional. [This is old style syntax, now a days nobody following this syntax]

The above syntax can be written as: **dataType variable1, variable2, variable3…;**
For example: **int k1, k2=10;**

K1	k2
Garbage value	10
2 bytes	2 bytes

This declaration creates 2 bytes of memory space for k1, k2 in RAM; Here k1 has not been assigned with any value, so by default, it contains garbage value (un-known value), whereas K2 has been initialized with 10;

Let us see more examples:

```
int  age, experience;     // holds person age and experience
int year;                 //  year of date
float salary;             //  salary of employee
```

Initialization vs assignment of variables

We can set a value to the variable at the time of declaration ie, at the time of creation of memory. This is called initialization of variable. Assigning a value after declaration is said to be assignment.

```
eg1:  int K1=10;        // this is called initialization, not an assignment, here k1 initialized with 10;
eg2:  int K2;
        ----
        K2=20;          // this is not initialization, this is called assignment.
```

Note: here above 2 examples initialization & assignment affects the same result, we can follow either.

What is Garbage value?

At the time of declaration, if variable is not initialized with any value, then by default it holds an unknown value called garbage value. This value may be +ve, or -ve or sometime zero.

After completion of execution of one program, the program's binary code or data will not be cleaned automatically from the memory by the O.S, it exists until the next program loads into the same memory; when a new program loads into this same memory, the old program's code & data gets overwritten (replaced). But when just a space is allocated for our program variables, in this space, the previous program's binary values found or collected to these variables, which are anyway garbage to the new program. This garbage exists until new values are assigned to these variables. For example,

```
int X=100, Y;      // here X contains 100, but Y contains garbage value.
```

Note: Some times, this garbage may belong to our current running program itself. When a function is terminated, its variable's space is freed and reallocated for next function's variable. So, in this way the garbage values come into picture.

‘C’ comment line

Comment lines are simply a passage of meaningful text, to give a brief idea about author of program, purpose of program, and other details. It is a good programming practice giving comments to increase the readability of a program. Programmers always add comments at instructions wherever complexity is seen. Thus, the logic of program can be understood by studying comments. We can write our comments anywhere in the program by enclosing the comment text within a pair of /* …. */ This is as

```
/*   ………..  comment  line 1  ……………
     ………… comment line 2   ……………
      ………… comment line 3   …………… */
```

for eg: /* this program is written by Srihari, dated on 10-7-2023
this takes two input values and finds sum of them */

This comment lines are ignored (skipped) while compiling a program, and hence they are not executed. Comments are not a part of the program, they are for documentation purpose. This is ‘C’ comment line and can be used as single line or multi-line in the program.

C++ comment line: C++ supports single line comment, the syntax is

```
// …………………………….comment line…………………………………….
```

The C++ is a superset of C, hence, we can compile C programs on C++ compiler, hence, we can use C++ comment lines in C programs (of course i myself used this comments during explanation of things).

Functions

In general speaking, in our real life, several people are required to accomplish one task with each one of them doing one's specialized sub task. Similar to this, a programmer cannot develop the entire code of one application; instead many programmers contribute the code in terms of sub programs called functions. Function is a sub-program that performs a given sub task in a program. Here each function is designed and written for specific purpose such as calculating power, square root, logarithms …etc; thus collection of such functions makes a C-program. For example, sqrt() is a function, it gives square root value. eg: sqrt(16)→4

The functions are classified into ① library functions, ② user–defined functions

Library functions

These are ready-made functions, designed and written by the C manufactures to provide solutions for basic and routine tasks in the programming. For example, the mathematical functions pow() & sqrt(), and I/O functions scanf() & printf (),etc. The vendor of C, supply these predefined functions in compiled form with C software.

There is huge collection of such compiled functions available in C, hence, these collections are called functions Library. The library functions-body placed in separate files with name "xxxx.lib" in compiled format and they are automatically linked to our program at the time of compilation, and these are hidden to the programmer. (We will see about user-defined functions in functions chapter)

1) Math library functions

The familiar math functions are pow(), sqrt(), sin(), cos(), tan(), log(), log10(),etc.

1) pow(x,y) is a function calculates x^y, where x is base and y is exponent.

```
k=pow(2,3);    // k=8; here 2^3 is calculated and assigned to 'k'
```

2) sqrt(x) is a function calculates square root of x.

```
K=sqrt(16);    // k=4;
```

3) log() is a function, finds the $\log_2$ value

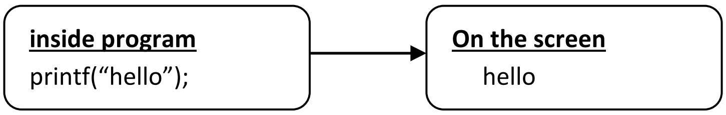
The syntax of these functions are defined in 'math.h' file and any function is used in the program, it should be included at the beginning of program as: **#include<math.h>**

2) Terminal I/O library functions

Generally, all programs are needed to interact with keyboard or monitor devices to accept and print values. Input is taken from the keyboard and the output is displayed on the screen.

The printf() and scanf() are familiar I/O functions defined in 'stdio.h' file

printf(): displays message or output values on the screen, the programs are stored permanently in secondary storage devices like hard disk and when they are gets executed, loaded from hard disk to RAM and then instruction by instruction is executed by the processor. Here nothing will be displayed on the screen while executing a program in the memory, if something wanted to be shown on the screen, we must add an instruction called "printf()", which displays data/message on the screen. For example



```
printf("hello");
```

```
hello
```

syntax: printf(" some text with **format-string** " , list of values to be shown);

the format strings are: %d , %f , %ld , %u , %c , %s , etc.

here format-string is a value injector for int, float, long, etc

here %d for int, %f for float, %ld for long-int etc.

example: let A=10, B=20, C=30;

printf(“ %d %d ”, A , B); → on the screen we get: 10 , 20 [here %d is a substituter for int-value]

printf(“ A&B values are %d %d ”, A , B); → A&B values are 10 20

printf(“ %d + %d = %d”, A,B,C); → 10+20=30

printf(“hello is %d+1, world is %d+2”, A , B); → hello is 10+1 , world is 20+2

printf(“ XX%dYY%dZZ%d” , A, B, C); → XX10YY20ZZ30

printf(“output is %f”, 10.45); → output is 10.45

printf() with width specifier

we can add **‘width’** to the format strings, for example: %5d , %7d , %05d , %f , %.2f , %05.2f, etc.

%5d → here 5 is said to be maximum width of output value which we want to show on the screen.

if width > output-value size then spaces added by the printf() statement.

if width < digits in value then no spaced added, it prints normally (here ignores the width).

For example,

printf(“%7d”, 523); → BBBB523 // Here B means blank space, added by printf()

printf(“%07d”, 523); → 0000523 // pads with zeros instead of spaces

printf(“%02d”, 7234); → 7234 // here width < digits in value, so prints normally (no affect)

printf(“%.2f” , 8671.4256); → 8671.42 // .2 is cutoff decimal values

printf(“%7.3f”, 823.4); → BBBBB823.400

List of all format strings.

%c → It denotes single character (signed/unsigned char)

%d or %i → It denotes signed decimal integer (signed int)

%f → single precision floating point number *(float)*

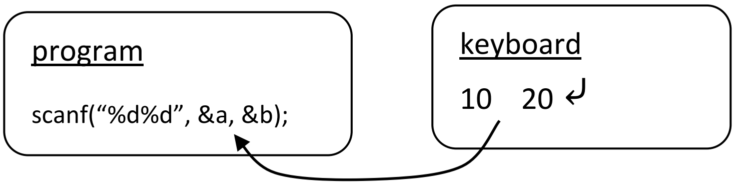
%u → unsigned decimal integer (unsigned int)

%ld → signed long decimal integer

%lu → unsigned long decimal integer

%lf → double precision floating point number (double)

scanf(): used to read input values from keyboard, here user has to feed/enter values in the keyboard while running a program, based on input values, the output is displayed. The scanf() waits for user response until he/she enters some values from the keyboard. ie, pauses the program execution until user enters input values with the end of input “enter-key”. When enter-key is pressed, then the program resumes its execution. For example,



```
scanf(“%d%d”, &a, &b);
```

syntax: **scanf("format string", list of variables with address);**

eg: scanf("%d %d", &age, &experience);

Here '&' is called address operator, it must be prefixed before every variable at scanf().

Let 34,28 are the input of A,B in the following example, then how the user should be given in the keyboard

scanf("%d%d", &A , &B); ← 34 space 28 ↲ // space is the default separator between two values

scanf("%dQQQ%d", &A , &B); ← 34QQQ28↲ // here QQQ is the separator for 34 and 28 values

scanf("%d/%d", &A , &B); ← 34/28↲ // here slash(/) is the separator or 34 and 28 values

scanf("%d,%d", &A , &B); ← 34,28↲ **// conclusion: other than %d should be given as it is**

scanf("Hello%dWorld%d", &A , &B); ← Hello34World28↲

scanf("XY%dZ", &A); ← XY34Z↲

scanf(" %d ", &A); ← space34space↲

note: My suggestion is, do not give any extra symbols in the scanf() format string including spaces.
It should be like "%d%d" should not like " %d %d " [this suggestion is for scanf() and not for printf()]

note: the format strings such as %d , %f tells the conversion of binary to decimal-text and decimal-text to binary while scanning and printing values. These are internally used by the scanf() and printf().

About main() function

The main() works like a start & end point of the program, the execution starts at "void main()" and ends at last closing braces; Actually, the main() is also a function; among all the functions, main() is an important and reserved function, it tells to the compiler the starting & ending point of program. The main() can be written in any of the following ways, and all of them have the same meaning & works in similar way;
(of course different vendors of C, suggested in different ways to this)

void main ()	main()	int main ()

{ --------
-------- 	
-------- 	
}
	{ ----
	----- 	
	----- 	
	}
		{ --------
		-------- 	
		return(0);
		}

Therefore, in a C program there must be at least one function must be exist, that is main(); its execution starts by operating system when user runs the program. So, the main function is called by the operating system which is indirectly by the user, for example, when u press F9 or F10 in IDE;

General Structure of C program

The C program looks like following way

Line1	// Comment line about program
Line2	file inclusion statements
Line3	void main()
	{
Line4	variable declaration
Line 5	input statements
Line 6	process statements
Line 7	output statements
	}

ine1: In this line, the comments are added about program, ie, the purpose of program, input/output of program, who & when has written, etc. Writing comments is a good programming practice. However, this is optional. Compiler ignores comment lines.

line2: Here header files of library functions and other files are included. For example "#include<math.h> for math functions, #include<stdio.h> for I/O functions ….etc;
Note: these files contain syntax of library function to check error at call statements.

line3: "void main ()" represents starting point of program, it tells, the main process begins here.

line4: variables are declared at the beginning of program block, and when this line is executed, space is allocated for variables in the RAM.

line5: the input statements are used to accept values from the keyboard and other terminals.

line6: Here process statements specify the processing of input data.

line7: here the output statements are used to show output on the screen or other terminals.

Note: the opening and subsequent closing braces {} defines a block; as per C syntax rules, the instruction must be given inside block; all instructions must be terminated by ";"

Coding style of C program

Unlike some other programming languages (COBOL, FORTRAN, etc), C grants a freedom to every programmer to follow his own style of coding. However, ensure that, the program should be readable and easy to debug by others. For example, see the following instructions

```
a=10;
b=20;
c=a+b;
```

the above sequence of instructions can be written in single line as: a=10; b=20; c=a+b;
to increase readability, we can give more spaces between expressions, that is, we can add extra spaces before and after of every token such as punctuation symbols, operators, variables and other places.
For example:

1. a+b → can be written as: a + b // observe spaces added before & after '+' symbol
2. a+b<d*e → can be written as: a + b < d * e // observe space added here
3. printf("hello"); → can be written as: printf ("hello") ;

1) Demo program addition of two int-values

```
#include<stdio.h>                   // we we use printf() or scanf(), we have to include this "stdio.h" file
void main()
{       int  A,B,C;                 //  here space creates for A,B,C.  Each takes 2byte
        A=10;                       // assigning 10 to A
        B=20;                       //  assigning 20 to B
        C=A+B;                      //  adding A,B and assigning result 30 to C
        printf("  output is  %d ",  C );  //  Here C-value substitute in  %d  place.
}                                   //  In this case '%' does not work like remainder operator
```

2) Demo program addition of two float-values

```
#include<stdio.h>
void main()
{         float  A,B,C;
          A=10.5;
          B=20.3;
          C=A+B;
          printf(“ output is %f ”,  C );        //  for float-type values use %f
}
```

3) Demo program how to sum up values one by one.

```
#include<stdio.h>
void main()
{         int sum=0;
          sum=sum+3;
          sum=sum+4;
          sum=sum+5;
          printf(“sum is %d ”, sum);
}
```

We will see in next chapter, how to compile and run the program on the computer.

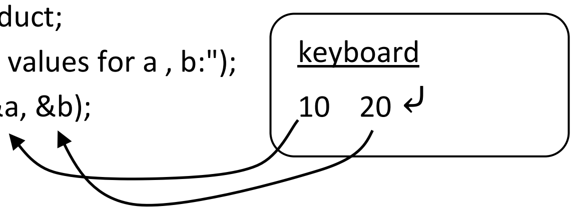
4) Finding sum and product of given two input values

This program scans two values from keyboard and prints the sum and product of two values.

ip: 10 20

op: sum is 30, product is 200

```
#include<stdio.h>
void main()
{         int a, b, sum, product;
          printf("Enter two values for a , b:");
          scanf("%d%d",  &a, &b);
          sum=a+b;
          product=a*b;
          printf("sum is %d , product is %d", sum , product);
}
```



printf("enter two values for a , b:") → this statement asks/displays to the user to feed 2 values for a,b

scanf("%d%d", &a, &b); → this statement reads two values from keyboard(KB) and inserts into given variables (a,b). For two values, we have to use two %d%d, we will see later why %d to be used.

printf("sum is %d, product is %d", sum , product); → this display result as: **sum is 30, product is 200**

5) Demo program how to use pow() and sqrt() math functions.

pow(x,y) → this finds x^Y , here X is base, Y is exponent.

sqrt(x) → this finds square root of x (√x)

```
#include<math.h>      // when we use pow() or sqrt() function, we have to  include this "math.h" file
#include<stdio.h>     // we we use printf() or scanf(), we have to include this "stdio.h" file
void main()
{         int  A , B;
          A=pow(2,10);      // A=2^10
          B=sqrt(16);
          printf(" A=%d  ,  B=%d", A , B );          //  output:  A=1024 , B=4
}
```

6) Finding sum of equation $5x^3 + 2x + 10$, here 'x' is input value

ip: if x is 3

op: 5*x*x*x +2*x+10 → 5*3*3*3+2*3+10 → 135 + 6 + 10 → 151

```
#include<stdio.h>
void main()
{         int  x , result;
          printf("enter x value :");
          scanf("%d", &x);
          result=5*x*x*x+2*x+10;             // we can't write X^3 in the program, we  need to write as  x*x*x
          printf("output is: %d",  result );
}
```

```
------- keyboard --------
enter x value: 3↵
```

Note: for bigger values like x^{10} use pow() function, but for smaller values like x^3 write as x*x*x.
because, the pow() function takes much machine code.

7) Finding Fahrenheit from given Celsius. (formula is: F = 9.0/5*C+32)

This program scans Celsius from keyboard and prints Fahrenheit as output.

ip: enter Celsius: 38

op: Fahrenheit is: 100.4

```
#include<stdio.h>
void main()
{         float   C , F ;
          printf("Enter Celsius:");
          scanf("%f", &C);
          F=9.0/5*C+32;
          printf("Fahrenheit is: %f", F);
}
```

```
------- keyboard -------
Enter Celsius : 38 ↵
```

8) Finding area and circumference of circle

The following program accepts radius from keyboard and prints area and circumference.

Logic: based on input value **radius**, the output (area and circumference) is calculated;

area is: πr^2 and circumference is: $2 \pi r$ (in keyboard there is no **pie** symbol , so use 3.14)

ip: enter radius: 5

op: circle area is 78.5 , circumference is 31.4

```
#include<stdio.h>
void main()
{           float radius, area, circum;
            printf("Enter radius of circle :");
            scanf("%f", &radius);
            area=3.14 * radius * radius;
            circum=2 * 3.14 * radius;
            printf(" circle area is %f  ,  circumference is %f", area , circum );
}
```

------ keyboard ------
Enter Radius : 5 ↲

9) Demo program on manipulating digits in a number

The operator '/' gives quotient of division, whereas the operator '%' gives remainder. For example

2345/100 → 23
2345%100 → 45
2345/10 → 234
2345%10 → 5

2345/1000 → 2
2345%1000 → 456
234/1000 → 0
27/10 → 2

(234/10)%10 → (23)%10 → 3
(234%100)/10 → (34)/10 → 3
23%10 → 3
7%10 → 7

```
void main()
{       int  k , n=2345;
        k=n/100+n%100;
        printf("\n sum = %d", k);          ?

        k=n%10+n/1000;
        printf("\n sum = %d", k);          ?

        k = (n/10)%10 + (n/100)%10;
        printf("\n sum = %d",  k );        ?
}
```

10. Converting time(H:M:S) into seconds format

This program accepts a time in H:M:S format as input and prints output in seconds format.

ip: 2 40 50 (2-hr , 40-min , 50-sec)
op: 9650 seconds

ip: 0 2 50 (0-hr , 2-min, 50-sec)
op: 170 seconds

logic: total seconds is N=H*3600 + M*60 + S // each hour has 3600 seconds, each minutes has 60 seconds.

```
void main()
{       int H,M,S,N;
        printf("Enter time as hours  minutes   seconds  format:");
        scanf("%d%d%d",  &H, &M, &S);
        N=H*3600+M*60+S;
        printf("output time in seconds is: %d", N);
}
Enter time as hours minutes seconds format: 2   40   50 ↲
output time in seconds is: 9650
```

11. Converting seconds time into H:M:S format

Code to accept a time(N) in seconds format and prints output in H:M:S format

```
 ip: 7270
op: 02:01:10    ( 2-hours, 1-minutes, 10-seconds )
```

logic: divide N with 3600 and take the quotient as hours, (1hour=3600seconds)
divide N with 3600 and take the remainder(R=N%3600). This R is the remaining seconds left after taking hours from N. Again divide R with 60 and collect quotient and remainder. The quotient would be minutes and remainder would be seconds.

```
void main()
{       int H,M,S,N,R;
        printf("Enter time in seconds  format:");
        scanf("%d",  &N);
        H=N/3600;
        R=N%3600;
        M=R/60;
        S=R%60;
        printf("output time is  %d : %d : %d", H, M, S);
}
```

12. Accepting basic salary of employee and calculating net salary

This program scans basic salary from keyboard and prints net-salary. The net is sum of BASIC+HRA+TA

```
Process: HRA is 24% on basic          ( HRA → House Rent Allowance)
           TA is 7% on basic           ( TA → Travelling Allowance)
           net salary = basic + hra + ta;
```

```
void main()
{       float basic, hra, da, ta, netSalary;
        printf("Enter basic salary :");
        scanf("%f", &basic);
        hra=24*basic/100;          // calculating 24% for house rent allowance
        ta=7*basic/100;            // calculating 7% for travelling allowance
        netSalary=basic+ta+hra;
        printf("Net salary = %f", netSalary);
}
```

13. Demo program for swapping two variable values

The following program explains how 2 variable values are exchanged one with the other. This is common task in the programming and also known as swapping of data.

Logic: Let the input values are 23 & 45 and assigned to X,Y. See what happens, if one has written as

```
X=Y;
Y=X;
```

When ‘X=Y’ executes, the value of Y is assigned to X and we lost the current value of X. Now the X & Y contains same value 45. When the second instruction Y=X executes, the same value 45 copied back to Y. Now both the variables hold the same value and our purpose can’t be survived. Therefore, to exchange two values a third variable is required. This is explained in the following program.

```
void main()
{     int x, y, temp;
      printf("enter 2 values :");
      scanf("%d%d", &x , &y);
      printf("\n before swapping the x,y are %d,%d", x, y);
      temp=x;         // saving ‘x’ value in temporary variable.
      x=y;            //  replacing x-value by y-value.
      y=temp;         //  now replacing  y-value by x-value.
      printf("\n after swapping the x,y are %d  %d", x, y);
  }
```

Output

```
ip:  enter 2 values: 23, 45
op: before swapping,  the x , y   are  23 ,  45
      after swapping,  the x , y   are  45 ,  23
```

14. Finding reverse of 2-digit number

This program accepts a two-digit number like 47 and prints the reverse of it (74).

Logic: For example, the value 735 composed as → 7*100 + 3*10 + 5*1.

Similarly, the reverse of it is → 5*100 + 3*10 + 7*1

In this way, we can find the reverse of two digit number

```
void main()
{     int n, reverse;
      printf(“\n enter two digit single number :”);
      scanf(“%d”, &n);
      reverse=n%10*10+n/10;
      printf(“\n reverse = %d”, reverse);
}
```

```
10 ) 47 ( 4 → n/10
     40
  ----------
      7 → n%10
```

Let us substitute the value 47 in the expression “reverse = n%10*10+n/10”

```
reverse   =   n%10*10 + n/10;
          =   47%10*10 + 47/10
          =   7*10+4;
          =   74
```

15. Printing sum of arithmetic, relational, and logical operations of two values

```
void main()
{   int a,b;
    printf("enter 2 values :");
    scanf("%d%d", &a, &b);
    printf("\n a+b=%d", a+b);
    printf("\n a-b=%d", a-b);
    printf("\n a%%b=%d", a%b);
    printf("\n a>b=%d", a>b);
    printf("\n a<b=%d", a<b);
}
```

output

```
ip: 11   4
op:
     a+b=15
     a-b=9
     a%b=3      (remainder)
     a>b=1      (true)
     a<b=0      (false)
```

Note: In condition place, the value non-zero is taken as true, whereas, zero is taken as false

16. Printing size of different data types

The following program shows memory occupied by each data type in C.
The sizeof() is an operator, it gives the size of variable or data-type.

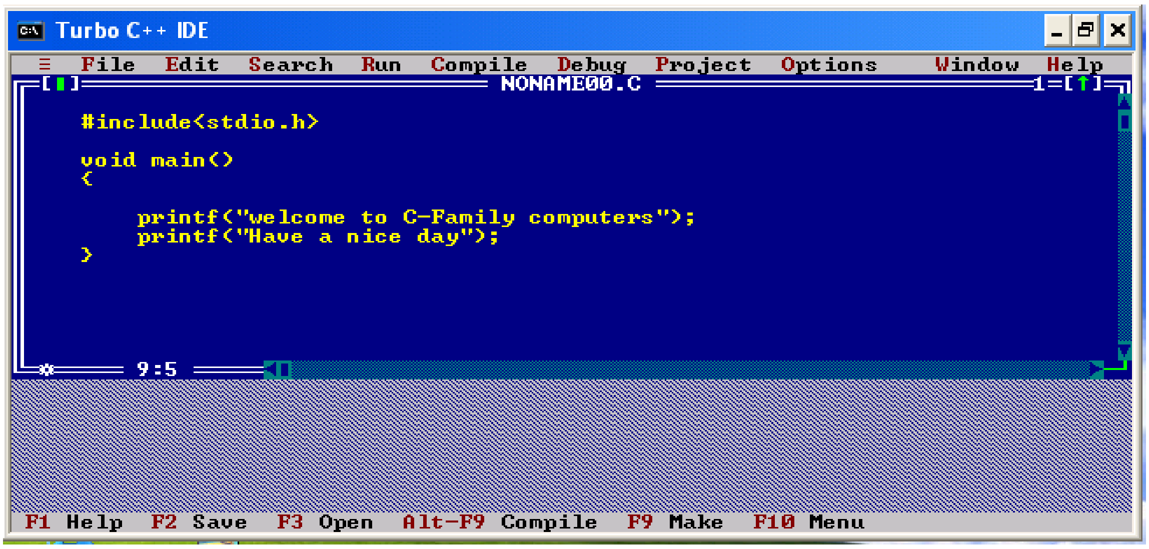
```
main()
{    long int k;
     printf("\n size of int is %d byte" , sizeof(int));
     printf("\n size of float is %d byte ", sizeof(float));
     printf("\n size of k is %d bytes" , sizeof(k));
     printf("\n size of 10 is %d bytes" , sizeof(10));
}
```

Output

```
size of int  is  2 byte
size of float is 4 byte
size of k is 4 bytes
size of 10 is 2 bytes (10 is int-type)
```

Developing, compiling, running, and debugging a program

A) Developing: Once a program has been designed and written, it must be entered before executing on the system. If one wants to feed the program into computer, he needs a software tool to type the instructions, and also allow modifying, deleting, copying, and other editing facilities. This is provided by the special software called Editor. We have several editors to feed (type) the program into computer. The editors like notepad, word pad, ms-word … etc. The notepad allows only typing and saving the code, it does not provide to compile/run the programs. Fortunately, today, every language supplier providing a special Editor along with compiling, debugging, running facilities. All these features are available in one editor, therefore it is called “integrated development environment (IDE)”.

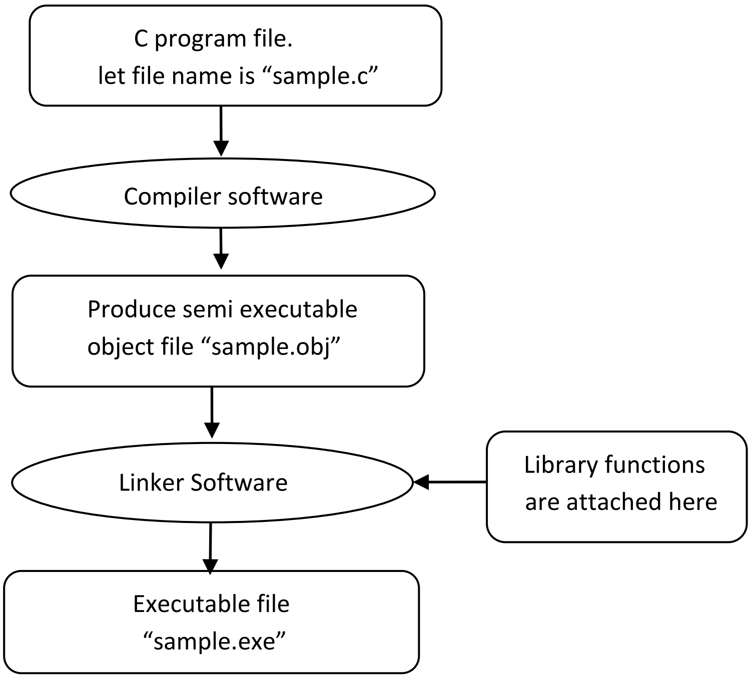

Turbo C++ IDE
≡ File Edit Search Run Compile Debug Project Options Window Help
[■]
NONAME00.C
1=[↑]
9:5
F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu

```
#include<stdio.h>

void main()
{
    printf("welcome to C-Family computers");
    printf("Have a nice day");
}
```

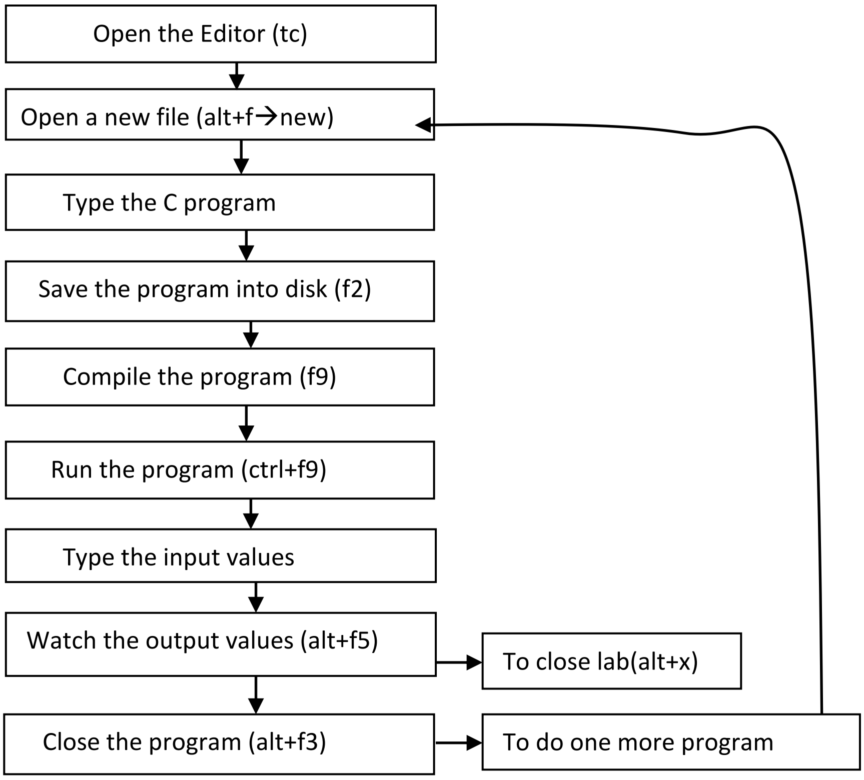
File menu: this menu-driven facility used to handle file operations such as opening a new file, or opening an existing file, saving a file, etc.
Edit menu: this is used to copy, cut, and paste a group of selected lines.
Compile menu: to create object files, making executable files, linking a program….

B) Compiling & Running: **compilation process has two stages called → compiling + linking**

In compilation stage, it checks the typing and syntax errors, after rectifying them, it creates object file, which is semi executable file. Here each function compiled independently and code is generated separately as they are in source file. Later linker links all library and user-defined functions together to form a single executable file. For example, main() function with printf() as shown in the above examples. The following figure shows how they are compiled and linked to the program.


C program file.
let file name is "sample.c"
Compiler software
Produce semi executable object file "sample.obj"
Linker Software
Library functions are attached here
Executable file "sample.exe"

Procedure to execute C-program in turbo C++ IDE on DOS environment


Open the Editor (tc)
Open a new file (alt+f→new)
Type the C program
Save the program into disk (f2)
Compile the program (f9)
Run the program (ctrl+f9)
Type the input values
Watch the output values (alt+f5)
To close lab(alt+x)
Close the program (alt+f3)
To do one more program

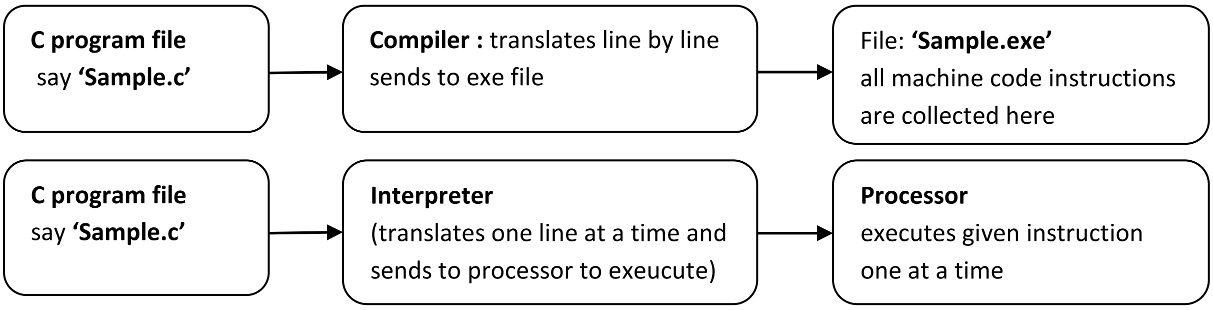
C) Compiler vs Interpreter

Interpreter is a translator, which translates and executes our program. It translates one line at a time and immediately executes on the machine. It translates line by line and executes instantly on the machine. So, it also called instant compiler. Whenever we run the program, the interpreter translates and executes, if we run more times, more translations it takes. It seems to be unnecessary translating same program again & again in every use. This redundant translation leads to wastage of CPU time and other resources.

Actually, interpreter is used for single-use applications like queries, commands, to check spellings in word processing, language translators, internet browsing, etc. The web-sites are coded in the form of html, css, etc and these files are downloaded and interpreted by the browsers, web browsers are nothing but interpreters. So, for single use applications interpreter is the choice.

Compiler translates line by line and stores into ".exe" file to execute later time, thus instead of executing each line instantly on the processor, it stores into .exe file. Now this file contains all machine code instructions of our C-program's source file. This file can be executed directly on the machine whenever we want, while running this .exe file, the compiler is not required. So here, no translation takes place again & again in every use. The interpreter doesn't create ".exe" file, so in every use translation is required.

Some kind of applications needed to execute regularly like calculator, business accounting, games, etc. Here compiler is used to generate machine code file, so that, such file can be executed at any time without translating again and again.
Conclusion: compiler is the choice for regular use code, whereas interpreter is the choice for single use code. For single use applications, it is un-necessary creating machine code file.


C program file say 'Sample.c'
Compiler : translates line by line sends to exe file
File: 'Sample.exe' all machine code instructions are collected here
C program file say 'Sample.c'
Interpreter (translates one line at a time and sends to processor to exeucute)
Processor executes given instruction one at a time

Escape sequence characters

The symbol back slash (\) is called escape character, used to represent some special keys or symbols like New Line, Carriage Return, Tab Spaces, etc. Escape sequence characters are

1. \n → new line (new line)
2. \t → horizontal tab space (4 gaps)
3. \\ → single back slash (\), inserts slash in output.
4. \" → double quotes character (differently with the string enclosure).
5. \' → single quote character (differently with the character enclosure).

Let us see the following examples.

1. printf("Hello\nHari"); → shows "Hello" in one line and "Hari" in next line on the screen
2. printf("Hello\tHai"); → Hello   Hai (4-gaps between Hello & Hai)
3. printf(" \"hello\" "); → "hello"
4. printf(" \'hello\' "); → 'hello'
5. printf("\\ hello\\") → \hello\

Type casting

Process of converting a value from one type to another type is called type casting. In C, the type casting is done in two ways. ①**Implicit type casting (automatic casting)** ②**Explicit type casting (manual casting)**

①Implicit Type Casting

For example, the expression 10+20.54 (int+float), here the value 10 automatically converts **int to float** (10 to 10.00) because, we must hand over same size and type of values to the processor before computing. The processor performs operations bit by bit on two values, so they must be same size and type. Compiler automatically does this conversion when two values are not same type in the expression. In this way, Compiler automatically converts/promotes this **lower-type values to higher-type** with in the expression.

Expression →	After conversion →	Result
int/int	no conversion	int
float/int	float/float	float
int * int	no conversion	int
int * long int	long int * long int	long int
int * float	float * float	float
float * double	double * double	double

Let us see some examples,

eg1: 10.5 + 20; here the value 20 converted into 20.00 by the compiler implicitly

eg2: 10 * 20.00; here the value 10 converted into 10.00 by the compiler implicitly

eg3:

```
int x=10;    float y;
y=x;   // Here 'x' value 10 is converted to float-type, so 10.00 stored into 'y'
```

eg4:

```
long int Y=9023455L;
int X;
X=Y;    //here we are trying to assign 4-byte value into 2-byte memory, so only right most 2byte Y value is assigned to X
                                                                                          results loss of some bits.
```

eg5:

```
int x;  float y=29.34;
x=y;   //Here the integer part of 'y' value 29 is assigned to 'x' ( fraction bits will be truncated/omitted )
```

eg6:

```
void main()
 {       int X=5;
         float Y=12.33, Z;
         Z=X*Y;                          //  here 'X' behaves like a float-type at this expression
         printf(" Z value = %f", Z);     // Z value  =  61.65
 }
```

eg7:

```
void main()
{        float k;
         int a=10, b=3;
         k=a/b+5;
         printf("\n  Result of K = %.2f",k);    // Result of K=8.00
 }
```

Here, the expression a/b gives **int** type result 3 (not 3.33) as both a&b are integer types.
Later, this result is added to 5, which is again **int** type; finally, this result gets converted implicitly to **float** and assigned to k. Therefore the output → 'Result of K=8.00' . The solution is, explicit casting required. This is as given below examples.

②Explicit type casting

Sometimes, we need to convert explicitly to get desired result from the expression.

syntax: (conversion-type) expression

eg1: (float) 15 → 15.00
(int) 2.6 → 2

In the above expression, 15 is **int**. But, upon prefixing **(float)15,** its type will be changed to float as 15.00

eg2:

```
void main()
{    int a=10, b=3;
     float k;
     k=a/b;
     printf("\n Before   type  casting K=%f " , k );
     k=(float) a/b;      // here 'a' converted by us,  whereas 'b' converts by the compiler.
     printf("\n After type casting K= %f ", k );
}
```

```
Before type casting K=3.000000
After type casting K= 3.333333
```

eg3:

```
void main()
{    int a=30, b=20000;
     long int k;
     k = 10 + a*b;
     printf("\n Before  type  casting K=%ld", k);
     k = 10 + (long int) a*b;
     printf("\n After type casting k=%ld", k);
}
```

```
Before type casting K=10186 (un-expected value)
After type casting K= 600010
```

K=10 + a*b → here first, the result of 'a*b' is calculated and stored into a temporarily variable (say it is T), later 10 is added to T. Finally, T value is assigned to K. The nameless variable T is automatically created by compiler and it is of type **int;** because a & b are **int** types, so T will be the **int** type.

But the result of a*b is bigger than the **int**eger range and it cannot hold by T, results loss of some bits while storing into T. Solution is, **K=10+(long int)a*b** → here 'a' is **long int,** which we converted, and 'b' is converted by the compiler, then T will be the **long int** and it can hold the result of a*b;

Representation of Constants in C

In programming, Data is represented in two ways in terms of constants and variable data. Constants are represented in special manner by adding format string or other symbols to the value. For example, '10L' is long int type, 10F is float type..

Automatically, compiler understands some types of constants in the program. For example, the constant '10' is considered as int-type, 34.55 as double-type, 'A' as char-type. These are default types and automatically understand by the compiler. But some constant types are needed to specify explicitly along with format string. The following list explains about all constant types. Do not use other symbols like comma, spaces, quote, etc while specifying constants.

integer Constants

A collection of one or more digits with or without a sign referred to as **singed int** constants. This is the default type for integral values.
eg. 5, 256, 9113, 6284, +25, -62, etc are valid **signed int**egers.
Similarly, to represent unsigned int, the format string 'u' or 'U' is suffixed to the value.
eg. 67U, 43U, 4399u, 45u are valid **unsigned int**egers.

Note that, -3428u is also a valid number. Here this -3428 is converted into equivalent unsigned integer 62108. Because, here the sign bit is also considered as data bit. (But this style is not recommended)
Some invalid declaration of integer constants is:
15,467 $25,566 4546Rs

long integer constants

for the long integer constants, the format string 'l' or 'L' is suffixed to the value.
eg: 2486384L –123456L 5434545l –34545l are valid long int constants
893434lu -3Lu … etc are valid unsigned long integers.

Octal int/long integer constants

for the octal integers, zero is prefixed before the value. (0 is like octal).
eg: 0123, 0574, 01, 046342L etc are valid octal integers.
0181, 09, 12, etc are in-valid octal integers. (In octal system 0-7 digits are used)

Hexadecimal int/long integer constants

in the hexadecimal number system, the symbols 0, 1, 2, 3 … 9, A, B, C, D, E, F are used.
(Total 16 symbols are used; because, the number system's base is 16)
The symbol "0x or 0X" is prefixed before the hexadecimal number.
For example: 0x1, 0x123, 0xa5, 0xfldb, 0x0, 0xABC, 0xflc etc are valid hexadecimals.

character constants

To specify a character constant, we must enclose the character within the pair of single quotes.
eg. 'A' 'z' 's' '8' '+' ';' etc

string constants

It is a collection of characters enclosed within double quotes, used to represent names, codes, abels, etc.
eg: "Hello! Good morning!" "Magic mania" "A" "123" "A1" "#40-5-8A" "C-Family"

float constants (Real numbers)

We can specify the real numbers in two different notations, namely general and scientific notation.
Here the format string 'f' is attached at the end of value.
In normal representation: 3.1412f -25.62f 98.12f -045632.0f 1.f 9.13f
In scientific exponential representation: the syntax is [-]d.dddde[+/-]ddd where d is any digit.
2.32e5f (means 2.32 x 10^5 i.e. 232000.0)
12.102e-3f (means 12.102 x 10^-3 i.e., 0.012102)
-2. 165e6f (means -2.165 x 10^6 i.e., -2165000.0)
3.68e3f (means 3.68 x 10^3 i.e., 3680.0)

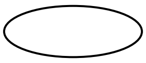
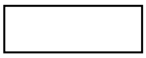
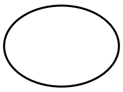
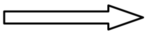
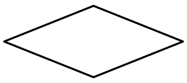
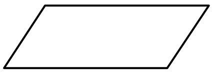
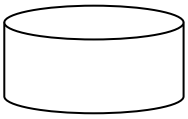
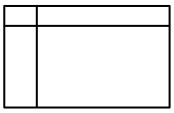
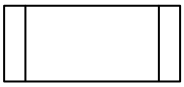
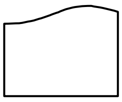
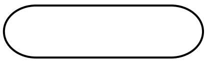
double constants: This is the default type for real numbers (floating point values).
eg: 1.2 45.3 4.39393 349.34899034
45.4e15 44.54e4 ('l' or 'L' is option for double)

long double: for the long double, the format string 'l' or 'L' is used.
eg: 3.14L
314.41 (Equivalent to 314. 0L)
3.68e3L (Means 3.68 x 103 i.e., 3680.0L)

Flow chart

It is a pictorial representation of flow of a program, which illustrates the sequence of operations to be performed to get the solution of a problem. It is often used as a visual planning tool, how the control to be moved from one point to another point to get a solution. It is like drawing a building plan before constructing a building.

Flowcharts are generally drawn in the early stages of formulating computer solutions. It facilitates the communication between programmers and business people. It plays a vital role to understand the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high-level languages. The following mathematical symbols are used to represent specifications with directions to indicate the flow of control.

	**S**tart/stop a program
	Rectangle for processing general instructions
	**C**ircle for connection
	Arrows for directions
	Rhombus for decision making
	Parallelogram for Input/output from terminal
	Cylinder for secondary Memory (external storage)
	RAM (internal storage)
	Predefined actions
	File
	Looping

Algorithm

In mathematics and computer science, an **algorithm** is a finite set of computational instructions that carry out a particular task, which takes input and produces desired output. In simple words, it is step-by-step oral explanation of logic how to construct instructions in a program; here English & math symbols are used to explain logic in words

Flowchart depicts how the control to be moved in the program from one point to another point to get a solution, whereas algorithm tells step-by-step explanation how to construct instructions in a program.

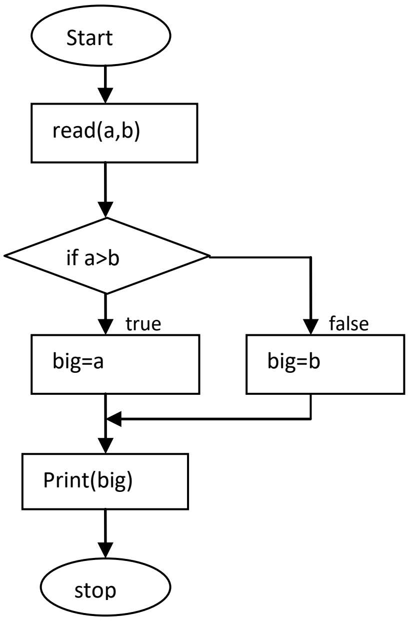
This is an independent method to explain the logic regardless of programming language used by the programmer. Here English and math symbols are used to explain the logic. People write algorithm for complex and complicated tasks to explain the logic in words, for example, sorting algorithms, searching algorithms, shortest path in network …etc. Any algorithm follows,

1. Describes the input and output
2. Describes the data entities with purpose (variables)
3. Explains process in step by step using English and math symbols
4. Add comments at every step what it does, use square brackets [ ] for comments
5. Each instruction is clear and unambiguous (definiteness)
6. The algorithm terminates after finite number of steps

Note: there are no standard rules and regulation to implement algorithm/flowchart, so one can implement one's convenience but ensure that other must be understood it.

The flowchart and algorithm to find big of two numbers

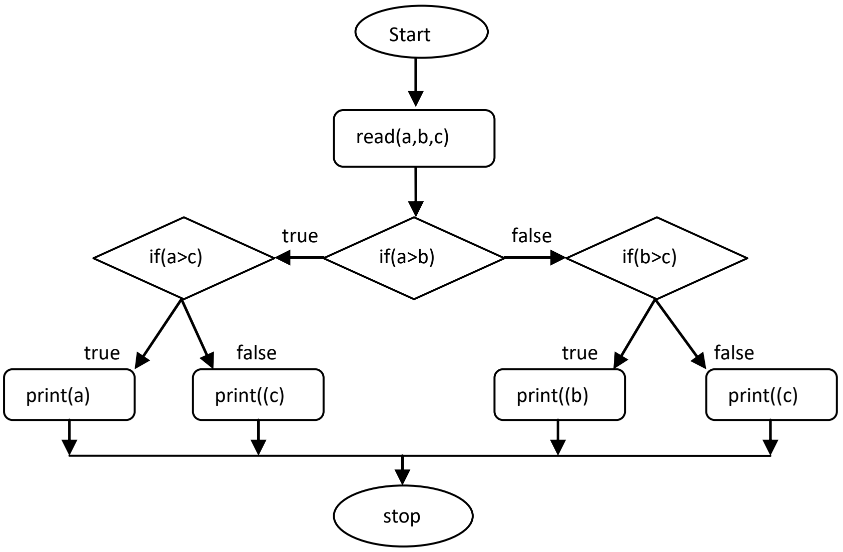
Flow chart to find big of two numbers



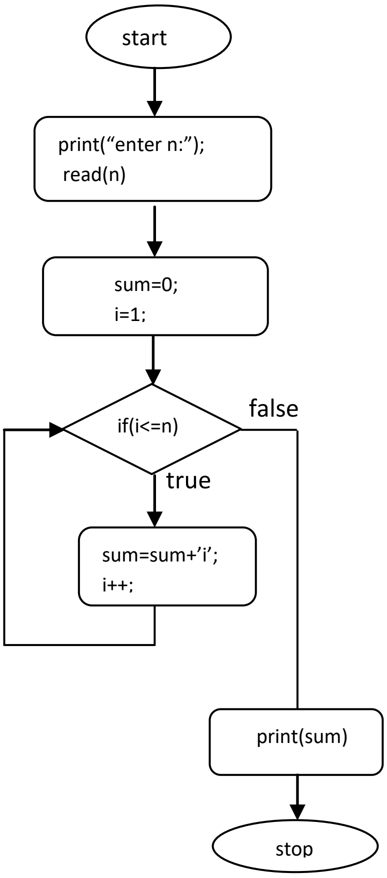
Algorithm to find big of two numbers

```
step1: let us take a,b as integer type variables
step2: let us take 'big'  to store big value
step3: [ scanning input from keyboard ]
          read(a,b)
step4: [ finding big of 2 numbers ]
          if(a>b) big=a
          else  big=b;
step5: [ printing output ]
          print(big)
step6: [ closing the program]
          stop
```

Flowchart for finding big of three numbers



Flowchart to Find sum of 1 to n numbers(1+2+3+4…+n)



Pseudo code

Pseudo code is a short hand way of describing a program or algorithm in general language syntax rather than in specific language syntax. Algorithm uses English and math symbols to explain logic in stepwise, whereas pseudo code uses programming language syntax to explain logic in term of instructions. So pseudo code resembles the program with general language syntax. Most of the people adopt **C** or **Pascal** language's syntax and their control structures to describe the pseudo code.

Algorithm is understandable by any nonprogrammer but pseudo code understandable by a person who knows at least one programming language. **So it is a method used to explain the logic between two programmers. It is easier for programmers to understand the general workings of one program which is written by another programmer.**

Flow chart shows only execution flow (directions), algorithm explains, what are the steps be taken to get a solution, but it does not give clear explanation of how to write instructions, for example, to swap 2 values in the algorithm, we write as: swap(a,b). In pseudo code we write as: t=a, a=b, b=t. So, in pseudo code every step is written in detailed, which is near to programming. So, one can say loosely, pseudo code is nothing but a program without accurate syntax (syntax is ignored).

[Flowchart→Algorithm→ Pseudo code→Program]

Example for Pseudo code: Calculating sum of 1 to n numbers (1+2+3+4…+n)

Algorithm (finding sum of 1 to n)

```
1. [ variable declaration]
     n, i, sum : integers
2. [ scanning input]
     read(n);
3. [ initializing variables]
     i=1; sum=0;
4. [ adding values 1,2,3,… n to sum]
     Repeat until<=n
          sum=sum+i;
          i++;
5. [ printing ouput]
     print(sum);
6. [ exiting the program ]
     stop;
```

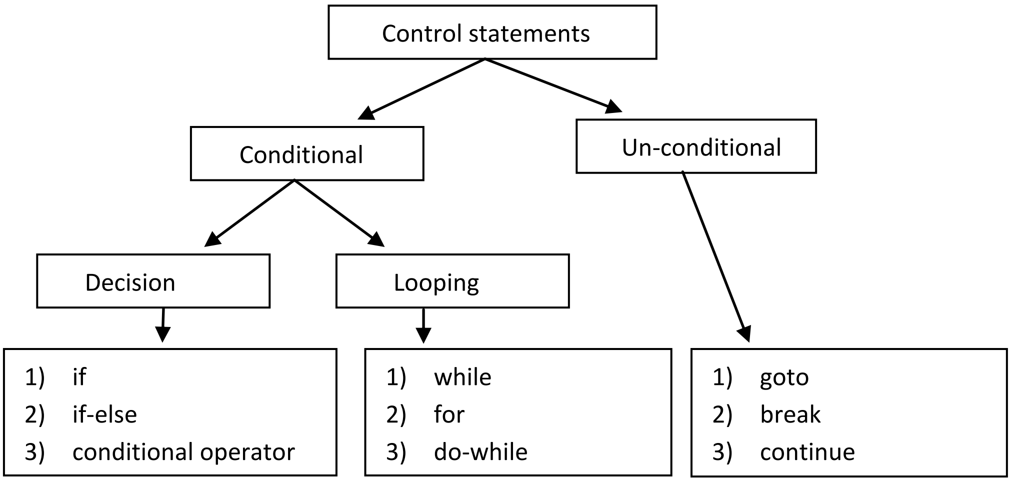
Pseudo code (finding sum of 1 to n)

```
1. Start
2. int sum,i;
3. read(n);
4. set i=1, sum=0;
5. while(i<=n)
   {   sum=sum+i;
       i=i+1;
   }
6. print(sum);
7. stop;
```

Control Statements

Generally, instructions in a program are executed sequentially one after the other, in an order in which they appear in the program. However, in practice, we need to bypass the control over some set of instructions, or repeatedly process some instructions or to alter some other sequence depending upon the requirement. For this purpose, special statements are necessary in programming to handle the control accordingly.
C possesses such a handy control statements like if, if-else, switch, while, etc. These statements alter the normal execution sequence of a program according to our requirement. Therefore, these statements are called as control statements. These control-handling statements are of two types.

① **Conditional control statements** ② **Unconditional control statements**

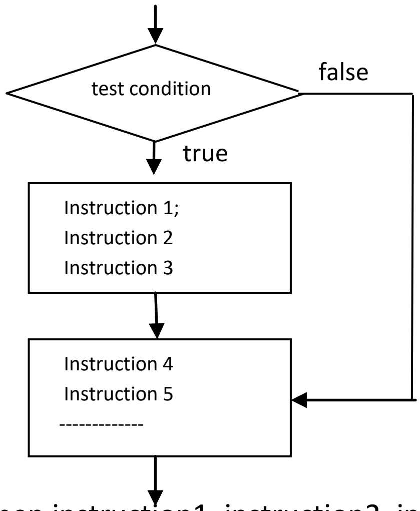


Decision control statements: These statements change the flow of execution depending on given logical condition. Used to execute or skip some set of instructions based on given condition.
Loop statements: These are also called iterative statements, used to execute some set of instructions repeatedly until a given condition is failed.
Unconditional control statements: These statements unconditionally transfer the control from one location to another specified location within a program.

if – statement

It is a decision control statement, executes or skips a given set of instructions depending upon the given test condition. The syntax is

```
if(test-condition)
{     instruction 1;
      instruction 2;
      instruction 3;
}
instruction 4;        // after "if" statement
instruction 5;
------
------
```



In the above **if-statement**, if the **test-condition** is success (true) then instruction1, instruction2, instruction3 will be executed, later the control comes out of **if-body** and proceeds with instruction4, instruction5 …etc. This is shown in the flow chart.
If the result of **test-condition** is false then control jumps out of **if-body** and proceeds with instruction 4, instruction5 …etc. In this case, the instruction1 to instruction3 will not be executed.

Demo1 program, explains how an if-statement executes

```
#include<stdio.h>
void main()
{
        if( 10 > 5 )
        {       printf("hello ");
        }

        if( 10 < 5 )
        {      printf("hai ")
        }
        printf("bye");
}
```

op: hello bye

In the above, the first condition (10>5) is a true, so **printf("hello")** will be executed, but the second condition (10<5) is false, so **printf("hai")** will not be executed. The final **printf("bye")** executes irrespective of if-statements.

Demo2 program, explains how an if-statement executes

```
void main()
{      if( 1 < 2 )
       {      printf(" Red ");
       }
       printf(" A ");
       if( 1 > 2 )
       {      printf(" Blue ");
       }
       printf(" B ");
       if( 1 == 1 )
       {     printf(" Green ");
       }
       printf(" C ");
       if( 1 != 1 )
       {      printf(" Yeloow ");
       }
       printf(" D ");
}
```

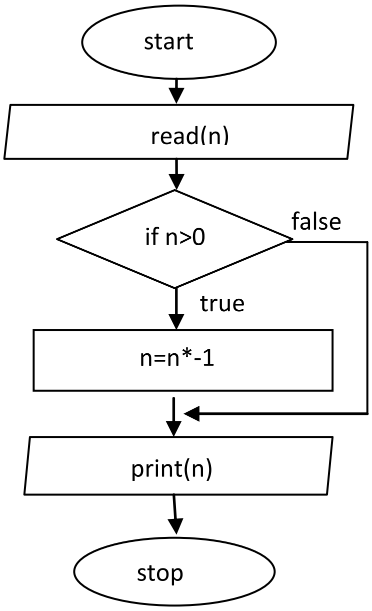
op: Red A B Green C   D

Finding absolute value of a given number ‘n’ modulus of |n|

This program accepts +ve or –ve integer from keyboard and displays result in +ve.

If user entered –ve value, converts it into +ve by multiplying it with -1.

```
ip:  -14                    ip: 15
op:  14                     op: 15
   void main()
   {   int n;
      printf(“Enter a +ve/-ve value ”);
      scanf(“%d”,&n);
      if(n<0)                // if n<0 means, it is -ve
      {    n=n*-1;           // if –ve, then converting to +VE
      }
      printf(“\n Result = %d”, n);
   }
```



In this program, the instruction n=n*-1 executes when the input is -ve.

Finding difference of two numbers, the difference must be +ve

```
ip:  12   18                    ip:  18   12
op:  6                          op:  6
   void main()
   {     int  A, B , diff ;
         printf(“Enter two values :”);
         scanf(“%d%d”, &A, &B);
         diff = A-B;
         if( diff<0 )     // if difference is -ve , then converting into +ve
         {       diff = -1 * diff;
         }
         printf(“difference is %d”, diff );
   }
```

Sorting 3 values into ascending order

If three integers(A, B, C) are input from KB, code to replace 3 values into sorted order(A<B<C)

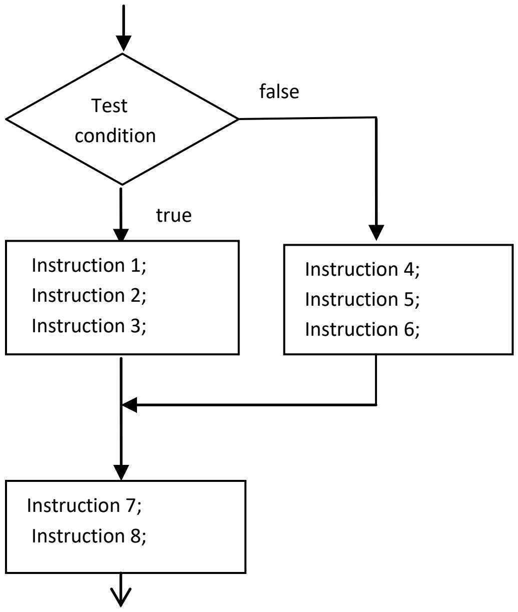
```
ip: 12   5   65        ip: 12  6   3             ip:  10    5    2
op:  5   12   65       op: 3   6   12            op:  2     5    10
if(A>B)
{    temp=A;             // this swaps   10       5     2  →  5    10    2
     A=B;
     B=temp;
}
if(A>C)
 {   -----              // this swaps   5     10     2  →  2    10     5
 }
 if(B>C)
 {   ------             // this swaps    2    10     5  →  2    5    10
 }
print A, B, C          // output will be sorted order, this logic works for any kind of input values.
```

---

if-else statement

This is two-way decision control statement, executes either if-block or else-block based on given condition. If the condition is true then if-block executes, otherwise else-block executes. Let us see the syntax,

```
   if (test-condition )
   {
          instruction 1;
          instruction 2;        // if-block
          instruction 3;
   }
   else
   {
          instruction 4;
          instruction 5;        // else-block
          instruction 6;
   }
   instruction 7;
   instruction 8;
    ----
```



In the above **if-else** statement, always only one block will be executed, either **if-block** or **else-block**. The test-condition is true, **if-block** is executed, and otherwise, the **else-block** is executed. If the test-condition is true, the instruction1, instruction2,instruction3 will be executed, later control jumps out of **'if'** construct and proceeds with instruction7 , instruction8 …

If the condition is false, the **else-block is** executed i.e., instruction4 , instruction5, instruction6 are executed and then control proceeds with instruction7, instruction8 … statements.

Demo program how an if-else executes

```
   void main()
   {    if (1<2)
        {     printf("sky is blue");
        }
        else
        {    printf("sea is blue");
        }

        if (1>2)
        {     printf("C is simple");
        }
        else
        {    printf("C++ is also simple");
        }
   }
```

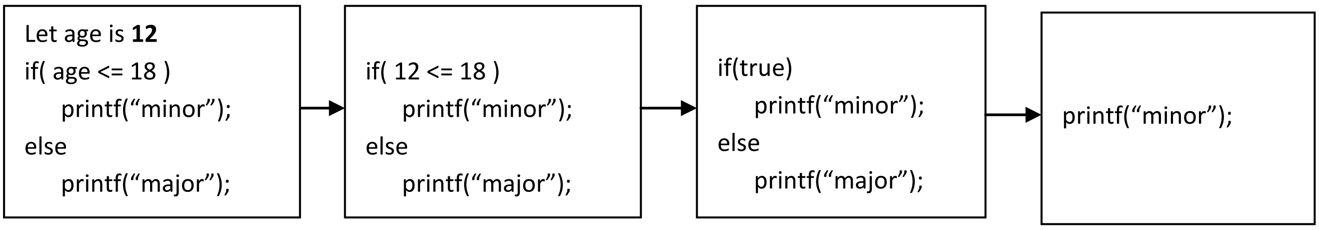
op: sky is blue   C++ is also simple

Finding a person is minor or major

This program finds whether a person is minor/major depending upon his age.

If age<=18 then it displays “minor”, otherwise “major”

```
  ip: 15                 ip: 25
 op: minor              op: major
 void main()
 {    int age;
      printf(“Enter the age:”);
      scanf(“%d”, &age);
      if(age<=18)
      {    printf(“minor”);
      }
      else
      {    printf(“major”);
      }
 }
```



```
Let age is 12              if( 12 <= 18 )             if(true)                   
if( age <= 18 )               printf(“minor”);           printf(“minor”);          printf(“minor”);
   printf(“minor”);   →    else                  →    else                   →
else                          printf(“major”);           printf(“major”);
   printf(“major”);
```

Finding given number is odd/even

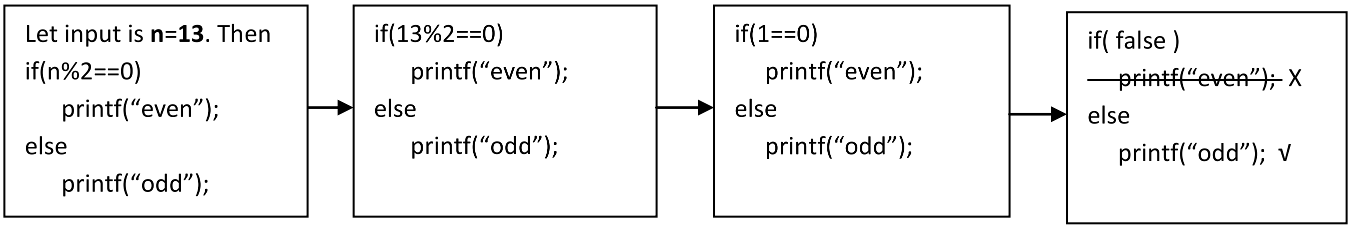
This program accepts a number from keyboard and finds whether it is odd or even?

```
     ip: 25                      ip: 24
    op: odd                     op: even
```

Logic: divide the input number with 2, and check the remainder; if the remainder is zero then it is said to be even, otherwise odd.

```
void main()
{     int n;
      printf(“Enter a number :”);
      scanf(“%d”, &n);
      if(n%2==0)                   // n%2 gives remainder of division
      {    printf(“even number”);
      }
      else
      {    printf(“odd number”);
      }
}
```

The evaluation of **if-statement** is as follows



```
Let input is n=13. Then    if(13%2==0)              if(1==0)                 if( false )
if(n%2==0)                    printf(“even”);          printf(“even”);          ~~printf(“even”);~~ X
   printf(“even”);    →    else                 →   else                 →   else
else                          printf(“odd”);           printf(“odd”);           printf(“odd”); √
   printf(“odd”);
```

About Pair of Braces { }

The pair of braces is optional when if-body or any control statement body contains only single instruction. That is, the pair of braces is not required when if-body or else-body contained only single instruction. The following code shows how to remove braces when single instruction exists in if-body or else-body.

```
if( A>B )
{      printf(“Red”);
}
printf(“Blue”);
printf(“Green”);
   -------
```

→

```
// code after removing unnecessary braces
     if( A>B )
          printf(“Red”);

     printf(“Blue”);
     printf(“Green”);
       -------
```

```
if( A>B )
{      printf(“Red”);
}
else
{      printf(“Blue”);
}
printf(“Green”);
 -------
```

→

```
// code after removing unnecessary braces
     if( A>B )
          printf(“Red”);
     else
          printf(“Blue”);

     printf(“Green”);
      -------
```

```
if( A>B )
{    printf(“Red”);
}
else
{    printf(“Blue”);
     printf(“Green”);
}
printf(“White”);
```

→

```
// code after removing unnecessary braces

     if( A>B )
          printf(“Red”);
     else
     {    printf(“Blue”);
          printf(“Green”);
     }
     printf(“White”);
```

```
if( A>B )
{      printf(“Red”);
       printf(“Blue”);
}
else
{      printf(“Green”);
}
printf(“White”);
```

→

```
// code after removing unnecessary braces
     if( A>B )
     {    printf(“Red”);
          printf(“Blue”);
     }
     else
          printf(“Green”);

     printf(“White”);
```

```
if(A>0)
{   printf(“ A is +VE”);
}
if(A<0)
{   printf(“ A is -VE”);
}
if(A==0)
{   printf(“ A is Zero”);
    printf(“ A is +VE”);
}
```

→

```
// code after removing unnecessary braces
     f(A>0)
        printf(“ A is +VE”);

     if(A<0)
        printf(“ A is -VE”);

     if(A==0)
     {   printf(“ A is Zero”);
         printf(“ A is +VE”);
     }
```

Code with Indentation

the instructions inside if-block or else-block or any block should be started with indentation.
The indentation means giving tab-space before instructions (it is like starting space before paragraph begins)
This **indentation** makes the program easy to read and understand. It signifies which instruction is inside and which is not. Remember if we do not follow indentation then code becomes difficult to read & understand.
C compiler does not force you to follow it, so it does not show any error if indentation is ignored.
(Python language forces you to follow it). Following example shows indentation.

```
if( A>B )
      printf(“Red”); → with indentation
printf(“Blue”);
printf(“Green”);
------
```

```
if( A>B )
printf(“Red”);  → without indentation (makes confusion)
printf(“Blue”);
printf(“Green”);
-------
```

```
 if( A>B )
      printf(“Red”);   → with indentation
else
      printf(“Blue”);  → with indentation
 printf(“Green”);
 ------
```

```
if( A>B )
printf(“Red”); → without indentation (makes confusion)
else
printf(“Blue”); → without indentation (makes confusion)
printf(“Green”);
------
```

```
if( A>B )
{
      printf(“Red”);   → with indentation
      printf(“Blue”);
}
printf(“Green”);
------
```

```
if( A>B )
{
printf(“Red”);  → without indentation (little confusion)
printf(“Blue”);
}
printf(“Green”);
```

with indentation, no confusion

```
if(a==10)
{
      printf(“red”);
      if(b==20)
             printf(“green”);
      else
      {     printf(“blue”);
            printf(“white”);
      }
      printf(“black”);
}
```

Code without indentation, here lot of confusion

```
if(a==10)
{
printf(“red”);
if(b==20)
printf(“green”);
else
{
printf(“blue”);
printf(“white”);
}
printf(“black”);
}
```

something wrong in this code, fix it

```
if( A<B)
    printf(“one”);
    printf(“two”);
else
   printf(“three”);
   printf(“four”);   // take into else-body
printf(“five”);     // don’t take into else-body
```

something wrong in this code, fix it

```
a=4, b=4;
if(a=b)        // takes as assignment,  not as comparison, so use ==
    printf(“a, b are equal”);
else
    printf(“a,b are not equal”);
```

Finding a year is a leap year or not?

This program accepts year in a date and finds whether it is leap year or not?

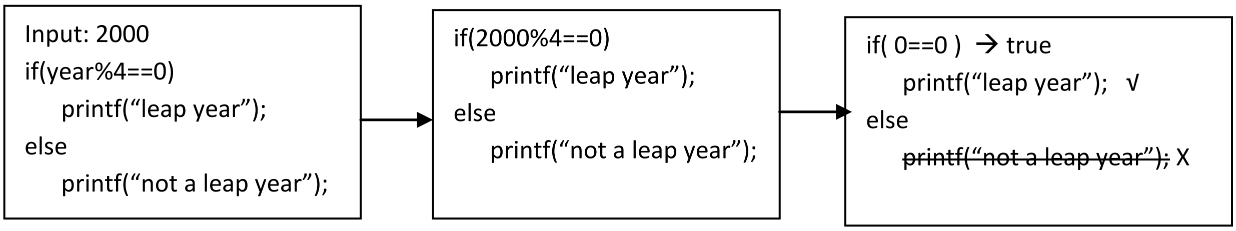
```
    ip: 2000                      ip: 2005
   op: 2000 is a leap year       op: 2005 is not a leap year
```

Logic: if a year is divisible with 4, then it is said to be leap year otherwise non-leap year.

```
void main()
{     int year;
      printf(“Enter year of date:”);
      scanf(“%d”,&year);
      if(year%4==0)                    // the operator % gives remainder of a division
           printf(“\n year is a leap year”);        // observe here , braces removed
      else
           printf(“\n  year is not a leap year”);
}
```

If the remainder of division “year%4” is zero, then year is said to be leap year, otherwise, not a leap year.

The evaluation of **if-statement** is as follows



```
Input: 2000
if(year%4==0)
    printf(“leap year”);
else
    printf(“not a leap year”);
```

```
if(2000%4==0)
    printf(“leap year”);
else
    printf(“not a leap year”);
```

```
if( 0==0 )  → true
    printf(“leap year”);   √
else
    ~~printf(“not a leap year”);~~ X
```

Finding biggest of two numbers

This demo program accepts two numbers from keyboard and displays big.

```
    ip: 12   5                          ip: 7    15
   op:  big=12                         op: big=15
```

```
void main()
{      int a,b,big;
       printf(“Enter any two numbers:”);
       scanf(“%d%d”, &a, &b);
        if(a>b)
            big=a;
       else
            big=b;
       printf(“biggest is %d “ , big );
 }
```

Here, if the condition a>b is true, then **big=a** will be executed, or else, the **big=b** will be executed. So, always either if-block or else-block will be executed. Observe above code, braces removed.

Above code in simple style

if-body & else-body contains *only single instruction* then better to write in single line as shown below (many professionals coders follow this style of coding only)

```
If( a>b)
    big=a;
else
    big=b;
```

→

```
// writing in same line
if( a>b)  big=a;
else  big=b;
```

Finding difference of two numbers

This program accepts two values from keyboard and prints the difference in +ve (like |x|) (absolute value of difference of two numbers)

```
 ip: 12   18                ip:  19   11
op: 6                      op:  8
```

Logic: User may enter numbers in any order, for example (12, 18) or (18, 12).

After subtracting, if the difference is –ve, then convert by multiplying with -1.

```
void main()
{     int a, b, diff;
     printf("Enter any two numbers:");
     scanf("%d %d", &a, &b);
     diff=a-b;
     if( diff < 0 )                    // if difference is  -ve
          diff = -1*diff;              // converting to +ve by multiplying with -1
     printf("\n difference = %d", diff);
}
```

We can write the logic in other way as (subtracting small from big)

```
if(a>b)   diff=a-b;
else   diff=b-a;
```

Finding reverse of 2 and 3 digit number

This program accepts a single number contained 2/3 digits and prints in reverse.

```
ip: 723                         ip: 26
op: 327                        op: 62
```

Logic: to get reverse of 29, compose the number as 9*10+2 → 92

to get reverse of 723, compose the number as 3*100+2*10+7*1

```
void main()
{          int n, rev;
           printf("Enter 2/3 digit single number :");
           scanf("%d", &n);
           if(n<100)                         // if n<100 then it is a two digit number
                rev=n%10*10 + n/10;          // reversing 2 digit number
           else
                rev=(n%10)*100 + (n/10%10)*10 + n/100*1;      //reversing 3-digit number
           printf("reverse = %d", rev);
 }
```

Count and Boolean logics

sometimes, the count logic makes the program easy, here we check and count the input values in favor of +ve or -ve side comparison and we take final decision on count. Let us see following examples.

Note: this logic explained without using logical operators (&& , ||)

example1: Let student has 2 subjects, If he obtained>50 in 2 subjects then print “passed” or else “failed”.

```
   ip:  51   60               ip: 30    60            ip: 90   60
  op:  passed                op: failed             op: passed
```

```
logic is:  step1:  first take ‘count’ as zero   // count=0;
           step2:  scan( m1 , m2 )
           step3:  if( m1 > 50 )
                       count++;
           step4:   if( m2 > 50)
                       count++;
           step5:   if( count==2 )  printf(“passed”);
                    else   printf(“failed”);
```

---

example2: Let there are 3 values A, B, C, now finding at least one value is -ve or not?

```
  ip: 10    20    -30                ip: 10    -20    -30                ip:  10   20   30
 op: yes , -ve exist                op: yes , -ve exist                op: no , -ve exist
```

```
logic is: step1: first take ‘count’ as zero         // count=0;
          step2:  scan( A,B,C )
          step3:  if( A < 0 )
                      count++;
          step4:  if( B < 0)
                      count++;
          step5:  if( C < 0)
                      count++;
          step6:  if( count==0)   printf(“no, -ve found ”);
                  else   printf(“yes, -ve found”);
```

P1) try yourself, check given input ‘time’ values are valid or not ? (take time as 3 values for H,M,S)
if seconds and minutes must be < 59 and hours must be < 12 (do not use logical operators)

```
 ip: 10   4  30            ip:  15   4   30          ip:   5   44   30           ip: 3   4  89
op: valid inputs          op:  invalid inputs       op:  valid inputs          op:  invalid inputs
```

P2) try yourself, find given input ‘N’ value is in between 10 to 20 or not? (do not use logical operators)

```
   ip: 12        ip:  25        ip:   5
  op:  yes      op:  no        op:  no
```

Boolean or bool logic

What is bool or boolean: A well-known Scientist **George Boole** applied probability theory to solve some kind of mathematical algebraic problems(Boolean Algebra). We know, in math, probability value ranges 0 to 1. Here the value 0 represents no-hope, whereas, 1 represents 100% hope. (0.5 represents 50% hope).
These values adapted to computer science and calling them as Boolean values, used for 2-way decisions output problems. The problems like pass/fail, exist/not-exist, even/odd, prime/not-prime, etc.
We can use any other values instead of 1/0, but it is informal.
Generally, 0 is used for –ve result like failed case, whereas 1 is used for +ve result like success case.

Procedure for Boolean logic:

step1: take Boolean variable mostly with name 'bool'

step2: first, we need to take assumption of result +ve or –ve side (most of the time it is +ve side)
so take boo1=1;

step3: now check each input value in opposite side of assumption, check all values one by one.
if any input value is true then set bool to 0.

step4: finally, the result in bool in the form of 1 or 0. Now check bool and print result.

Let us see following examples

example 1) If marks of 2 subjects scanned from kb, write a program to print student is passed or failed;
If student obtained>=50 in 2 subjects then print "passed" or else "failed".

```
   ip:  51   60             ip: 30    60         ip: 90   60
  op:  passed              op: failed          op: passed

  bool=1;               // let us assume student has passed, so take 'bool' as 1. (we can take any name for bool).

 if(m1<50)              // now checking opposite side of above assumption
     bool=0;            // if this condition is true, the student failed, so set  'bool' to  0

 if(m2<50)
      bool=0;           // if this condition is true, the student failed, so set  'bool' to  0

 //  in the above code, if at least one if-condition is true then 'bool' becomes 0, it indicates that, the student failed.
 // now result is in 'bool' in the form of 1 or 0,  so check 'bool' value and print "pass" or "fail"
 if(bool==1)  printf("passed");
 if(bool==0)  printf("failed");
```

---

example 2) finding given input number(N) is in b/w 10 to 20 or not? (finding using bool logic)

```
  ip: 12                              ip: 25
 op: yes, it is                      op: no, it is not

 k=1;   // here 'k' is Boolean variable, and assume N is in b/w  10  to 20, so taking k=1
 if( N<10)
    k=0;            // here N is below 10, so not in 10 to 20,  then taking k=0

 if( N>20)
    k=0;            // here N is above 20, so not in 10 to 20, then taking k=0

 if( K==1)  printf("yes, it is in b/w 10 to 20");
 else   printf("no, it is not in b/w 10 to 20");
```

P3) try yourself, if three integers are input through the keyboard, then find at least one value is –ve or not?
try using Boolean logic (do not use logical operators && , ||)

```
 ip: 10   20   -30             ip: 10   -20   -30           ip:  10   20   30
op: yes , -ve exist           op: yes , -ve exist           op: no , -ve exist
```

P4) try yourself, check given input time (h, m, s) values are valid or not ? use bool logic.
if seconds and minutes must be <59 and hours must be <12

```
 ip:  10   4  30           ip:  15   4   30          ip:  5   44   30
op: valid inputs          op:  invalid inputs       op:   invalid inputs
```

Programs using Logical operators (&& , ||)

If marks of 2 subjects scanned from keyboard, printing student is passed or failed;
if student obtained>=50 in 2 subjects then print "passed" or else "failed".

```
 ip:  51   60              ip: 30    60          ip: 90   60
op:  passed               op: failed            op: passed
```

A) using logical operator AND (&&)

```
for example:   if (m1>=50  &&  m2>=50)  printf("passed")
               else   printf("failed");
```

B) using logical operator OR (||)

```
 for example:  if (m1<50  ||  m2<50)  printf("failed")
               else  printf("passed");
```

---

above program by taking 3 subject marks.

A) using logical operator AND (&&)

```
for example:   if (m1>=50  &&  m2>=50 && m3>=50 )  printf("passed")
               else   printf("failed");
```

B) using logical operator OR (||)

```
          if (m1<50  ||  m2<50 || m3<50 )  printf("failed")
          else  printf("passed");
```

P5) try yourself, code to find given input number(N) is in between 10 to 20 or not?

```
 ip: 12                ip:  25
op: yes, it is        op:  no, it is not
```

A) find using logical operator && (AND operator)

B) find using logical operator || (OR operator)

P6) try yourself, if three integers are input through the keyboard, then find at least one value is –ve or not?

```
 ip:  -12   34   -42            ip:   52   64   -72            ip:   62   44    42
op:  "yes, –ve exist"          op: "yes, –ve exist"           op: "no, –ve is not exist"
```

A) find using logical operator || (OR operator)

B) find using logical operator && (AND operator)

Nested-If Construct

Construction of one if-statement within another if-statement is called as nested-if. The inner and outer 'if' may have 'else' part of it. Look at the following different syntaxes of nested-if construct.

The simple form of nested-if statement is

```
    if(condition)              // outer if
    {
        if(condition)          // inner if
        {
            instruction 1;
            instruction 2;
            ---------
            instruction n:
        }
    }
```

Let us see some more examples of **nested-if** with different styles

Example 1	**Example 2**	**Example 3**	**Example 4**

if(condition)
{   if(condition)
        instruction1;
    else
        instruction2;
}
else
{   if(condition)
        instruction3;
    else
        instruction4;
}
	if(condition)
	{ if(condition)
	        instruction1;
	}
	else
	{ if(condition)
	        instruction2;
	    else
	        instruction3;
	}
		if(condition)
		{ if(condition)
		        instruction1;
		}
		else
		{ if(condition)
		        instruction2;
		}
			if(condition)
			    instruction1;
			else
			{ if(condition)
			        instruction2;
			    else
			    { if(condition)
			            instruction3;
			        else
			            instruction4;
			    }
			}

There is no specific syntax for nested-if, we can write in any permutation and combinations, besides, it can be nested as many times as we need in the program. Let us have some examples,

Demo program finding big of 3 values using nested-if style	
// simple nested-if style	**// nested-if with logical operators**

if(A>B)
{   if(A>C)
            X=A;
        else
            X=C;
}
else
{   if(B>C)
            X=B;
        else
            X=C;
}
printf("big is %d", X);
	if(A>B && A>C)
	        X=A;
	else
	{ if(B>C)
	            X=B;
	        else
	            X=C;
	}
	printf("big is %d", X);
	
	// in this way, by adding logical operators to
	nested-if, we can simplify the code.

Demo finding student is passed or not? Here student has 3 subjects, if he got>50 in all then pass or fail

// nested-if without using logical operators

```
if( A>50 )
{       if( B>50 )
        {       if( C>50 )
                       x=”passed”;
                else
                       x=”failed”;
        }
        else
            x=”failed”;
}
else
     x=”failed;
```

// opposite way comparison

```
if( A < 50 )
       x=”failed”;
else
{      if( B < 50 )
              x=”failed”;
       else
       {         if( C < 50 )
                        x=”failed”;
                 else
                        x=”passed”;
       }
}
```

By using logical operators, we can simplify the nested-if statements. But, this is not possible in all cases. The above code can be simplified using logical operator as

```
if( A>50  &&  B>50 && C>50 )
    x=”passed”;
else
   x=”failed”;
```

// opposite way comparison

```
if( A<50  ||  B<50 || C<50 )
       x=”failed”;
else
       x=”passed”;
```

Demo program finding Tax from Salary of employee

if salary<=10000 then tax is zero

if salary>10000 and <=20000 then tax is 5% on salary

if salary>20000 then tax is 8% on salary.

Following code explains how to write using 3 independent if-statements and using nested-if

```
if( salary<=10000 )
{       tax=0;
}

if( salary>10000  && salary<=20000 )
{        tax=5*salary/100;
}

if( salary>20000 )
{       tax=8*salary/100;
}
```

// nested-if style (simple and best)

```
if( salary<=10000 )
        tax=0;
else
{
        if( ~~salary>10000  &&~~ salary<=20000 )
                 tax=5*salary/100;
        else
                 tax=8*salary/100;
}
```

control comes to else-part when salary is above 10000, so no need to check salary>10000 in the above code

Checking given number in between 10 & 20 or not?

```
void main()
{       int k;
        printf("enter a number");
        scanf("%d", &k);
        if(k>=10)
        {     if(k<=20)                                   // nested if construction starts here
                     printf(" yes, it is in between 10 & 20");
              else
                     printf("no, it is above 20");
        }
       else
            printf("no, it is below 10");
}
```

Above code can be simplified using logical operators as

```
   if( k>=10  &&  k<=20 )
          printf("yes, it is in limits of 10-20");
   else
          printf("no, it is not in limits of 10-20");
```

Note: for nested-if, there is no specific style or syntax, it can be in any form, it can be in any permutation and combination, according to problem we can construct in any style. It can be nested as many times as we want. Every outer or inner if-statement may or may not have its own else-part.

Most of the problems have alternative decisions like biggest of 4 numbers. Here every else part is nested. First, we check A for big, if not then we check alternatively B for big, and then C, and then D. In this way most of the problems exist, following examples may help you how alternative decisions to be written.

Demo program finding biggest of 4 numbers. Let A,B,C,D are 4 numbers

1) writing code using 4 independent **if-statements** 2) using **nested-if** style

// using 3 independent if-statements

```
if( A>B && A>C && A>D)
{ X=A;
}

if( B>A && B>C && B>D )
{ X=B;
}

if( C>A && C>B && C>D)
{ X=C;
}

if( D>A && D>B && D>C)
{ X=C;
}
```

Note: even if first condition is true, the computer unnecessary checks all the remaining.

// nested-if style (simplified solution)

```
if( A>B && A>C && A>D )
         X=A;
else
{        if( B>C && B>D )
               X=B;
         else
         {        if(C>D)
                        X=C;
                  else
                        X=D;
         }
}
```

here, if one condition is true, then rest of the conditions are by passed(not checked). So this is best logic.

Program finding how many digits exist. Let us assume input value of N lies in between 0 to 99999

```
  ip: 234        ip: 3456      ip: 12234      ip: 3
 op: 3          op: 4         op: 5          op: 1
```

```
   // simple-if  style
    if(N<10)
          count=1;

    if(N>=10 &&  N<100)
           count=2;

    if(N>=100 &&  N<1000)
           count=3;

    if(N>=1000 &&  N<10000)
          count=4;

    if(N>=10000 )
         count=5;
```

```
   // nested-if  style
    if( N<10 )
              count=1;
    else
    {       if( ~~N>=10 &&~~ N<100 )
                    count=2;
            else
            {       if( ~~N>=100 &&~~ N<1000 )
                            count=3;
                    else
                    {       if(~~N>=1000 &&~~ N<10000)
                                    count=4;
                           else
                                    count=5;
                    }
            }
    }
```

Printing status of person using age

If age<=12   printf(“child”);

If age>12 && age<20   printf(“teenager”);

If age>=20 && age<=50   printf(“young”);

If age>50   printf(“old”);

```
void main()
{      int age;
       printf(“enter age :”);
       scanf(“%d”, &age);
       if( age<=12)
             printf(“child”);

       if( age>12 && age<20 )
             printf(“teenager”);

       if( age>=20 && age<=50 )
             printf(“young”);

       if( age>50 )
             printf(“old”);
}
```

Instead of writing four independent if-statements, it can be simplified using nested-if structure as

```
       -----------


       if( age<=12)
             printf(“child”);
       else
       {      if( age<20 )
                     printf(“teenager”);
              else
              {        if(age<=50 )
                              printf(“young”);
                       else
                              printf(“old”);
              }
       }
}
```

Finding roots of a quadratic equation

This program to display the root values of a quadratic equation given by its coefficients say a, b and c. Here a, b and c are input numbers.

Logic: First find the value of (b^2-4ac), let it is x

- if x<0 then print "roots are imaginary"
- If x == 0 print "roots are equal", the root1 and root2 are is **-b/(2*a)**
- If x > 0 then print root1 = (-b+sqrt(x))/(2*a), root2 = (-b-sqrt(x))/(2*a)

```
    float a, b, c, x, root1, root2;
    printf("enter a, b, c :");
    scanf("%d %d %d", &a, &b, &c);
    x=b*b-4*a*c;
    if(x<0)
        printf("roots are imaginary");
    if(x==0)
        printf("roots are equal, roots are %f", -b/(2*a));
    if(x>0)
    {   printf("Root1 = %f", (-b+sqrt(x))/(2*a));
        printf("Root2 = %f", (-b-sqrt(x))/(2*a));
    }
}
```

simplifying using nested-if structure

```
 if(x<0)
     printf("roots are imaginary");
else
{     if(x==0)
          printf("roots are equal, roots are %f", -b/(2*a));
      else
      {   printf("Root1 = %f", (-b+sqrt(x))/(2*a));
          printf("Root2 = %f", (-b-sqrt(x))/(2*a));
      }
}
```

About braces: observe if-block have 2 statements in the following example

complete the code to find age group and normal weight of a person as given below

if age<=12 then say child and normal weight is 20

if age>12 and age<=19 then say teenager and normal weight is 40

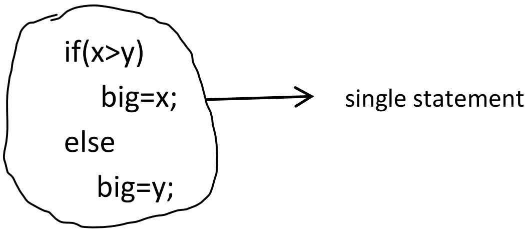
if age>19 and age<=50) then say younger and normal weight is 60

if age>50 then say old-age and weight is normal 70

```
    if( age<=12)
    {         X="child"                // braces needed because two instructions exist here.
              weight=20;
    }
    else
    {         if(age<=19)
              {        X="teenager";
                       weight=40;
              }
              else
              {        if( age <= 50 )
                       {       X="younger";
                                -------
                       }
                       -------
              }
    }
    printf(" age is  %s , normal weight is %d " ,  X, weight );
```

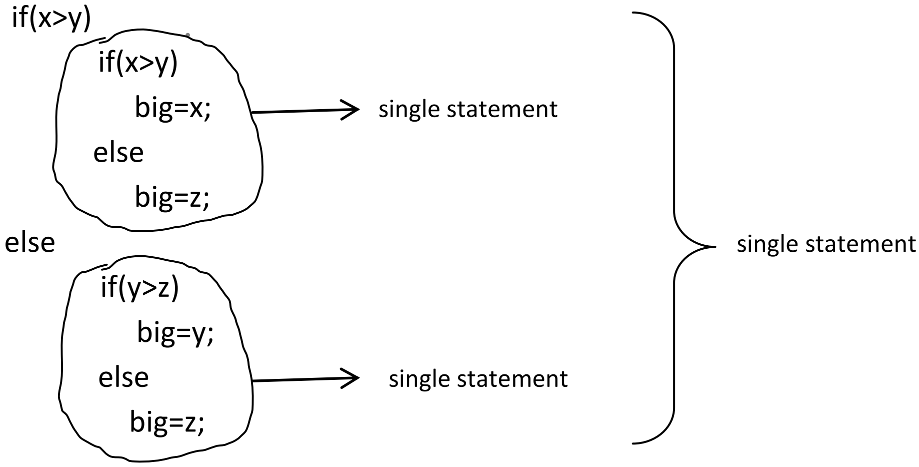
More about Pair of Braces { }

*** As per C language syntax, if-else control statement can be taken as single-compound-statement even though they seem to be two, for example, the following if-else can be taken as single.**



```
if(x>y)
    big=x;
else
   big=y;
```

*** As inner if-else statement is single, braces are not required to make into single for outer-if.**
Based on this theory, we can remove braces for outer-if also. Let us see following example

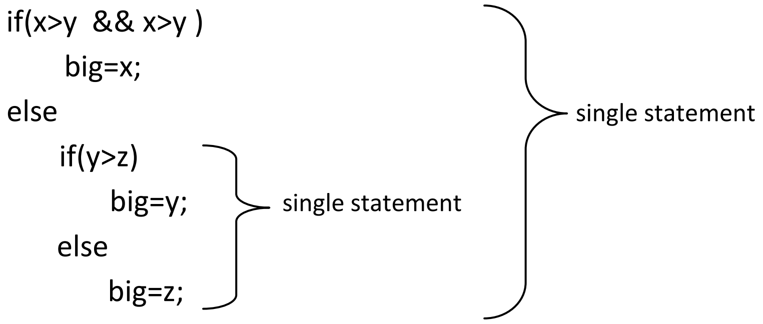


```
if(x>y)
   if(x>y)
      big=x;
   else
      big=z;
else
   if(y>z)
      big=y;
   else
     big=z;
```

In this way, in C, any control statement can be taken as single compound statement including nested-if, thus above entire if-statement can be taken as single-compound-statement.

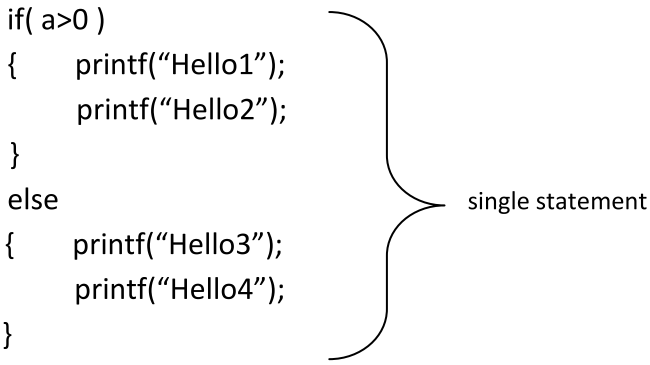
But sometimes, some people used to give braces for clarity purpose even though they are not required.

Example-3



```
if(x>y  && x>y )
     big=x;
else
    if(y>z)
        big=y;
    else
        big=z;
```

Example-4



```
if( a>0 )
{      printf("Hello1");
       printf("Hello2");
 }
else
{      printf("Hello3");
       printf("Hello4");
}
```

Following examples explains the code with and without braces

Demo program finding how many digits exist. Let us assume input value of N lies in between 0 to 99999

ip: 234	ip: 3456	ip: 12234	ip: 3
op: 3	op: 4	op: 5	op: 1

with braces

```
if( N<10 )
       count=1;
else
{     if(  N<100 )
           count=2;
      else
      {          if(  N<1000 )
                      count=3;
                 else
                 {          if( N<10000 )
                                   count=4;
                            else
                                   count=5;
                 }
      }
}
printf("count of digits is %d" , count);
```

without braces

```
if( N<10 )
        count=1;
else
       if(  N<100 )
             count=2;
      else
             if(  N<1000 )
                         count=3;
             else
                        if( N<10000 )
                                count=4;
                         else
                                count=5;


printf("count of digits is %d" , count);
```

Demo program finding biggest of 4 numbers. Let A,B,C,D are 4 numbers

with braces

```
if( A>B && A>C && A>D )
          X=A;
else
{       if( B>C && B>D )
                X=B;
       else
       {       if(C>D)
                     X=C;
               else
                     X=D;
       }
}
```

without braces

```
if( A>B && A>C && A>D )
       X=A;
else
       if( B>C && B>D )
              X=B;
       else
              if(C>D)
                    X=C;
             else
                    X=D;
```

If-else-if Ladder style

This is not a special control statement; it is one kind of written style of **nested-if** when several alternative decisions exist. Normal nested style leads to heavy indentation when more alternative exist.
So it is rearrangement of **nested-if** like a straight-line unlike crossed-line as shown in the below syntax.
This is applied, when needed to process one decision from several alternative decisions. This structure is also called '**if-else-if**' staircase because of its appearance. The main advantage of this structure is, easy to code, understand, and debug. The syntax is

Normal nested-if

```
if(condition1)
      instruction 1;
else
{    if(condition 2)
             instruction 2;
     else
     {
             if(condition 3)
                     instruction 3
             else
             {
                     ---------
                     ---------
             }
     }
}
```

if-else-if ladder style

```
if(condition 1)
          instruction 1;
else if(condition 2)
          instruction 2;
else if(condition 3)
          instruction 3;
else if(condition 4)
         instruction 4;
      ------------
     ------------
     -------------
else
     instruction n;
```

By removing un-necessary braces and arranging all nested-if statements **in one column** makes the if-else-if ladder style

Here conditions are evaluated from top to down, as soon as a true condition is found, the instruction associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, the final '**else-part**' will be executed. That is, if all other condition tests fail, the final '**else**' instruction-n will be performed. If the final '**else**' is not present, no action takes place.

Although, the above '**if-else-if**' ladder is technically correct, it can lead to bit confusion to some programmers. For this reason, the '**if-else-if'** ladder style also written as

```
If(condition 1)
      instruction 1;
else
if(condition 2)
      instruction 2;
else
if(condition 3)
      instruction 3;
-----------
-----------
else
      instruction n;
```

Observe here, the nested statements are started in a new line unlike same line as above said.

Above program using if-else-if ladder style (finding biggest of 4 numbers)

<table>
<tr><th>Normal nested-if</th><th>if-else-if ladder style</th></tr>
<tr><td>if(a>b && a>c && a>d)
    x=a;
else
  if(b>c && b>d)
    x=b;
  else
    if(c>d)
      x=c;
    else
      x=d;
printf(“\n biggest=%d”, x)</td><td>if(a>b && a>c && a>d)
  x=a;
else if(b>c && b>d)
  x=b;
else if(c>d)
  x=c;
else
  x=d;
printf(“ \n biggest=%d”, x);</td></tr>
</table>

Accepting student marks of 3 subjects & printing result

If student got <35 in any subject then print “F grade”

If student got average>=60 print “A grade”

If student got average>=50 && <60 print “B grade”

If student got average <50 print “C grade”

<table>
<tr><th>Normal nested-if style</th><th>if-else-if ladder style</th></tr>
<tr><td>void main()
{   int m1, m2, m3;
     float avg;
     printf(“\n Enter 3 subject marks:”);
     scanf(“%d%d%d”, &m1, &m2, &m3);
     avg=(m1+m2+m3)/3;

     if(m1<35||m2<35||m3<35)
         printf(“\n F grade”);
     else
     {   if(avg>=60)
             printf(“A grade”);
          else
          {   if(avg>=50)
                  printf(“B grade”);
               else
                  printf(“C grade”);
          }
     }
}</td><td>void main()
{   int m1, m2, m3;
     float avg;
     printf(“\n Enter 3 subject marks:”);
     scanf(“%d%d%d”, &m1, &m2, &m3);
     avg=(m1+m2+m3)/3;

     if(m1<35||m2<35||m3<35)
         printf(“\n F grade”);
     else if(avg>=60)
         printf(“A grade”);
     else if(avg>=50)
         printf(“B grade”);
     else
         printf(“C grade”);
}</td></tr>
</table>

The above left side given nested-if cross aligned, however it is readable (because few decisions exist). If the programmer did not take care while aligning the code properly, it makes un-readable and difficult to debug. Hence, if-else-if ladder style is always best. If more than 10 alternative decisions exist, the normal nested-if seems to be awkward and confusion, so the ladder-style is the alternative under any case.

Finding fine amount for a late returned book at a library

The C-Family library charges a fine for every book returned late

- For first 10 days late, the fine is 5/- rupees
- For 11-20 days late, the fine is 10/-
- For 21-30 days late, the fine is 20/-
- For >=31 days late, 1/- per every day;

Write a program to accept the number of days late and display fine or the appropriate message.

These two statements are same with little difference, you can follow whichever you like

```
void main()
{    int daysLate, fine;
     printf("the number days late :");
     scanf("%d", &daysLate);
     if( daysLate <=10 )
            fine=5;
     else
     if( daysLate<=20)
            fine=10;
     else
     if( daysLate<=30)
           fine=20;
     else
           fine=daysLate;
     printf("fine amount = %d", fine);
}
```

```
void main()
{    int daysLate, fine;
     printf("the number days late :");
     scanf("%d", &daysLate);
     if(daysLate <=10)
               fine=5;
     else if(daysLate<=20)
               fine=10;
     else if(daysLate<=30)
               fine=20;
     else
               fine=daysLate;

     printf("fine amount = %f", fine);
}
```

Finding state of a person

This program accepts age of a person and prints his stage of life.

- age <=12 → Child
- age>12 and <=19 → Teenager
- age>19 and <=50 → Youngman
- age>50 → old

```
if(age<=12)
    printf("Child");
else if(age<=19)
    printf("Teenager");
else if(age<=50)
    printf("Young man");
else
     printf(" old ");
}
```

Finding no.of days in a month

This program accepts month & year in a date, and find no.of days in a month

```
    ip: 2   2011              ip: 4   2010              ip: 12    2010
   op: 28 days               op: 30 days               op:  31 days
 void main()
 {    int  m, y, days  ;
      printf("enter month & year  :");
      scanf("%d %d",  &m, &y);
     if(m==2 )
     {        days=28;
              if(y%4==0)       // if leap year then month has 29 days.
                   days=29;
     }
     else if( m==4 ||  m==6 || m==9 || m==11 )    // april  , june , setp , nov has 30 days
              days=30;
     else
             days=31;
     printf(" no.of days in a month = %d", days);
 }
```

Checking given date is valid or not?

This program checks the given date is valid or not?

```
   ip: 29/2/2011                 ip: 31/4/2010              ip: 1/12/2010
 op: invalid date              op: invalid date            op: valid date
```

Logic: if month & day is not in between 1-12 & 1-31, report "invalid date"

If day is in between 1-31, check the day according to days of the month and print the result

```
 void main()
 {    int d, m, y, bool;
     printf("enter date :");
     scanf("%d%d%d", &d, &m, &y);
     bool=1;                                    // let us take date is valid
     if(m<1 || m>12 || d<1 || d>31)
              bool=0;
     else  if(m==2 && y%4==0 && d>29)           // for February and leap-year checking
              bool=0;
     else  if(m==2 && y%4!=0 && d>28)           // for February and non-leap year checking
              bool=0;
     else  if((m==4 || m==6 || m==9 || m==11)  && d>30)   // for 30-days month checking
              bool=0;

     if(bool==1) printf("date is valid");       //finally check the bool and print the output
     else printf("date is in-valid");
 }
```

Incrementing given date by 1 day (let the input date is valid-date)

```
  ip: 29  2  2012             ip: 31  12  2010              ip: 21   1   2010
op:  1   3   2012           op: 1  1  2011               op: 22    1   2010
void main()
{    int d, m, y;
     printf("enter a valid date :");
     scanf("%d%d%d", &d, &m, &y);
     d++;             // incrementing day by 1-day
                      // after incrementing, if 'day' crosses the end of month limits, then shift to next month
    if(m==2)
    {       if(y%4==0 && d>29)
            {       d=1;
                    m++;
            }
            else if( y%4!=0 && d>28)
            {       d=1;
                    m++;
            }
    }
    else  if((m==4 || m==6 || m==9 || m==11)  && d>30)
    {       d=1;
            m++;
    }
    else if(d>31)
    {       d=1;
            m++;
            if(m==13)
            {       d=1;
                    y++;
            }
    }
     printf("date after incrementing is %d-%d-%d", d, m, y);
}
```

Conclusion: The choice of usage of **if**, **if-else**, **nested-if**, **if-else-if** ladder is left to the programmer. There are no specific situational suggestions which statement is to be used when and where. However, ensure that, the written code should be correct, clear, and easy to understand. While writing code, the programmer should remember that, the C possesses, there is always a good structured alternative control statement exist whenever complexity rises with one control statement.

if - else linking

eg1) In C, if & else are 1:1 type, one else-statement always linked to only one if-statement in the program, we cannot write one else-statement for two or more if-statements, following example explains it.

```
if( marks1>50)
    if(marks2>50)
        if(marks3>50)
                printf("passed");
else
   printf("failed");
```

in above , the else-part joins only to the last if-statement(marks3>50), so we can't write one else-part to many if-statements. The solution is, we need to write else-part to every if-statement, this is as

```
if(m1>=35)
    if(m2>=35)
        if(m3>=35)
            printf("passed");
        else
            printf("failed");
    else
      printf("failed");
else
    printf("failed");
```

---

eg2) In some situations, where the **'else'** is available to the outer-if and not to the inner-if , in that case, the inner-if should be enclosed within braces { }, otherwise compiler links outer-else body to the inner-if. Observe the following example

```
if(x==1)
    if(y==2)
         printf(" x is 1, y is 2 ");
else
    printf("x is not 1");
```

Here, the programmer wrote else-part to the outer-if, but compiler links to the inner-if, to avoid this logical error, enclose inner-if within braces { }. This is as follows

```
 if(x==1)
 {     if(y==2)               // now this 'if' has no else part of it
           printf("x is 1, y is 2");
 }
else
     printf("x is not 1");
```

About Boolean value

In C, any relational or logical expression results either 1 or 0 to determine true or false respectively. Suppose the relation a<b is true then the operator '<' gives the result 1, otherwise 0. These are known as Boolean values in computer science.

But, In C, in place of test condition, we can use any arithmetic expression; in such case, the resultant value of the expression decides the success/failure of the condition. If the expression results any non-zero number then that condition is said to be true, otherwise false.

That means **any non-zero value (not necessarily 1) is considered to be true, whereas zero is false**.

Observe the following program output

```
void main()
{   int x=0, y=1, z=10;

    if( x+y )                       // x+y → 0+1 → 1 → true
        printf(" White+Yellow ");
    if( x*y )                       // x*y → 0*1 → 0 → false
       printf(" White * Yellow ");
    if(  x  )                       // if( x ) → if(0) → if(false)
      printf(" White  ");

    if(  y  )                       // if( y ) → if( 1 ) → if( true )
      printf(" Yellow  ");

    if( 1 )                         // if( 1 ) → if(  true )
      printf(" Red  ");

    if( 5 )                         // if( 5 ) → if( true )
      printf(" Green  ");

    if( -5.34 )                     // true
       printf(" Blue  ");

    if(  0 )                        // false
      printf(" Black  ");

   if( !0 )                         // ! false → true
      printf(" Black  ");

    if(  !5 )                       // ! true → false
      printf(" Black  ");


    if(x&&y)
        printf("Black");            // if( x&&y ) → if( 0 && 1 ) → if(false && true)  → if( false)

    if(x||y)
       printf("Grey");              //  if( x||y ) → if( 0 || 1 ) → if(false || true)  → if( true)
}
```

About logical operators

When more conditions exist then they must combine using logical operators, otherwise, program may show wrong output or syntax error. Let us observe the output of following program.

```
void main()
{    int k = -5;
     if(0 < k <10)
        printf("\n  k is in between 0 & 10");
     else
        printf("\n k is not in between 0 & 10");
}
```

output: "k is in between 0&10" → wrong output, the evaluation of if-statement is as follows

```
➢ if(0< -5 <10)        //  0 < -5 → false → 0
➢ if(0<10)             // 0 < 10 → true → 1
➢ if(1)
➢ if(true)
```

Here the test condition made true hence if-block executed. When two or more relations exist, they must be composed by logical operators. The correct code is:

```
void main()
{     if(0 < k  && k < 10)
           printf("\n k is in between 0 & 10");
      else
           printf("\n  k is not in between 0 & 10");
}
```

The condition is evaluated as if(0<k && k<10) → 0<-5 && -5<10 → 0&&1 → 0 (false)

Null instruction caused by semicolon(;)

The extra semicolon(;) in the program goes to null-instruction, that is, if anybody by mistake is given extra-semicolon then it forms as null-instruction. For example,

```
a=10;
b=20;
    ;            // observe the extra semicolon
c=30;           // observe the one more extra semicolon
d=40;
```

here we have 5 instructions not 4, because extra semicolons considered as null-statement by the compiler. This extra semicolon does nothing in this example.

Null instruction as if-statement's body

Observe the following example

```
    int A=10, B=20;
     if(A<B)
           ;      // null-instruction
     else
         printf(" sky is blue ");

Output: no output displayed
```

```
    int A=10, B=20;
     if(A>B)
            ;
     else
        printf(" sky is blue ");

output: sky is blue
```

```
    int A=10, B=20;
     if(A>B)
              ;
     else
            ;
     printf(" sky is blue ");

output: sky is blue
```

Do not terminate any control statement's header with the semicolon (;), otherwise, head and its body get separated and the semi-colon will be formed as null-instruction as body. For example

```
      A=5;
      if( A<0) ;
           printf("A is -VE");
      prntf("\nGood Night");

op:  A is –VE
     Good Night
```

→

```
      A=5;
      if( A<0)
              ;
      printf("A is +VE");
      printf("\nGood Night");
```

```
void main()
{     int a=10;
       if(a<50)
           printf(" I love C ");
        else ;
          printf(" I marry C ");
}
Output: I love C, I marry C
```

→

```
 void main()
  {    int a=10;
      if(a<50)
           printf(" I love C " );
      else
            ;
      printf(" I marry C ");   // this is not in else-part
  }
```

```
void main()
{     int A=10,B=20;
      if(A>B) ;
          printf("A>B"); ➔ compile time  error
      else
          printf("A<=B");
 }
```

→

Output: compile time error
unfortunately here if-block contained two instructions.
1. Null instruction by semicolon
2. printf("A>B");

when two or more instructions exist between if & else then they must enclosed by braces, if not, the compiler raises an error.
(Of course in our view it is only one)

Check out errors, if no errors then guess the output values

```
 void main()
 {        if(!0)
              printf(“test passed”);
          else
              printf(“test failed”);
 }
```

```
void main()
{  if( .00001 )
          printf(“test passed”);
   else
          printf(“test failed”);
}
```

```
void main()
{    int k=0;
     if(k)
        printf(“hello”);
     else
             ;
}
```

```
 void main()
{   int k=20;
    if( k > 10 )
        printf(“I am in Right”);
    else  ;
       printf(“I am in Wrong”);
}
```

```
void main()
{   int k=1000;
    if(10 > k < 35)
         printf(“ I love C”);
    else
          printf(“ I marry C”);
 }
```

```
void main()
{
     if(-1.4)

      else
          printf(“test failed”);
}
```

```
void main()
{     if(-1.4)
          ;
     else
        printf(“test failed”);
}
```

```
void main()
{   int k=1;
    if(k>10)
      printf(“k is +ve”);
      printf(“k is > 10”);
 }
```

```
void main()
{   int k=1;
    if(k>10)
      printf(“ k is +ve”);
      printf(“ k is > 10”);
    else
      printf(“ k is <=10”);
}
```

```
void main()
{   int a=100,b=1;
    if( a ||b )
       printf(“Good Morning”);
    else
       printf(“Good Night”);
 }
```

```
void main()
{   int a=10, b=5, c, d;
    c=a>3;
    d=a>6 && b!=10;
    printf(“c= %d , d=%d”,c ,d);
}
```

```
void main()
{   int a=30, b=40,c;
    c =(a!=10 && b==50);
    printf(“b=%d, c=%d”, b ,c);
}
```

```
void main()
{  int a=20, b=230,c;
   c =(a==100||b>130);
   printf(“c=%d”,c);
 }
```

```
void main()
{   int k=1;
    if(k < 0)
    else
       printf(“k is +ve”);
 }
```

```
void main()
{   int k=1;
    if(k < 0)
         printf(“k is -ve”);
    else
   }
```

```
void main()
{     if( 0 )
          printf(“A”);
      else
          printf(“B”);
}
```

```
void main()
{     if( 10 )
          printf(“A”);
      else
          printf(“B”);
}
```

```
void main()
{     if( 0-1 )
          printf(“A”);
      else
          printf(“B”);
}
```

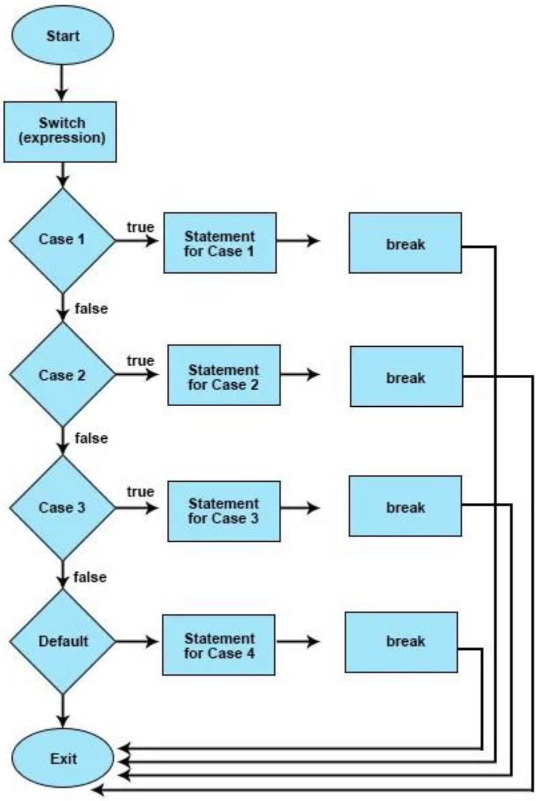
```
void main()
{     if( 10 ) ;
          printf(“A”);
      else
          printf(“B”);
}
```

```
void main()
{     if( 10 ) ;
          printf(“A”);
 }
```

Switch Statement

Switch is another conditional control statement used to select one option from several options based on given **expression** value. This is an alternative to the **if-else-if** ladder when comparisons found against with constants. The **if-else-if** seems to be lengthy and awkward when more comparisons exist. **Switch** is a good alternative as its construction is simple and looking comfortable even if more cases found or nested. However it does not support in all situations in which **if-else-if** supports. ie, it compares only against with integral constants. In **switch** construct, the instructions are divided into case blocks based on logical condition and any number of case blocks can be defined. Observe the following syntax and flow chart.

```
switch(expression)              // ← switch header
{
        case constant-1:   instruction-1;
                           instruction-2;
                             .....
                           break;
        case constant-2:   instruction-1;
                           instruction-2;
                            ......
                           break;
        .........
        .........
        default:           instruction-1;
                           instruction-2;
                           ......
}
```



* The **expression** in switch() header should be an integral type value such as **char, int, long int**.

* The **expression** value compares with constant-1, constant-2,… etc, if anyone matches then the control jumps into associated case block and executes all the instructions within it, later comes out by **break**.
The **break** throws the control out of switch statement. In other words, keyword **break** used to come out of switch after execution of selected case block. However, **break** is an optional statement. If it is not present at end of case-block, the control enters into next case block even though its constant value differs.

* The case constants may not be in ascending or descending order. The two or more case constants may be associated with single block.

* The **default** block executes when no case is matched. The default is optional block and generally it is used to handle input errors.

* In case block of switch, we can write any other control statement, sometimes it can be nested

* remember, this is an alternative statement of if-else-if ladder style, but this is used only when comparisons made against with ***constants***. This is the limitation of this statement.

Printing single digit value in true English words

input: 3
output: three

```
void  main()
{         int digit;
          printf("Enter any single digit:");
          scanf("%d",&digit);
          switch(digit)
          {      case 0:  printf("zero");
                           break;
                 case 1:  printf("one");
                          break;
                 -------
                 case 9:  printf("nine");
                          break;
                default:  printf("Invalid input");
          }
}
```

- Here the value of **digit** is compared with given case constants 0, 1, 2, 3,… one by one, if any case matches then the corresponding printf() prints the English word.
- If no case matches with the given value, then the **default** block executes and we get "Invalid input". Generally, **default** block used to handle errors like "invalid-input" and other things.
- Suppose the input is 4, then the "case 4:" matches and executes printf("four"). Later control comes out when break gets executed.
- The above program can be written using if-else-if ladder form as shown below.

```
void main()
{         int k ;
          printf("Enter a digit:");
          scanf("%d",&k);
          if(k==0)
               printf("zero");
          else if(k==1)
               printf("one");
          else if(k==2)
               printf("two");
          ……
          else if(k==9)
               printf("nine");
          else
               printf("Invalid input");              // like default block
}
```

The above two programs generate exactly same result. However, as you might notice that the first program is easier to write, understand and modify. Structure of switch statement is more readable even if it is nested 2 or 3 times.

Demonstration of break usage in switch

Note: If break is not given at the end of case block, the control enters into next case block even though it conveys different case-constant-value.

```
void main()
{    int a;
    printf("Enter a number 10/20/30:");
    scanf("%d", &N);

    switch(N)
    {      case 10:  printf("red");           // observe "break"  not given
           case 20:  printf("green");
                     break;
           case 30:  printf("blue");
           default:  printf("black");
    }
}
```

```
if  input: 10  → red  green      (output  of  case-10 and case-20)
if  input: 20  → green           (output  of  case-20 )
if  input: 30 → blue  black      (output of  case 30 and default )
if  input: 40 → black            ( output of default)
if  input: 50 → black            ( output of default )
if  input: 0   → black           ( output of default )
```

Finding number of days in a given month date

This program scans month & year in a date, and prints no.of days in that month

```
void main()
{       int m,y;
        printf("enter month & year  :");
        scanf("%d%d", &m, &y);
        switch(m)
        {       case 2:  days=28;
                         if( y%4==0 )
                              days=29;
                         break;
                case 4:
                case 6:
                case 9:
                case 11:  days=30;
                          break;
                default : days=31;
        }
        prinstf("no.of days is %d", days);
}
```

Here case blocks 4, 6, 9, 11 are associated with only one instruction, days=30. Thus two or more cases can be associated with one block of code.

Testing whether a given alphabet is vowel/consonant

Note: two or more cases may associate with one block of instructions.

```
void main()
{       char ch;
        printf(“Enter any lower case alphabet:”);
        scanf(“%c”,&ch);
        switch(ch)
        {   case ‘a’:
            case ‘e’:
            case ‘i’:
            case ‘o’:
            case ‘u’:  printf(“It is a vowel “);  break;
            default:   printf(“It is a consonant”);
        }
}
```

Here all case blocks associated with only one instruction, which is printf(“it is a vowel”); Thus, for any input vowel, the same message “It is vowel” is displayed. Otherwise, we’ll get the “It is a consonant”.

Conditional operator (?)

This is the simple conditional operator used to process one instruction from a given two instructions. It is an alternative to the simple **if-else** construct when its body consists of single instruction. It works like inline version of **if-statement** (single line). This conditional operator also called ternary operator since it takes three operands.

The general syntax is: **condition? expression1: expression2;**

here, the condition is true then the **expression1** is evaluated (executed) otherwise **expression2** is evaluated.

eg1: big = X>Y ? X : Y; this is equal to: if(X>Y) big=X; else big=Y;

eg2: X>Y ? printf(“X is biggest”) : printf(“Y is biggest”);

eg3: diff = A>B ? A-B : B-A ; Here, if A**>B** is true then A**-B** is assigned to **diff,** otherwise, B**-A** is assigned.
This is equivalent to

```
if(a>b)  diff=a-b;
else  diff=b-a;
```

The ternary operator can be nested, but it makes confusion to the programmer, therefore, it is suggested to use only when two way decisions exist and their expressions are simple.

Let us see the following example, how nesting makes complexity to the programmer. Program prints the biggest of three numbers, which are located in variables a, b, c.

```
void main()
{    int a,b,c;
     printf(“\n  Enter any three numbers:”);
     scanf(“%d%d%d”, &a,&b,&c);
     a>b ? (a>c ? x=a : x=c) : (b>c ? x=b : x=c);      // observe the complexity in nested usage
     printf(“Big = %d”, x);
}
```

In the above program, as conditional operator nested, it seems to be complex and unreadable. So when nesting takes place then it is better to use **if- statement**.

goto Statement

It is used to jump the control from one location to another location within a function. This unconditional control statement diverts the control to a specified location where the label is mentioned. Using this, the control can be moved to forward or backward or in any order in the function.

The early high level languages such as ALGOL, FORTRAN were influenced by a combination of **if & goto** statements. The goto was considered to be a powerful statement as any problem (program) can be solved using this. But program becomes complex, if the code is big and many **goto** statements are used.

As a result this statement fell out of favor some years ago, hence several alternative rich set of control statements are provided in modern languages such as switch, while, for, do-while, break, continue, and return. However, for some kind of situations like getting out of nested loops, to solve recursive like programs using loops, …etc. We cannot get a solution using these rich set, so, **goto** occasionally has its uses.

Syntax: goto **label**;   // Here, the **label** is a valid identifier followed by a colon, indicates a location to where the control has to be jumped using goto. Label name follows the rules of a variable name.

```
void main()
{    printf(“India\n”);
      goto end;                    //observe here, the control will be transferred to “end”
     printf(“American\n”);
     printf(“Australia\n”);
     end:                          // here ‘end’ is a label
    printf(“Russia\n”);
    printf(“Japan\n”);
}
```

```
Output:  India
         Russia
         Japan
```

When “goto end” is executed, the control simply jumps over to label “end” and follows next instructions.

Another demo program , how the control jumps using goto

```
void main()
 {       printf(“Red\n”);
         goto one;
two: printf(“Green\n”);
         printf(“Blue\n”);
         printf(“Yellow\n”);
         goto end;
one:  printf(“White\n”);
         printf(“Cyan\n”);
         goto two;
end:
}
```

By observing the above code, we can say that, goto is a powerful statement as control can be moved to any place but it led to unreadable and complex because of many goto used.

Demo program for identifier “label”

The identifier **label** is not like a variable and it does not occupy any space in machine code file, instead it works like a line indicator in the program. Generally, after compilation of a program, every instruction is associated with a line number in the machine code, it tells the sequence of instructions in the program. Let us see a program, how it creates line numbers in the machine code file.

Code with labels	**Code after compilation**

void main()
{ printf(“Red”);
goto one;
two: printf(“Blue”);
goto end;
one: printf(“Green”);
goto two;
end: printf(“end of program”);
}
	void main()
	{ line1: printf(“Red”);
	line2: goto line5;
	line3: printf(“Blue”);
	line4: goto line7;
	line5: printf(“Green”);
	line6: goto line3;
	line7: printf(“end of program”);
	}

Unconditional control transfer

```
main()
{    wish:                                    // line1
         printf(“Hello, how are you?\n”);     // line2
     goto wish;
}
```

In this example ‘**wish’** is the label name, which specifies the line number for the goto statement, here it is line1 .When the ‘**goto wish’** executes, resulting diversion of control back to the location **wish**. Here, the execution of printf() repeats endlessly.

```
Hello, how are you?
Hello, how are you?
Hello, how are you?  ….
  :
```

(To stop or kill the indefinite loop of this program, we have to press the keys **control +break** or **control+c** (in Dos), **alt+control+del** (in Windows) or other keys provided by o/s)

As we got to know that, goto is an unconditional (condition less) control statement, therefore, it is always association with **if-statement** to provide the conditional control transfer.

The following program prints “Welcome” message 10 times

```
void main()
{       int counter=1;               //take a counter with 1, for counting 10 times
        wish:
        if( count<=10)
        {      printf(“Welcome to C-Family, Vijayawada\n”);
               counter++;              // increment the counter
               goto wish;             // repeat until counter <=10
        }
}
```

Here, in this program, the variable **counter** is used to count the number of times the printf() is repeated. After completing 10 cycles, the given **if-condition** will fail and thus the goto statement is skipped, which turns to end of the program.

Program displays the given multiplication table up to 10 terms.

If input is 8

then output is:

```
        8*1=8
        8*2=16
        8*3=24
        ---
```

```
void main()
{     int i=1;            // take a counter variable 'i' and set to 1
      int n;              // 'n' holds the table number
      printf("\nEnter the table number:");
      scanf("%d",&n);
      nextRow:            //  setting label for next cycle
      if( i<11)
      {    printf("\n %d  *  %d  = %d", n, i, n*i);    //  printing terms
           i++;                  //  incrementing 'i' to generate next term
           goto nextRow;      // go back to print next term
      }
}
```

Program to print 1 to 20 tables (extension to above program)

```
void main()
{        int n=1;          //  'n' for table-numbers
         int  i;           //   'i'  to print 1-10 terms of each table
         nextTable:
         if( i<21)
         {        i=1;
                  nextRow:
                  if( i<11)
                  {        printf("\n %d  *  %d  = %d", n, i, n*i);   //  printing terms
                           i++;                //  incrementing  'i' for generate next term
                           goto nextRow;   // go back to print next term
                  }
                  n++;     // incrementing n for next table
                  goto nextTable;
         }
}
```

Control Looping Statements

In programming, it is common process to repeat instructions more than one time. Using goto statement, it can be achieved, however, it is an unstructured control transfer statement, which will cause a severe programming complexity upon nested usage or if their number increases. To eliminate such complexity, C provides some structural control statements for looping such as while, for and do-while.
These 3 structures manage the execution sequence over some set of instructions repeatedly until a certain condition is satisfied

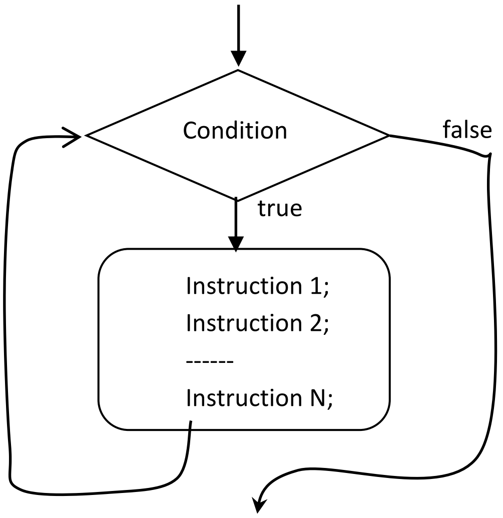
These loop control statements are used to execute some set of instructions repeatedly until a given test condition is attained. Each loop structure has its own style and provides a convenience to the user. The user can select any loop control statement in the program according to the requirement and convenience and there are no specific situational suggestions, which loop to use when and where.

All loop control statements are attached with a test-condition and a body. This body contains one or more instructions, which are to be executed repeatedly until the test condition becomes false. We can write any type of instructions or even another control statement with in this body.

The while loop

This is a simple loop statement with a terminating condition and associated block of instructions. Syntax is,

```
while (test condition)
{
      instruction 1;
      instruction 2;
      -------
      -----
      Instruction N;
}
Instruction N+1;
instruction N+2;
   ---
   ---
```



In the while loop, initially, the **test-condition** is checked and if it is true, the control transfers into the loop-body and process the instruction-1, instruction-2,… and instruction-N. Later, control jumps back to the beginning of loop for next cycle i.e., again the **test-condition** is checked and if it is true then the control re-enters into the loop-body and executes all instructions within it. This process continues as long as **test-condition** becomes false.

If the **test-condition** becomes false then the control jumps out of loop-body and follows the bottom instructions “instruction n+1, instruction n+2 ….”

If loop body has only one instruction, the pair of braces {} are optional. This is as said in if-statement.
If loop-body has more instructions exist, they must enclose by pair of braces{}.

The **test-condition** must fail after certain number of times. Otherwise, it goes to infinite loop.
Don’t terminate loop header with semi-colon, otherwise, the head & body gets separated.

The following examples explain the different formats of while-loop usage

Printing 1 to 10 natural numbers using while loop (op: 1 2 3 4 5 6 7 8 9 10)

step1. take ‘i’ as loop variable with 1 as starting value, here ‘i’ works as counter for 1 to 10

step2. print ‘i’ value as output

step3. increment ‘i’ by 1

step4. repeat step2 and step3 until i<=10

step5. stop.

```
void main()
{     int i=1;                                  // initialization of loop variable with 1
      while(i<=10)                              // loop begins here
      {     printf(“%d  ”, i );                 // printing ‘i’ value on the screen
            i++;                                // incrementing ‘i’ by 1
      }
      printf(“\n this is end of loop ”);
}
```

Generally, programmers always take loop variable name as **<u>‘i’</u>** , because it shows index(i) value.

Printing 1 to 10 and 10 to 1 using while loop

Write two loops, loop after loop, first loop prints 1 to 10, and second loop prints 10 to 1

```
void main()
 {     int i=1;                                 // initialization of loop variable with 1

       while(i<=10)                             // loop begins here
       {   printf(“%d  ”, i);                   // printing ‘i’ value on the screen
           i++;                                 // incrementing ‘i’ by 1
       }

        i=10;
       while(i>=1)                              // loop begins here
       {   printf(“%d  ”, i);                   // printing ‘i’ value on the screen
           i--;                                 // decrementing ‘i’ by 1
       }
}
```

Let see what happens if loop header terminated by semicolon (;) ?

Do not terminate the loop header with semicolon, otherwise the head and body gets separated and an empty loop is formed with a null instruction. The semicolon forms as null instruction as loop body. The following example gives the clarity how the semicolon formed as null instruction as loop body.

Observe semicolon at the end of loop-header	**Let us imagine this semicolon in the next line and how it forms as null instruction as loop body**

i=1;
while (i<=10) **;**
{
printf(“%d “, i);
i++;
} →
	i=1;
	while(i<=10)
	;
	{
	printf(“%d “, i);
	i++;
	}
	this loop goes infinite, because ‘i’ is not incrementing towards to 10. The empty loop repeats again and again

Printing numbers between two given limits

ip: 14 45

op: 14 15 16 17 18 …..45

step1: take **lower** & **upper** as input variables and scan data from KB(keyboard)

step2: take **‘i’** as loop counter and initialize with **lower** limit (i=lower)

step3: print ‘i’ value

step4: increment ‘i’ by 1, to generate next number

step5: repeat step3 and step4 as far as i<=**upper**

step6: stop.

```
void main()
{   int lower, upper, i ;
    printf(“\n enter lower and upper limits :”);
    scanf(“%d%d”, &lower, &upper);
    i=lower;                       // begins ‘i’ with lower limit
    while(i<=upper)                // loop to repeat up to upper limit
    {   printf(“%d\t “, i);
        i++;
    }
}
```

Printing 1, 10, 100, 1000, 10000, 1000000 for 6 times [do not use pow() fn]

1. take ‘i’ as loop variable with start value 1, here ‘i’ works as loop variable as well as to print output values
2. print ‘i’ value as output
3. multiply 10 to ‘i’, to generate next output value
4. repeat step2 and step3 as far as i<=100000
5. stop.

```
void main()
{       int i=1;
        while(i<=1000000)
        {       printf(“%d   “, i );
                i=i*10;
        }
}
```

Printing 1000000, 100000, 1000, 100, 10, 1 for 6 times

1. take ‘i’ as loop variable with starting output value 1000000
2. print ‘i’ value as output
3. divide ‘i’ by 10, to generate next output value
4. repeat step2 and step3 until i>0
5. stop.

```
void main()
{       int i=1000000;
        while( i>0 )
        {       printf(“%d   “, i );
                i=i/10;
        }
}
```

Printing 1, 2, 4, 8, 16, 32 ------ for 10 times [do not use pow() fn]

1. take ‘i’ to repeat loop for 10 times
2. take ‘p’ to generate values 1, 2, 4, 8, 16, … etc [2^0 , 2^1 , 2^2 , 2^3 , 2^4 , 2^5 ...]
3. set i=1, p=1 // starting values of i & p
4. print(p)
5. multiply 2 with ‘p’, to get next output of 2^i value
6. increment loop variable ‘i’ by 1
7. repeat step4 to step6 until i<=10
8. stop.

```
void main()
{       int i=1, p=1;
        while( i<=10 )
        {       printf("%d   ", p );
                p=p*2;
                i++;
        }
}
```

Printing N, N/2, N/4, N/8, …1.

```
ip: 100
op: 100   50  25   12   6   3    1
```

1. read N value from keyboard
2. print N value
3. cutdown N to N/2, to get next value of output
4. repeat step2, step3 until N>0 and stop

```
void main()
{     int N;
      printf("enter N value :");
      scanf("%d", &N);
      while( N>0 )
      {   printf("%d ", N);
          N=N/2;
      }
}
```

Printing odd numbers for first N terms

```
 ip:  10
op:  1   3   5  7   9   11   13    15   17   19
```

```
void main()
{       int i=1, N;
        printf("enter number of terms to print :");
        scanf("%d", &N)           //  scanning N as no.of terms to print
        while( i < 2*N)           //  Nth term of odd series ends with 2*N
        {       printf(" %d   ", i );    //  printing odds one by one
                i=i+2;            //  incrementing i by 2 to get next odd value
        }
}
```

Printing odd numbers from N to 1 in reverse order

Note: The input value **N** may be odd or even entered by the user, for example

```
   ip:  15                    ip: 24
  op:  15   13    11 ….1      op: 23   21    19 …..1
void main()
{     int N;
      printf("enter n value :");
      scanf("%d", &N);
      if(N%2==0)      // if input N is even then change to odd, because we have to start with odd value.
           N=N-1;     //  if input N is even like 24, then changing to next odd 23
      while( N>0 )              // now printing using loop from N to 1
      {   printf("%d    ", N);  // printing odds one by one
          N=N-2;                // decrement N to N-2 to get next odd number of next cycle
      }
  }
```

The above code can be written in other way as

[Here each number from N to 1 checked in favor of oddness and printing it on the screen]

```
  while(N>0)                  // loop to print odd numbers up to 1.
  {   if( N%2 == 1 )          // if N is odd then print it
          printf("%d  ",  N ); // remember braces not given
       N--;                   // decrement N towards to 1.
   }
input: 25
Sample run 25%2==1  →  1==1  →   true → prints  25 as output
           24%2==1  →  0==1  →   false→  skips
           23%2==1  →  1==1  →   true → prints  23 as output
           22%2==1  →  0==1  →   false → skips
           21%2==1  →  1==1  →   true  → prints  21 as output
           20%2==1  →  0==1  →   false → skips
           19%2==1  →  1==1  →   true  → prints  19 as output
           ----
```

Printing 10 numbers before and after of a given N

```
   ip:  36
  op:  26  27 28….34  35   36  37  38…   45   46
  void main()
  {       int i , n;
          printf("\n enter n value :");
          scanf("%d", &n);          //  scanning 'n' from keyboard
          i=n-10;                   //  start from n-10
          while(i<=n+10)            //  repeat loop up to n+10
          {   printf("%d   ", i);
              i++;
          }
   }
```

Generate and print list of numbers from N to 1

Here N is input from keyboard and print the list of numbers as long as the value of N >1.

if N is even then next number of N is → N/2 (half),

if N is odd then next number of N is → 3N + 1

if input N is 13, then we have to print like: 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1

```
void main()
{     int N;
      printf(“enter N value :”);
      scanf(“%d”, &N);
      while( N>1 )
      {   printf(“%d   “, N);
          if( N%2==0)            // if N is even then reduce  N=N/2
              N=N/2;
          else
              N=3*N+1;           // if N is odd then raise N=3*N+1
      }
}
```

Printing values of 1, -2, 3, -4, 5, -6, 7, …N times

1. take a loop variable ‘i’ for 1 to N do.
2. take a variable ‘s’ with 1, and change its value alternatively to +1, –1, +1, –1,… in the loop.
   for this multiply ‘s’ with –1 in the loop, so that the sign changes alternatively.
3. print s*i value in the loop, then s*i values are: 1, -2, 3, -4, 5, …

In the loop, the values of ‘i’ and ‘s’ are

```
    i → 1,  2,   3,   4,  5,    6,  7,  …
    s → 1 , -1,  1,  -1,  1,  ,-1,  1, …
  s*i → 1,  -2,  3,  -4,  5,
```

```
void main()
{   int i, s, n;
    printf(“enter n value:”);
    scanf(“%d”, &n);
    i=s=1;
    while( i<=n)
    {   printf(“%d  “,  s*i );
        s = -1*s;
        i++;
    }
}
```

Printing 1 to 10 numbers by skipping 5

op: 1 2 3 **4** **6** 7 8 9 10

```
void main()
{       int i=1;
        while(i<=10)
        {   if( i!=5)                  // if ‘i’ value is not 5 then print(i)
                printf(“%d   “, i );
            i++;
        }
}
```

Printing multiplication table up to 10 terms for a given input N

```
     ip:  9
    op:  9*1=9
         9*2=18
         9*3=27
         ....
         9*10=90
```

Logic: print(N*i) for 10 times where i=1 to 10

```
void main()
{   int n , i;
    printf("enter table number :");
    scanf("%d", &n);
    i=1;
    while(i < 11)             // loop to print 10 terms
    {   printf("\n %d * %d = %d", n, i, n*i );
        i++;
    }
}
```

Printing factors of a given number (N)

```
   ip:  18
  op:  1   2   3   6   9   18.
```

Logic: Here the N value can be anything, to know its factors, divide N with all possibility numbers from 1,2,3,…N, which ever divides the N then it will be a factor, so print it on the screen. This is as shown below

Let **N** is : **15**

check as : **15%1==0 → true→ print(1) as a factor**

15%2==0 → false→ skips

15%3==0 → true→ print(3) as a factor

15%4==0 → true→ skips

15%5==0 → true→ print(5) ……….

```
void main()
{   int  i , N;
    printf("\n enter n value :");
    scanf("%d", &N);
    i=1;                            // checking for factors starting from 1.
    while(i<=N)                     // loop to check  N with all possibilities 1,2,3,4…for finding factors
    {   if(N%i==0)                  // checking N, whether it is divisible or not
                printf("%d\t ", i); // if N is divisible then print 'i' as factor
        i++;
    }
}
```

Finding small factor of N (excluding '1' as factor)

ip: 15	ip: 18	ip: 35	ip: 17
op: 3	op: 2	op: 5	op: 17

logic: for small factor, divide N with 2,3,4,5,… upto N, that is check with all possibilities from 2 to N, but whichever divides first then it will be small factor and stop the loop.

```
void main()
{       int i, N;
        scanf("%d", &N);
        i=2;                    // checking for factors starting from 2
        while(i<=N)             // we have to check up to N.
        {   if(N%i==0)          // verifying N with each number in 'i'.
                break;          // break stops the loop when divides.
            i++;
        }
        printf(" small factor is %d ", i);
}
```

Here N verified repeatedly for small factor from 2 to N, meanwhile, if it is divided with any number then loop stops by break and prints it as small factor. The **break** statement stops the loop and moves the control out of loop-body, which is as above shown arrow. This program can be simplified as

```
i=2;
while(N%i!=0)  i++;             // the loop stops when 'i' divides the N
printf("small factor is %d", i);
```

Finding biggest factor of N (excluding N as a factor)

ip: 15	ip: 18
op: 5	op: 9

Hint: Generally, for any number, the possible factors lies in range 1,2,3,…N/2 (exclude N). That is, there should not be factors found beyond N/2. For example, if we take 100 then possible factors are 1,2,3, …50. The value 100 never divides with 51, 52, 53,…,98, 99, so it is useless to check beyond N/2.

In this program, as we want big factor, it is wise to check from N/2 to 1 (in reverse order)

```
void main()
{       int i,n;
        scanf("%d", &n);
        i=n/2;                  // checking for factors starting from n/2
        while(i>0)              // check with all possibilities up to 1.
        {   if(n%i==0)          // verifying whether 'n' divides or not ?
                break;          // break stops the loop, when divided.
            i--;
        }
        printf(" big factor is %d  " , i );
}
```

Above program can be written in simple form as

```
i=n/2;
while( n%i!=0 )  i--;           // the loop stops when 'i' divides the N
printf("big factor is %d", i);
```

printing common factors of two numbers

```
  ip:  12   18                          ip: 10   30
 op:  1, 2, 3, 6                       op: 1, 2, 5, 10
   step1: take two input values into x , y
   step2: find smallest of  x , y              // because common factors exist from 1 to smallest of x,y
              if(x<y) small=x
              else small=y;
   step3: repeat the loop from 1 to 'small'  (i = 1 to small)
             if(x%i==0  && y%i==0)
                   print( 'i' as common factor )
```

note: code the program yourself

printing biggest common factor (GCD) of two numbers

```
  ip:  12   18                          ip: 10   30
 op:  6                                op: 10
   step1: take two values into x,y
   step2: find  smallest of x,y                // because biggest common factor exist from small to 1
   step3: repeat loop from 'small' to 1   (i=small to 1)
           if(x%i==0  && y%i==0)
                 print( 'i' as common factor )
                 break;          // whichever divides for first time then it will be GCD, so stop the loop
    step4: stop
```

Checking given number is perfect or not?

perfect: if sum of all factors is equal to given N then it is said to be perfect. (don't take N itself as a actor)

logic: Check for factors from 1 to N/2 and add all divisible to variable 'sum'.

For example: 6 (1+2+3→6),   28(1+2+4+7+14→28)

```
   ip: enter N value: 6        ip: enter N value: 7              ip: enter N value: 28
  op: yes                     op: no                           op: yes

   step1: take variables N, sum and i;
   step2:  scan N                          // because biggest common factor exist from small to 1
   step3: sum=0 , i=1,
   step3: repeat loop 'i' from 1 to N/2
              if( N%i==0)
                   sum=sum+i;         // adding all factors
   step4: if sum==N then print(" N is perfect number");
           else  print(" N is not perfect number");
   step5: stop.
```

**Printing given number is prime or not?

Write a program to find the given number N is prime or not?

```
     ip: N = 17                          ip: N = 18
    op: yes, it is prime                op: no, it is not prime
```

Prime number does not divide with any number except 1 & N itself. So, to know primness of N, check the N by dividing from 2,3,4,….N/2. If not divided with any of these numbers then it is said to be prime. Based on this concept, a beginner may write prime logic **as bad as**

```
   i=2;
   while( i <= N/2 )
   {       if(N%i==0)
                   printf("\n not prime");
           else
                   printf("\n prime");
           i++;
    }
  ip:  N=10                  (many outputs)
 op: not prime               // when 10%2==0  is true
       prime                 // when 10%3==0  is false
       prime                 // when 10%4==0  is false
       not prime             // when 10%5==0  is true
```

Here we get many outputs and it is wrong, because prime-ness can be declared only after checking with all numbers from 2 to N/2. Here we have to take output decision only after completion of loop(not inside loop), So, to solve this problem, better to use Boolean logic, there several other logics to find prime-ness, but this Boolean logic is standard and best, the code is as given below

```
void main()
{     int n, i, bool ;
      printf("enter N value :");
      scanf("%d", &N);
      i=2, bool=1;                 // assume 'N' is prime, so take bool as 1 (true).
      while(i<=N/2)
      {       if(N%i==0)
              {    bool=0;         // here N divided, therefore N is not prime, so set bool to 0 (false) and stop
                   break;
              }
              i++;                 // if N is not divided then check with next 'i' value until  i<=n/2
      }
      if( bool==1)   printf("prime");       // printing output, after completion of loop
      else   printf("not prime");
}
```

Logic2: Count all factors of N, that is, divide N with all numbers from 1 to N and count them, if factors count==2 then say it is "prime" or else "not prime". This is simple logic but execution takes much time than above bool logic, because it is unnecessary checking with all numbers if once N is divided. (here we are not stopping by break)

```
count=0; i=1;
while(i<=N)              // loop to count all factors between 1 to N
{     if(N%i==0)
           count++;      // increments 'count' when N is divided with 'i'
      i++;
}
if( count==2)  printf("not prime");
else  printf("prime");
```

Summing 2+2+2+2+ …. N times (do not use multiplication operator)

Logic: take a variable ‘sum’ with zero, and repeatedly add 2 to ‘sum’ for N times.

step1: take variable ‘sum’ to store sum of 2+2+2+ … for N times

step2: set sum=0 to clean the garbage

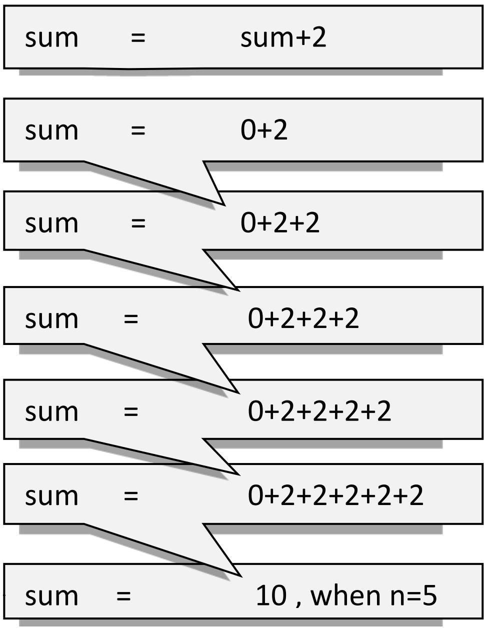
step3: add ‘2’ to sum, this is as: sum=sum+2

step4: repeat step3 for N times

step5: print (sum)

Look at the picture, how 2 is added to ‘sum’ in cycles

```
void main()
{  int sum , i , N ;
   scanf(“%d”, &N);
   i=1, sum=0;
   while(i<=N)
   {  sum=sum+2;
      i++;
   }
   printf(“sum is %d”, sum);
}
```



sum = sum+2

sum = 0+2

sum = 0+2+2

sum = 0+2+2+2

sum = 0+2+2+2+2

sum = 0+2+2+2+2+2

sum = 10 , when n=5

Note: if sum is not initialized to 0 in the beginning, then it contains garbage value, so the instruction sum=sum+2 executes for first time as “sum=GarbageValue+2”. So, set sum=0 at beginning of loop.

Finding sum of series of $1^2+2^2+3^2+4^2+5^2$ …..$+10^2$

Prove that sum of $1^2+2^2+3^2+4^2+5^2$ …..$+10^2$ is equal to 385 or not?

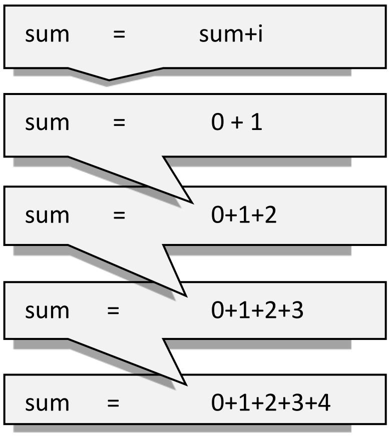
output: yes/no

Logic: take ‘sum’ with zero, repeatedly add i^2 value to sum for 10 times, where i=1 to 10 after finding sum, compare with 385 for equality.

```
void main()
{    int  i, sum;
     sum=0; i=1;
     while(i<=10)                // loop to add up to 10 terms
     {  sum=sum + i*i;
        i++;
     }
     if(sum==385)  printf(“program is correct”);
     else  printf(“program has some logical mistakes”);
}
```

Finding sum & product of 1 to n. (1+2+3+4…+n; 1*2*3*4*…*n)

```
void main()
{     int  i, n, sum, product;
      printf(“enter n value :”);
      scanf(“%d”, &n);
      sum=0,  product=1, i=1;
      while(i<=n)
      {     sum=sum+i;
            product=product*i;
            i++;
      }
      printf(“\n sum=%d \n product=%d”, sum, product);
 }
```



sum = sum+i

sum = 0 + 1

sum = 0+1+2

sum = 0+1+2+3

sum = 0+1+2+3+4

Finding sum of given N values

This program accepts numbers one by one from keyboard until last input value is 0, when zero is entered then stops scanning and prints sum of all values.

```
    ip: 11↲     3↲     44↲     30↲     23↲   0↲
   op:  sum = 111
   void main()
   {   int  n, sum=0  ;
       while(1)
       {       printf("enter a value [ zero to end ] :");
               scanf("%d", &n);       // scanf() must be written inside loop, to scan value by value until n==0
               if(n==0) break;
               sum=sum+n;             // summing value by value
       }
       printf("\n sum=%d ", sum );
  }
```

Finding sum of 2^0+ 2^1+2^2+……+2^n without using pow() function

```
   ip:  n=5
  op:  sum=31
 void main()
 {      int  n, sum, p, i;
        printf("enter n value :");
        scanf("%d", &n);
        sum=0; p=1; i=1;       //  here 'i' values are 1,2,3,4, …. N,   the 'p' values are 1,2,4,8,16,32, ….
        while(i<=n)            //   loop to add 'n' terms
        {   sum=sum + p;       //   adding 2^i values to sum
            p=p*2;             //    generating next 2^i value
            i++;
        }
        printf("\n sum=%d", sum);
 }
```

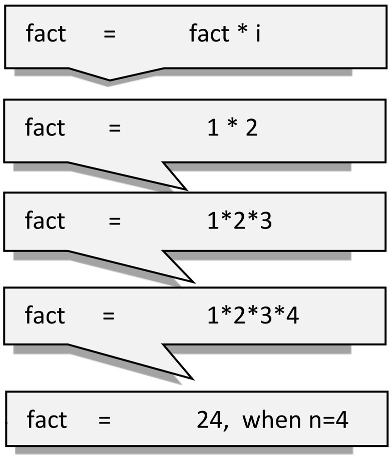
Finding sum of series of x^1+x^2+x^3+x^4+ ….x^n (without using power function)

```
    ip:   3     5   (x=3, n=5)
   op:   3+9+27+81+243 → 363
   void main()
   {       int  i=1, n, sum=0, x, p;
           printf("enter x , n value :");
           scanf("%d%d", &x, &n);
           p=x;                           // first value of p is x^1
           while(i<=n)                    // loop to add up to 'n' terms
           {   sum=sum + p;               // adding x^1, x^2, x^3 …  to sum
               p=p*x;                    // to generate next x^i value
               i++;
           }
           printf("\n sum=%d", sum);
   }
```

Finding factorial of N. (1*2*3*4*…*N)

Here the formula N(N+1)/2 is not used to get the sum of all terms

```
   ip:  N=4
  op:  product=24      (1*2*3*4)
 void main()
 {      int  i, N, fact;
        printf("enter N value :");
        scanf("%d", &N);
        fact=1; i=1;
        while(i<N)
        {   i++;
            fact=fact * i ;
        }
        printf("\n factorial = %d", fact);
  }
```



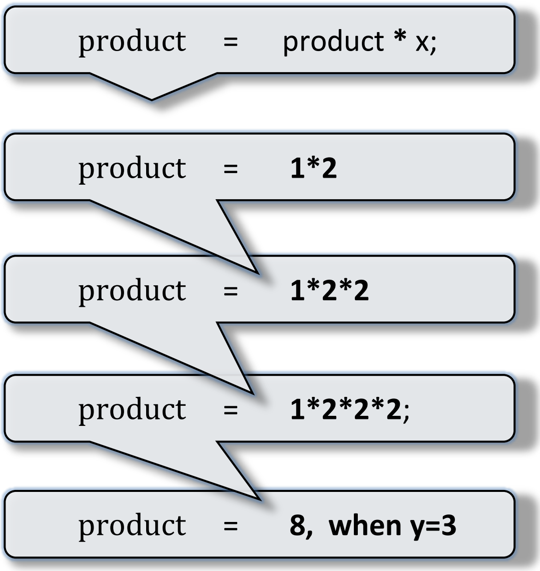
Finding x^y value based on given input values (x,y)

ip: 2 3

op: 8   (2*2*2)

1. scan input values to x, y
2. take variable **'p'** to store value of x*x*x … for y-times
3. initialize **'p'** with 1 to clean the garbage
4. now repeatedly multiply **'x'** to **'p'** for y-times to get x^y value
5. print(p)
6. stop

```
    void main()
    {      int  i, x, y, p=1;
           printf("enter x , y values :");
           scanf("%d%d", &x, &y);
           while(i<=y)      // loop to multiply  y-times
           {    p=p*x;
                i++;
           }
           printf("\n x^y =%d", p);
    }
```



$X^1/1!$ + $X^2/2!$ + $X^3/3!$..... N times [do not use pow() fn]

ip: x=3, N=5

op: sum= 17.4 [3.0 + 4.5 + 4.5 + 3.37 + 2.02 → 17.4]

1. Take 'i' for the looping 1 to n times.

2. Take a variable 'p' and generate its values to x^1, x^2, x^3, x^4, x^5, …
for this multiply 'x' to the 'p' to get next value.

3. Take a variable 'fact' and generate its values to 1!, 2!, 3!, 4!, …
for this multiply 'i' to the 'fact' to get next factorial value.

4. Take sum and add all these 'p/fact' values, this is like sum=sum + p/fact;

```
 void main()
 {      int i, fact, x, p, n;
        float sum;
        printf("enter x  n value :");
        scnaf("%d%d", &x, &n);
        i=1, fact=1, sum=0, p=x;
```

```
     while( i<=n)
     {   sum = sum + (float) p/fact;
         p=p*x;
         i++;
         fact=fact*i;
     }
     printf("sum = %f", sum);
  }
```

Finding sine series value: sin(x)→ $x^1/1! - x^3/3! + x^5/5! – x^7/7!$ … up to 10 terms

```
    ip: x=3, N=5
   op: sum=0.1453  [ (3.0) + (-4.5)  + (2.025) + (-0.4339) + (0.05424) ]
void main()
{     int i, fact, sign;
      float x, sum, p;
      printf("enter sign(x) value :");
      scanf("%f", &x);
      i=fact=1; p=x; sign=1; sum=0;
      while(i<20)                       // adding  10 terms ( not 20, because  'i' is incrementing by 2)
      {   sum=sum+ sign* p/fact;
          sign=-1*sign;                 // changing 'sign' for next term
          p=p*x*x;                      // getting next x^i value
          i=i+2;                        // getting next odd number
          fact=fact*(i-1)*(i);          // getting next factorial value
      }
      printf("\n sum =%f", sum);
}
```

Printing prime factors of given N (The product of factors should be equal to N)

ip: N=100

op: 2 2 5 5

step1: divide N with first possible factor 2, here the number 2 is prime.
if N is divided with 2 then print(2) as prime factor and decrement N to N/2.

step2: repeat step1 as long as '2' divides the N.

step3: Now take 3 and proceed as long as 3 divides the N, as said in step1. Of course '3' is also prime.

step4: Now take 4, we know 4 is not prime, but 4 will not be divided the N because we already did with 2 before, so there should not be 2 multiples left behind in N. [you may ask one question, why to divide with 4 when it is not prime, because it is difficult to take only primes for checking, taking only primes is another problem. So continuously/blindly divide the N with 2,3,4,5,6,7,8,9 ….

step5: repeat this process until N>1

```
      i=2;
      while(N>1)
      {     if( N%i==0)
            {     printf("%d   ", i);
                  N=N/i;
            }
            else   i++;
      }
```

Printing prime factors of given N without duplicates

This process is same as above program but it prints factors only once. (not like 2, 2, 5, 5)

ip: N=100
op: 2 5

logic: In loop cycles, previously printed factor is not equal to current factor then it prints it. Here extra variable 'prev' is taken to store previous prime factor of 'i' (for first time, take prev=1)

```
i=2;  prev=1;
while( N>1 )
{      if( n%i==0 )
       {       if( prev != i )     // if previous printed factor is not equal to current factor then print
               {   printf(" factor %d ", i );
                   prev=i;        // take this current 'i' value as 'prev' for next cycle
               }
               n=n/i ;
       }
       else  i++;
}
```

In programming, the most of the logics fall under 5 types

1. **Sum accumulation logic: used to find sum of values. for example**
   - finding total bill of all items at supermarket
   - sum of polynomial equation terms like $x^1 + x^2 + x^3$…
2. **Product accumulation or diminishing logic: used to find product of n terms**
   - finding product values like factorial
   - finding x^y like values
3. **Boolean logic: Mainly used checking or searching algorithms**
   - finding given number is a prime or not
   - finding specific record in a file
4. **Selection logic: used to select a desired value from a group of values**
   - finding big/small from array of elements
   - finding smallest path in the network

5. updating or advancing data values

finding GCD of two numbers
printing Fibonacci series

Other miscellaneous logics like counting, checking for factors, increment/decrementing …etc

Printing each digit separately in a given number (N) [printing right to left]

ip: 2345 ip: 724
op: 5,4,3,2 op: 4, 2, 7

Logic: Extract digit by digit from right-to-left in N, and print on the screen, following steps explains it.

step1: divide N with 10 and collect the remainder(s), the remainder is always the last digit of N, when it is divided with 10 (lastDigit=N%10)

step2: now print the last digit

step3: to get next digit of N easily, remove current last digit from N, this is by doing N=N/10

step4: repeat step-1, step-2, step-3 until N>0

Let the input is 2345, following table explains how digits are extracted.

Iteration-1 Here N is 2345	10) 2345 (234 2340 ----------- 5 → printf(“5”)
Iteration-2 after N=N/10 Here N is 234	10) 234 (23 230 ----------- 4 → printf(“4”)
Iteration-3 after N=N/10 Here N is 23	10) 23 (2 20 ----------- 3 → printf(“3”)
Iteration-4 Here N is 2	10) 2 (0 0 ---------- 2 → printf(“2”)
Here N is 0 (stop)	0 (stop)

```
void main()
{
          int  N  ,  lastDigit;
          printf(“enter N value :”);
          scanf(“%d”, &N);
          while(N>0)
          {
                    lastDigit=N%10;              // gives last digit of N

                    printf(“%d,”,  lastDigit );  // printing lastDigit

                    N=N/10;                      // removes last digit of  N
          }
}
```

Finding sum of all digits in a given number (N)

ip: 2345
op: 2+3+4+5 → 14

step1: take ‘**sum’** to store sum of all digits

step2: take ‘N’ to store input value

step3: repeat following instructions until N>0

- get the last digit(d) of N by dividing 10, the remainder will be last digit [lastDigit=N%10]
- add lastDigit to **sum [** sum = sum + lastDigit]
- remove lastDigit from N, for this divide N with 10 and collect the quotient to N [N=N/10]

step4: print(sum)

step5: stop

Note: code the program yourself

Finding whether the given number (N) contains a digit 5 or not ?

```
ip:  2357                    ip: 2346
op:  yes, 5 is exist         op: no, 5 is not exist
```

without deep study on logic, one may write the program as bad as

```
void main()
{     int N, d;
      scanf("%d", &N);
      while(N>0)
      {   d=N%10;                     //  gives last digit of N
          if(d==5)    printf(" 5 exist");
          else     printf("5 not exist");
          N=N/10;                     // removing last digit of N
      }
}
```

if input is: 2357 → output is: 5 not exist, 5 exist, 5 not exist, 5 not exist (output of each digit)

the output **"5 is not exist"** can be conformed only after comparing with all digits of N, but here, in this loop, for each digit comparison the output has shown on the screen. Therefore, the result displayed 4 times.

Solution is, use Boolean logic, compare digit by digit until '5' is found or N>0.

If 5 is found in loop, then set bool=1 and stop, after looping, show the result based on bool.

step1: first of all, assume, the digit '5' is not exist in N, so initialize '**found=0'** (here 'found' is bool variable)

step2: repeat following instructions until N>0

- get the last digit in **N** [**d=N%10**]
- if d==5 then set '**found=1'** and stop the loop
- if not 5, then go back and check next digit, before going back, remove the current last digit.

step3: after loop, print the result based on the **'found'**

```
if(found==1) print(" 5 found");
else print("5 not found");
```

step4: stop.

```
void main()
{     int N, d, found;
      printf("enter N value :");
      scanf("%d", &N);
      found=0;                    // let us assume digit '5' not exist in N.
      while(N>0)
      {     d = N%10;
            if(d==5)  { found=1; break;}      //  now the digit '5' found, so set found=1 and stop the loop
            N=N/10;
      }
      if(found==1)  printf("the digit 5 found");
      else  printf("the digit 5 not found");
}
```

Finding sum of even and odd digits separately in a number (N)

```
  ip:  23456
op:  evens sum: 2+4+6 →12
        odds sum: 3+5 → 8
```

```
 void main()
{     int N, evenSum, oddSum, d;
      printf(“enter N value :”);
      scanf(“%d”, &N);
      evenSum=oddSum=0;
      while(N>0)
      {   d=N%10;              // gives last digit of N
          if( d%2==0)          // checking digit is odd or even
                 evenSum+=d;
          else  oddSum+=d;
          N=N/10;
      }
      printf(“\n even sum = %d, odd sum = %d”, evenSum, oddSum);
}
```

Counting no.of digits in a given number (N)

```
 ip: 234                 ip: 45
op: count=3              op: count=2
```

- take variable **digitCount** to count digits of N
- Remove the last digit of **N** by doing **N=N/10** and count it as one digit by incrementing **digitCount** Repeat this step until **‘N’** becomes zero
- Print output **‘digitCount’**

The variable values in every cycle as

	digitsCount++	N=N/10
Initially→	0	2345
After 1st cycle	1	234
After 2nd cycle	2	23
After 3rd cycle	3	2
After 4th cycle	4	0

```
 void main()
{     int N, digitCount=0;
      printf(“enter N value :”);
      scanf(“%d”, &N);
      while(N>0)
      {   N=N/10;              // removing last digit from N.
          digitCount++;        // counting the digits
      }
      printf(“no of digits = %d”, digitCount);
 }
```

*Finding given number (N) is Armstrong or not?

The number 153 is Armstrong, as sum of cubes of all digits is equal to N: $1^3+5^3+3^3$ → 1+125+27 → 153

```
 ip: 153                  ip: 234
op: Armstrong            op: not Armstrong
```

```
  void main()
  {     int N, sum, rem, temp;
        printf(“enter N value :”);
        scanf(“%d”, &N);
        sum=0; temp=N;
        while(N>0)
        {       rem=N%10;
                sum = sum + (rem*rem*rem);       // adding sum of cubes of digit
                N=N/10;
        }
        if(sum==temp) printf(" Armstrong");
        else  printf("not a Armstrong");
    }
```

The instruction N=N/10 makes the 'N' to zero by the time of closing loop. There by, the original value of N is lost after loop. So it saved in 'temp' before looping, later checked with '**sum'** and result printed.

*Finding reverse of a given number (N)

ip: 234 ip: 4673
op: 432 op: 3764

- let us assume the variable **'rev'** already has a value 23 at the moment (rev=23)
- now to insert '4' at the end of **'rev',** we do as "rev=rev*10+4" → 23*10+4 → 234
- in this way, the digits are extracted one by one from N, and inserted at the end of **'rev'**.

```
    void main()
    {   int N, rev=0, digit;
        printf("enter N value :");
        scanf("%d", &N);
        while(N>0)
        {   digit=N%10;
            rev=rev*10+digit;      // adding digit at end of 'rev'
            N=N/10;
        }
        printf("reverse = %d", rev);
  }
```

Let input is 2345, then the instruction **"rev=rev*10+digit"** in every iteration is as follows

Let N=2345	d=n%10	rev = rev*10+digit	N=N/10
After 1st cycle	2345%10 → 5	0*10+5 → 5	234
After 2nd cycle	234%10 → 4	5*10+4 → 54	23
After 3rd cycle	23%10 → 3	54*10+3 → 543	2
After 4th cycle	2%10 → 2	543*10+2 → 5432	0

Finding given number (N) is palindrome or not

If the number and its reverse are equal then it is said to be palindrome

```
      ip:  234                         ip:  434
     op:  not a palindrome          op:  palindrome
```

step1: find the reverse of given number(N) as said in above program

step2: after loop, compare reverse-value with N and print output

```
void main()
{       int N, rev=0, digit, temp;
        printf("enter N value :");
        scanf("%d", &N);
        temp=N;         // saving 'N' value in temp
        while(N>0)
        {       digit=N%10;
                rev=rev*10+digit;
                N=N/10;
        }
        if(temp==rev) printf(" given number is palindrome");
        else printf("given number is not palindrome");
}
```

Finding the biggest digit of a given number (N)

```
     ip:  5394                  ip:  2375
    op:  big= 9               op:  big= 7
```

Take a variable called **bigg** and compare it with all digits of N, if any digit is bigger than the **bigg** then copy it to **bigg,** finally big-value is collected to **bigg**. Remember, initially the bigg value is zero.

```
void main()
{       int N, digit, bigg=0;
        scanf("%d", &N);
        while(N>0)
        {       digit=N%10;
               if(bigg < digit )
                      bigg=digit;
              N=N/10;
        }
       printf("\n big digit = %d", bigg);
 }
```

Let us see, how digits are compared and assigns to bigg

	d=N%10	If(bigg<digit)
		bigg=digit
			N=N/10

At cycle1	2375%10 → 5	0<5 → 5	2375/10 → 237
At cycle2	237%10 → 7	5<7 → 7	237/10 → 23
At cycle3	23%10 → 3	7<3 → 7	23/10 → 3
At cycle4	2%10 → 2	7<2 → 7	2/10 → 2

Finding left most odd digit in a given number (N)

Note: If odd digit not found then show an error message called "odd digit not found"

```
      ip: 2345                     ip: 2468
     op: 3                        op: odd digit not  found
```

Logic: extract digit by digit in N, if any odd digit is found then store into a variable called 'temp' and finally print it. Initially take 'temp' with '0'.

```
void main()
{       int N, rem, temp=0;
        printf("enter N value :");
        scanf("%d", &N);
        while(N>0)
        {       rem=N%10;
                if(rem%2==1)
                     temp=rem;          // if odd found then store into temp
                N=N/10;
        }
        if(temp==0) printf("no odd digit found");
        else printf("\n the found odd digit is %d", temp);
    }
```

Printing each digit of a number(N) in English words

This program extracts all digits of **N** from left to right and prints each digit separately in English words.

```
     ip: 2345                          ip: 724
    op: two three four five           op: seven two four
```

It is hard to get digits from left-to-right of N; because, the input N <u>may have any number of digits</u> entered by the user (not exactly 4-digits as above shown in first input)

For example to get the first digit '3' from 3456, it has to be divided by 1000, similarly to get 7 from 724, it has to be divided by 100. According to number of digits in N, it has to be divided with 1000/100/10/1. To generate this type of 10^x value based on digit of N, the code is

```
    p=1;
    while(N/p>9)         //  loop to generate P value to 10^(x-1)
    {   p=p*10;  }
```

The program code is

```
void main()
{        int N, q, P=1;
         printf("enter N value:");
         scanf("%d", &N);
         while(N/P>9)       // generating 'p' to 10^(x-1)
         {    P=P*10;
         }
         while(P>0)      // printing each digit in English words
         {    q=N/P;      // gives first digit of N as quotient
              switch(q)
              {  case 0:  printf("zero"); break;
                 case 1:  printf("one");  break;
                 case 2:  printf("two");  break;
                   ...
                 case 9:  printf("nine");  break;
              }
              N=N%P;     // removing first digit from N
              P=P/10;
         }
}
```

N=2345, P=1000	p) N (q
Iteration1 N→2345 P→1000	1000)2345(2 ⟶ printf("two") 2000 ----------- 345 ↓
Iteration2 N→345 P→100	100) 345 (3 ⟶ printf("three") 300 ----------- 45 ↓
Iteration3 N→45 P→10	10) 45 (4 ⟶ printf("four") 40 ----------- 5 ↓
Iteration4 N→5 P→-1	1) 5 (5 ⟶ printf("five") 5 ---------- 0
N→0	↘ Stop

Finding decimal value from a given binary input

ip: 1101 ip: 1111
op: 13 op: 15

1. multiply all digits of N with 2^0 2^1 2^2 2^3… from right-to-left
2. The sum of such all products forms a decimal number.

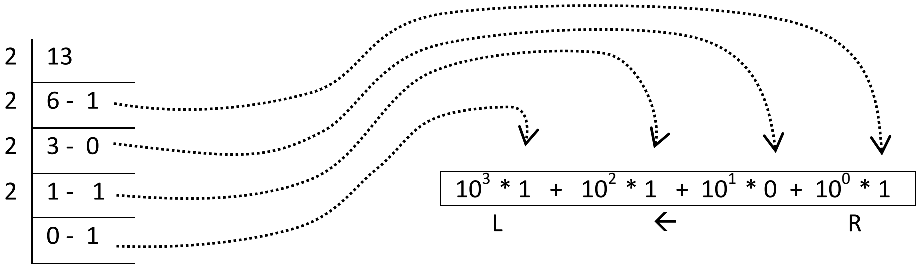
$$\underset{2^3\quad 2^2\quad 2^1\quad 2^0}{\boxed{1\quad 1\quad 0\quad 1}} \rightarrow \underset{8\ +\ 4\ +\ 0\ +\ 1}{\boxed{1*2^3 + 1*2^2 + 0*2^1 + 1*2^0}} \rightarrow 13$$

```
void main()
{       long  int rem, N, sum, p;
        printf("enter any binary number :");
        scanf("%ld", &N);
        sum=0 , p=1;
        while(N>0)
        {       rem=N%10;
                sum=sum + rem * p;
                N=N/10;  p=p*2;
        }
        printf("\n decimal number = %ld", sum);
}
```

Finding binary of a given decimal value

ip : 13 ip: 10
op : 1101 op: 1010

1. let us suppose the given number is 13 (N=13)
2. to get binary equivalent of 13, divide N repeatedly by 2 until N>0.
3. collect the remainders obtained in the division
4. this collection of remainders from bottom to top forms a binary as: **1101**



step1: multiply each reminder with 10^i and sum up
step2: the binary number formed as **1000*1+100*1+10*0+1*1 → 1101**

```
void main()
{       int N, i, rem , p;
        long int sum;
        printf("enter any decimal number :");
        scanf("%d", &N);
        sum=0; i=0; p=1
        while( N>0 )
        {   rem=N%2;
```

```
            sum=sum + rem*p;
            N=N/2;   p=p*10;
        }
        printf("\n binary number = %d", sum);
    }
```

Printing hexadecimal value from a given number

ip: 370359 ip: 159

op : 5A6B7 op: 9F

process: repeatedly divide the N with 16, and collect(insert) the remainders into variable **'sum'**

step1: R=N%16

step2: insert this R into sum, this is like "sum=sum*100+R" //here R can be 2-digit number, so multiply with 100

step3: cut down N to N/16

repeat above 3 steps until N>0

Let N=370359

16	370359	
16	23147 → 7	0*100+7 → 7
16	1446 → 11 (B)	7*100+11 → 711
16	90 → 6	711*100+6 → 71106
16	5 → 10(A)	71106*100+10 → 7110610
	→ 5	7110610*100+5 → 07 11 06 10 05

The hexadecimal value collected in '**sum'** as → 07 11 06 10 05 (7B6A5)

but output should be displayed in reverse form as → 5A6B7

to print this number in hexadecimal form, extract 2-digits at a time right-to-left from 'sum' and print

Use two loops, one is to generate 'sum' and second one is to print in hexadecimal form.

```
    void main()
    {       long int N, sum=0, rem;
            printf("enter N value:");
            scanf("%ld", &N);
            while(N>0)
            {   rem=N%16;
                sum=sum*100+rem;
                N=N/16;
            }
         // after above loop, the hexadecimal collected in 'sum' , now printing on the screen
            while(sum>0)
            {   rem=sum%100;
                if(rem<10)  printf("%d", rem);
                else   printf("%c", 'A'+ rem%10 );          //  'A'+1 → 'B' ,  'A'+2 → 'C'
                sum=sum/100;
            }
    }
```

Printing first 10 terms of Fibonacci series

The first two terms in this series are 0,1 and remaining terms are generated by adding previous two terms. This is endless series, but here we are printing first 10 terms

op: 0 1 1 2 3 5 8 13 21 34 55….

step1: Let us take first two terms are X=0, Y=1;

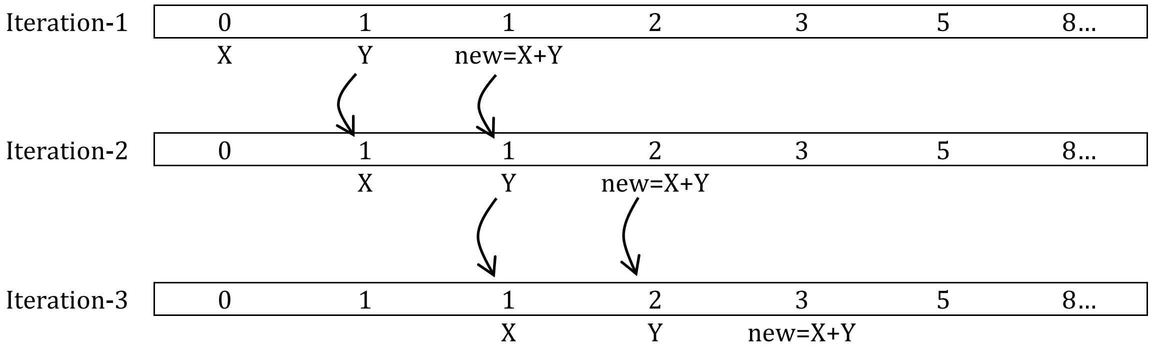
step2: Print the term X.

step3: Generate next term by adding X+Y to 'new**'**

step4: Now advance X to Y and Y to **'new'** for next cycle

step5: Repeat step2 to step4 for 10 times

Let us see how the X,Y are advancing in every iteration of loop



```
void main()
{     int  X, Y, new, i;
      X=0; Y=1; i=0;
      while(i++ < 10)
      {   printf("%d\t", X);
          new=X+Y;                // generating next  fibo number
          X=Y;                    // advancing X to Y and Y to new
          Y=new;
      }
}
```

Finding GCD of two numbers

ip: 12 18

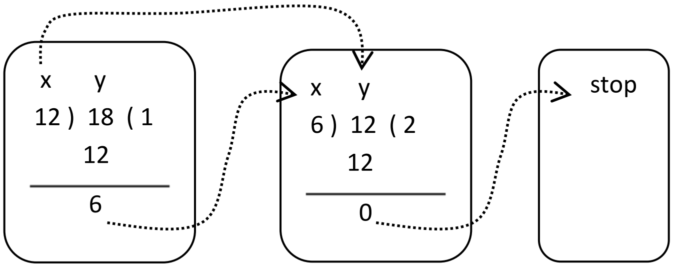
op: gcd = 6

step1: Let X , Y are input values

step2: Divide Y with X and collect the remainder to R (don't care if X>Y)

step3: If remainder(R) is zero then stop and print X as GCD

step4: If R is not zero then take X as Y and R as X for next cycle and continue this process until R is zero.



```
int  x  ,  y , rem;
scanf("%d%d", &x, &y);
while(1)
{      rem=y%x;
       if(rem==0)  break;
        y=x;                 // take x as y
        x=rem;               // take rem as x
 }
printf("\n GCD = %d", x);
```

Finding LCM of three numbers

ip: 12 18 45

op: 180

1. Let X, Y, Z are three input numbers
2. Start finding LCM of three by dividing with first factor 2
3. Now divide each X, Y, Z with 2
4. If any X, Y, Z divided with 2 then take ‘2’ as factor in LCM. (LCM=LCM*2)
   and cut down all divided numbers to its quotient obtained in the division.
   For example, if X is divided with 2 then cut down X=X/2
5. Now repeat step-4 as long as ‘2’ divide the any of X, Y, Z
6. If 2 is no longer divide the X,Y,Z, then next try with next factors 3, 4, 5…etc,
   Repeat this process until any of these (X, Y, Z)>1. For example, while(X>1 || Y>1 || Z>1) {….}

Let the numbers are 20, 15, 35 and following table shows how …

2	20	15	35
2	10	15	35
~~2,~~ 3	5	15	35
~~3,4,~~5	5	5	35
~~5,6~~,7	1	1	7
	1	1	1

```
void main()
{       int x,y,z, lcm=1, i=2, bool;
        printf("enter 3 numbers :");
        scanf("%d%d%d", &x, &y, &z);
        while( x>1 || y>1 || z>1)          // repeat until  all becomes 1.
        {       bool=0;
                if( x%i==0 )
                {   bool=1;  x=x/i;
                }
                if( y%i==0 )
                {   bool=1;  y=y/i;
                }
                if( z%i==0 )
                {   bool=1;  z=z/i;
                }
                if(bool==1) lcm=lcm*i;    // then take 'i'  as factor
                else  i++;                // if no number divisible then try with next number(s)
            }
            printf("\n lcm = %d", lcm);
    }
```

Incrementing given date by N days

Note: here Date is incrementing by 1 in each cycle for n times

1. Take three variables day, month, year to hold the given Date
2. Scan the Date
3. Repeatedly increment day in a Date
4. If ‘day’ is reached to end of month then update ‘day=1’ and ‘month++’
5. If ‘month’ is reached to end of year then update ‘month=1’ and ‘year++’
6. Continue this process for N times

```
void main()
{       int d, m, y, i=0, N;
        printf(“ente date :“);
        scanf(“%d%d%d”, &d, &m, &y);
        printf(“enter how many days to increment:”);
        scanf(“%d”, &N);
        while( i++ < N )
        {       d++;                    // incrementing day by 1-day
                if(m==2)
                {       if( y%4!=0 && d>28  || d>29 )    // if day is reached to end of February
                        {       d=1;   m++;
                        }
                }
                else if((m==4 ||m==6 || m=9 || m==11) && d>30)
                {       d=1;   m++;         // if day is reached to 30-days month ending like april, june,sept…
                }
                else if( d>31 )              // if day is reached to 31-days month’s ending
                {        d=1;   m++;
                        if(m==13)            // if date is reached to end of year
                        {       m=1;  y++;
                        }
                }
        }
        printf(“\n date after incrementing = %d/%d/%d”, d, m, y);
}
```

Finding difference of two dates in days

1. scan two dates date1(d1,m1,y1), date2(d2,m2,y2) from keyboard
2. if(date1>date2) then swap them
3. here every time in the loop, the date1 is incrementing until it reaches date2
4. the 'count' increments by 1 in each cycle to count the no.of days between two dates.

```
    void main()
    {   int d1, m1, y1, d2, m2, y2, count=0;
        scan (d1,m1,y1)
        scan (d2,m2,y2)
        while(d1<d2 || m1<m2 || y1<y2)  // loop to repeat until  date1<date2
        {       // incement  date1 by 1-day, for this write above program logic here
                count++;      // counting no.of days between dates
        }
        printf("\n no.of days between two dates  = %d", count);
  }
```

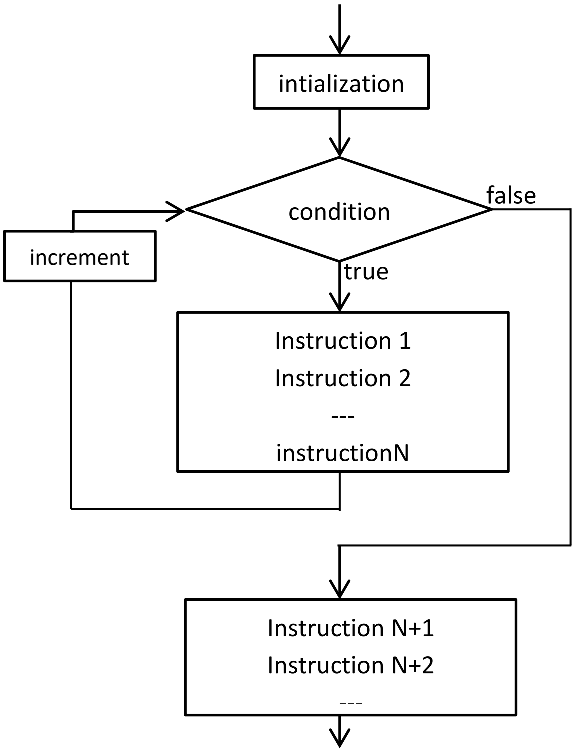
The for loop

The structure of **while-loop** is simple and used when loop construct contains only condition part, however, most of the times, loop constructs involve initialization, incrementing or decrementing. In such case, **for-loop** is more suitable than any other loop. **For-loop** consists of three expressions in its header: initialization, test condition, and incrementing of loop variable.
When your loop to be repeated 10 or N times then for-loop is the choice otherwise while-loop is choice. Remember, when your loop has parameters like i=1 and i++ then for-loop is the choice.

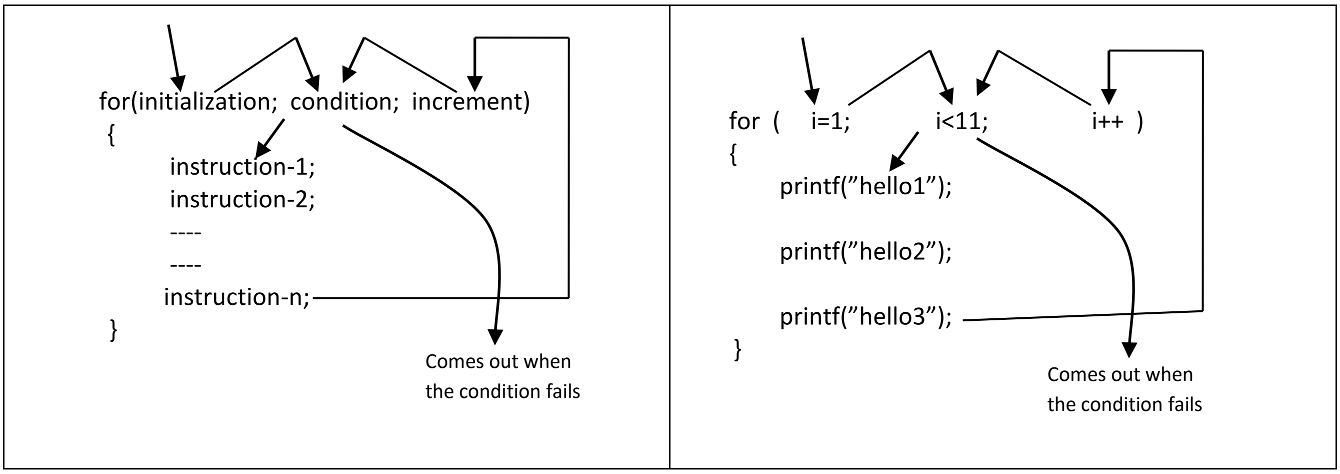
The **For-loop** is more flexible as it gives complete idea about the loop such as initial value, condition and increment. It is the programmer's choice, which loop to use when and where and it is fairly depends on his convenience. Some people say for-loop is faster than while-loop, actually same machine code is generated for both loops

Syntax for for-loop is

```
for(initialization;  test-condition;  increment)
{   instruction 1;
    instruction 2;
    ----
    ----
     instruction N;
}
instruction N+1;
instruction N+2;
----
----
```



This loop repeats as long as the condition is true, first it enters into loop-body through initialization-part. From 2nd cycle, the control enters through increment/decrement part. This is as given below



```
for(initialization;  condition;  increment)
{
        instruction-1;
        instruction-2;
        ----
        ----
       instruction-n;
}
```

```
for (    i=1;        i<11;            i++  )
{
      printf("hello1");

      printf("hello2");

      printf("hello3");
}
```

Note: for-loop header must have only two semicolons, if not, it is an error.

Let us have some examples,

Loop to print 1 to 10 numbers
for(i=1; i<11; i++)
{
printf("%d ", i);
}
	Same code in while loop
	i=1;
	while(i<11)
	{ printf("%d ", i);
	i++;
	}

Loop to print 5 multiples for 10 times
for(i=5; i<=50; i=i+5)
{
printf("%d ", i);
}
	Loop to print 10 to 1 numbers
	for(i=10; i>0; i--)
	{
	printf("%d ", i);
	}

loop to print odd numbers from 1 to 100
for(i=1; i<100; i=i+2)
{
printf("%d ", i);
}
	loop to print odds from n to 1.
	for(i=n; i>0; i--)
	{ if(i%2==0)
	printf("%d ", i);
	}

eg) loop with more initializations

we can initialize more than one variable in the loop header, in that case use comma (,) operator to separate them. (do not use semi-colon)

```
for( i=1 ,   j=2,  k=3;    ----- ;      ----- )
{
                                   separate by coma (not semicolon)
}
```

eg) loop with more increments or decrements

we can increment/decrement more than one variable in header, but use comma operator to separate them

```
for(---------- ;      ---------;     i++ ,   j=j+2 ,  k-- )
{
                                                    separate by coma (not semicolon)
}
```

eg) loop with more relational conditions in its part

logical operators are used to combine two or more relations

```
for(  ----;   i<10 && j<20;    ---- )
{
}                                    use logical operators to combine relations
```

eg) infinite loop

```
for(    ;    ;    )        // If three expressions are empty then compiler take it as infinite loop, like while(1) { … }
{
}
```

eg) infinite loop   // like while(1){…}, the for loop can be made as

```
for(  ;  1  ;    )
{
}
```

eg) all the three parts of loop header may or may not be presented, we can avoid any part just by leaving it blank. However, the two semicolons must be presented as they indicate separation of three parts.

void main()
{ int i=1;
for(; i<11 ; i++)
{
printf(“%d “, i);
}
}
	void main()
	{ int i;
	for(i=1; i<11;)
	{ printf(“%d “, i);
	i++;
	}
	}

i=1;
for(; i<11 ;)
{ printf(“%d “, i);
i++; →
}
this structure is same as while-loop
	i=1;
	while(i<11)
	{ printf(“%d “, i);
	i++;
	}

Displaying natural numbers between two given limits

```
 ip: 23     45
op: 23  24  25 ….45
void main()
{   int  lower ,  upper ,  i  ;
    printf(“Enter lower & upper limits :”);
    scanf(“%d%d”, &lower, &upper);
    for(i=lower;  i<=upper;  i++)
    {   printf(“%d  “, i);
    }
}
```

Displaying factors of a given number

```
 ip: 18
op: 1  2  3   6   9   18
```

Logic: User may enter any input value, to find factors of such unknown value, divide N with each number from 1 to N and print factor when it is divided.

```
void main()
{   int  N, i ;
    scanf(“%d”, &N);
    for(i=1; i<=N;  i++)
    {   if( N%i==0)             // if i divided the N then print ‘i’ as factor
            printf(“%d  “, i);
    }
}
```

Displaying small factor of a given number(N) (exclude 1 as factor)

Divide N with 2, 3,….N, the first divisible factor is the small factor then stop the loop and print it.

```
for(i=2; N%i!=0;  i++)    // loop stops when ‘i’ divides the N.
{   // empty loop
}
printf(“%d  “, i);
```

Finding given number is prime or not?

ip: 18 ip: 17

op: “prime” op: “not prime”

Logic: As we know prime numbers does not have factors except 1 and itself.

So check N by dividing from 2 to N/2; if anywhere divides then print “not prime” otherwise “prime”

```
void main()
{   int  n,i,bool;
    printf(“Enter number:”);
    scanf(“%d”, &N);
    bool=1;                 // let N is prime
    for(i=2; i<=N/2;  i++)
    {    if( N%i==0)
        {   bool=0;
            break;
        }
    }
    if(bool==1)  printf(“prime”);
    else  printf(“not prime“);
}
```

Printing uppercase alphabets and its ASCII values

op: upper case alphabets (A B C D E F G …. Z)

The ASCII values of A-Z is: 65 66 67 68 69 ….90

```
void main()
{   char ch;
    printf(“\n upper case alphabets :”);
    for (ch=’A’;  ch<=’Z’;  ch++)                or    //  for(i=65; i<=90; i++)
        printf(“%c”,ch);                                     printf(“%c “, ch );
    printf(“\n the ASCII values are:”);
    for (ch=’A’;  ch<=’Z’;   ch++)
        printf(“%d”,ch);
}
```

In the above program, the variable **‘ch’** internally (in memory) contains the **ASCII** value of characters. Therefore, we can perform arithmetic operations like incrementing and decrementing. When **‘ch++’** is done then it attains next character **ASCII** code.

Printing odd numbers form N to 1.

Scan N value, if N is even then decrement to next odd, now print(N) repeatedly until N>0

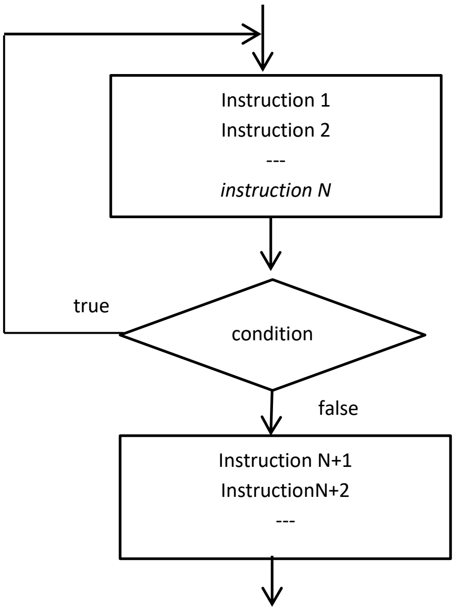
```
scanf(“%d”, &N);
if(N%2==0)
     N- -;
for( ; N>0; N=N-2 )          //  here N=N-2 is to get next odd number
{   printf(“%d “, N);
}
```

do-while loop

The do-while loop is quite opposite to while-loop, in case of while-loop, the **condition** part appears at the top of loop-body, whereas for do-while loop, the **condition** part appears at bottom of loop-body.
Thus while-loop is **top-test** construct, and do-while is **bottom-test** construct.
As a result, the do-while body executes **at least once** irrespective of condition is true/false.

The while-loop may or may not be executed; it may fail at the beginning, at least without executing once.
But **do-while** loop executes at least once irrespective of condition is true/false.
This is rarely used in the programing, who wants to execute loop body at least once.
Syntax is

```
do
{   instruction 1;
    instruction 2;
    -----
    ----
    instruction N;
} while( condition );   // bottom test-condition
instruction N+1;
instruction N+2;
```



Some programmers dislike do-while loop because of its structure, so they always avoid with the help of while-loop using break statement, which is as given below

```
while(1)
{       ----
        ----
        ----
        ----
        if(condition) break;        // bottom test-condition, now it is like do-while loop
}
```

Printing 1 to 10 numbers using do-while loop

```
void main()
{       int i=1;
        do
        {       printf(“%d “, i );
                i++;
        }  while( i <= 10 ) ;
}
```

Printing sum of ‘n’ values using do-while

This program accepts values one by one from keyboard until last value is zero, when zero is entered then stops scanning and prints sum of all values.

```
void main()
{       int n, sum=0;
        do
         {       printf(“enter values one by one :”);
                 scanf(“%d”, &n);
                 sum = sum+n;
         }  while(n!=0);              // if input value is zero then program terminates
        pirntf(“sum = %d”, sum);
}
```

Suggestion: try this program using while loop

The break and continue statements

Jumping out of loop using “break”

Loops perform a set of operations repeatedly until given condition is reached. However, in some situations, loop need to be stopped unexpectedly when required result is attained earlier.

The **break** is a condition-less control statement, used to terminate the loop unexpectedly when required result found earlier. The break throws the control out of loop-body, that is, it stops the execution of loop. Let us see syntax and example

```
    while (condition)
    {       instruction-1;
            instruction-2;
            if( condition)
                 break;
            instruction-3;
            instruction-4;
            ...
            instruction-N;
}
```

When the if-condition is true, the control move out of loop as shown like arrow, for example:

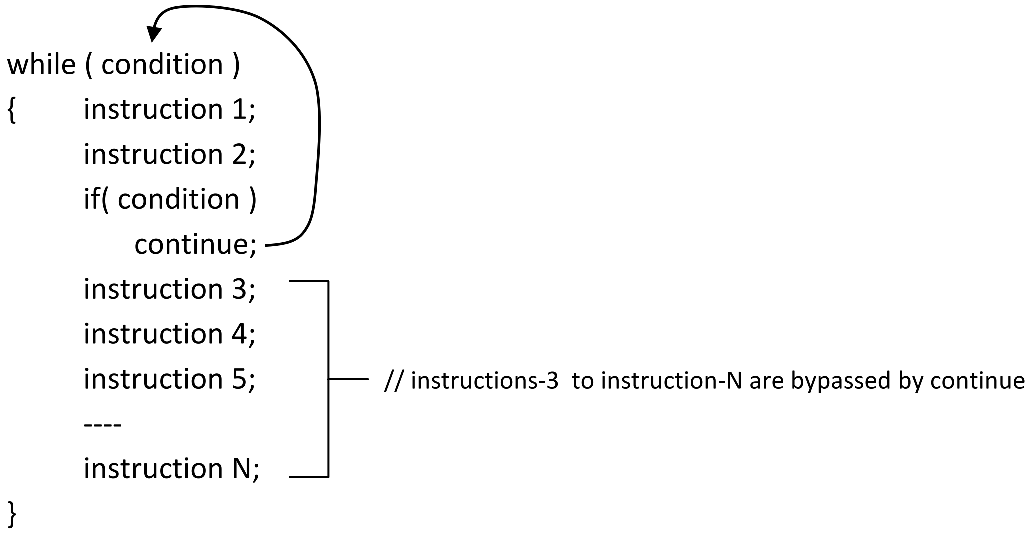
```
i=1;
while( i<10)
{       printf(“%d  “, i );
        if( i==5 ) break;
        i++;
}
output:  1 2 3 4 5
```

```
for( i=1; i<10; i++)
{    printf(“%d  “ ,  i );
     if( i==5 )  break;
}
output:  1 2 3 4 5
```

Skipping some iterations of loop by “continue”

Sometimes, some iteration of loop needs to be bypassed for some kind of data, for example, we want to calculate rank of each passed student in a class, but if student is failed then we have to skip such record.

“Continue” is also a condition less control statement, skips some instruction in the loop when it gets executed. When **continue** gets executed then the control immediately transfers back to beginning of loop and follows next cycle, so that bottom instructions are bypassed. This is as shown below



```
while ( condition )
{       instruction 1;
        instruction 2;
        if( condition )
             continue;
        instruction 3;
        instruction 4;
        instruction 5;       // instructions-3  to instruction-N are bypassed by continue
        ----
        instruction N;
}
```

Examples for “break”

```
void main()
{   int i;
    for(i=1; i<=10; i++)
    {      if(i==5)
                  break;
           printf(“%d “, i);
    }
}

output: 1 2 3 4
```

Example for “continue”

```
void main()
{   int i;
    for(i=1; i<=10; i++)
    {      if(i==5)
                  continue;
           printf(“%d “, i);
     }
}

output: 1 2 3 4 6 7 8 9 10
```

Nested Loop construct

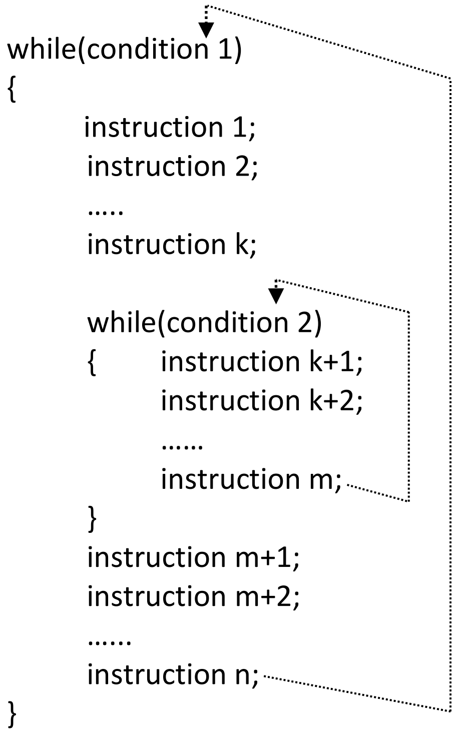
Construction of one loop with in another loop is known as nested loop, the inner and outer loop can be same or different combination. For example, while-loop can be nested by for-loop. Similarly, we can nest as many times as we want in the programming. Some examples of nested loop structures are as follows

```
while( -----)
{
      while(----)
      {
          ----
          ----
      }
}
```

```
For( -----;  -----;  -----)
{
      while(----)
      {
          ----
          ----
      }
}
```

```
for( -----;  -----;  -----)
{
       for( -----;  -----;  -----)
       {
           -----
           -----
       }
}
```

```
while( -----)
{
       for( -----;  -----;  -----)
       {
              while(----)
              {
                  ----
                  ----
              }
       }
}
```



```
while(condition 1)
{
       instruction 1;
       instruction 2;
       …..
       instruction k;

       while(condition 2)
       {      instruction k+1;
              instruction k+2;
              ……
              instruction m;
       }
       instruction m+1;
       instruction m+2;
       ……
       instruction n;
}
```

```
while(condition )
{
       instruction 1;
       instruction 2;
       …..
       instruction k;
       for( initialization;  condition;  increment )
       {
              instruction k+1;
              instruction k+2;
              ……
              instruction m;
       }
       instruction m+1;
       instruction m+2;
       ……
       instruction n;
}
```

Printing following pattern(s)

```
3 4 5 6 7 8
3 4 5 6 7 8
……
10 rows
```

Here the pattern looks like a matrix, it is like columns and rows. We know, to display numbers in a row, we need a loop. Similarly, to display all such rows in 10 times, we need an extra loop. This extra loop is an outer loop which repeats the inner loop 10 times.

```
void main()
{   int i, j;
    for( i=1; i<=10; i++)              // loop to print 10 rows
    {
          for(j=3; j<=8; j++)          // loop to print 3-8 column values of each row
              printf("%d ", j);

          printf("\n");                // inserting new line after each row
    }
}
```

Here the outer-loop is used to repeat the pattern for 10 rows and inner-loop is to print 3to8 of each row.

==

```
8  7  6  5  4  3
8  7  6  5  4  3
……..
10 rows
```

```
void main()
{   int i, j;
    for( i=1; i<=10; i++)        // loop to print 10 rows
    {     for(j=8; j>=3; j--)    // loop to print 8 to 3 column values of each row
              printf("%d ", j);
          printf("\n");          // inserting new line after each row
    }
}
```

Here outer loop is used to repeat for 10 rows, whereas the inner loop displays numbers from 8 to 3.

==

```
1
1  2
1  2  3
………..
10 rows
```

In first row, only one number is presented
In second row, two numbers are presented
In i^{th} row, 'i' numbers are presented

```
for(i=1;  i<=10;  i++)
{     for(j=1; j<=i;  j++)       // as 'i' increases, the number of columns increases
          printf("%d  ", j);
      printf( "\n");             // inserting new line after each row
}
```

Observe the inner loop ‘j’ in each & every iteration at outer loop is

```
1st iteration, when i=1 →   for(j=1; j<=1; j++)
                               printf(”%d “, j );       // 1

2nd iteration, when i=2 →    for(j=1; j<=2; j++)
                               printf(”%d “, j );     // 1  2

3rd iteration, when i=3 →    for(j=1; j<=3; j++)
                               printf(”%d “, j );     // 1  2  3

4th iteration, when i=4 →    for(j=1; j<=4; j++)
                               printf(”%d “, j );     // 1  2  3  4
```

==

123456
12345
1234
123
12
1
	for(i=6; i>=1; i--)
	{
	for(j=1; j<=i; j++)
	printf(“%d “, j);
	printf(“\n”);
	}
		The ‘j’ loop repeats as
		
		for(j=1; j<=6; j++) // 1 2 3 4 5 6
		printf(“%d “, j);
		
		for(j=1; j<=5; j++) // 1 2 3 4 5
		printf(“%d “, j);
		
		for(j=1; j<=4; j++) // 1 2 3 4
		printf(“%d “, j);
		…

654321
54321
4321
321
21
1
	for(i=6; i>=1; i--)
	{
	for(j=i; j>=1; j--)
	printf(“%d “, j);
	printf(“\n”);
	}
		The ‘j’ loop repeats as
		
		for(j=6; j>=1; j--) // 6 5 4 3 2 1
		printf(“%d “, j);
		
		for(j=5; j>=1; j--) // 5 4 3 2 1
		printf(“%d “, j);
		
		for(j=4; j>=1; j--) // 4 3 2 1
		printf(“%d “, j);
		….

123456
23456
3456
456
56
6
	for(i=1; i<7; i++)
	{
	for(j=i; j<7; j++)
	printf(“%d “, j);
	printf(“\n”);
	}
		The ‘j’ loop repeats as
		
		for(j=1; j<7; j++) // 1 2 3 4 5 6
		printf(“%d “, j);
		
		for(j=2; j<7; j++) // 2 3 4 5 6
		printf(“%d “, j);
		
		for(j=3; j<7; j++) // 3 4 5 6
		printf(“%d “, j);
		….

```
1111111
2222222
3333333
4444444
---
---
8 rows
```

```
for( i=1;  i<=8; i++)
{
        for(j=1;  j<=7;   j++ )
                printf("%d ", i );
        printf("\n");
}
```

The 'j' loop repeats as

```
 for(j=1;  j<=7;   j++)       // 1111111
    printf("%d ",  i );

for(j=1;  j<=7;   j++)        // 2222222
    printf("%d ",  i );

 for(j=1;  j<=7;   j++)       //3333333
    printf("%d ",  i );
....
```

```
1  2 3 4 5
6 7 8 9 10
11 12 13 14 15
…….
8 rows
```

```
for(x=i=1; i<=8; i++)
{       for(j=1; j<=5; j++)
        {       printf("%d ",  x );
                x++;
        }
        printf("\n");
}
```

the 'i' loop repeats 8 times to print 8 rows in the pattern
the 'j' loop repeats '5' times to print 5 columns in a row;
the 'x' value continuous increments as output is continuous numbers

```
1234554321
1234554321
1234554321
1234554321
 … 8 rows
```

```
for(i=1; i<=8; i++)
{       for(j=1; j<=5; j++)            //  to print 5 columns in first half of row
                printf("%d ", j);
        for(j=5; j>=1; j--)            //  to print 5 columns in second half of row
                printf("%d ", j);
        printf("\n");
}
```

The first 'j' loop prints the 12345, whereas second 'j' loop prints the 54321

5 blanks →						1					
4 blanks→					2		2				
3 blanks→				3		3		3			
2 blanks→			4		4		4		4		
1 blanks→		5		5		5		5		5	
0 blanks→	6		6		6		6		6		6

…6 rows

```
for(x=5,i=1; i<=6; j++)
{       for(j=1; j<=x; j++)      // loop to print blanks
                printf("⊔");     // '⊔'  this is space bar
        for(j=1; j<=i ; j++)
                printf("%d⊔" , i );
        printf("\n");
        x--;
}
```

The 1st inner j-loop prints the spaces before printing numbers
the second inner loop prints the numbers.
Try without using 'x'.

```
       11
      1221
     123321
    12344321
   1234554321
  123456654321
 12345677654321
1234567887654321
```

to get this pattern, some spaces need to be added before each row, which is below shown

7 blanks →								1	1							
6 blanks →							1	2	2	1						
5 blanks →						1	2	3	3	2	1					
4 blanks→					1	2	3	4	4	3	2	1				
3 blanks→				1	2	3	4	5	5	4	3	2	1			
2 blanks→			1	2	3	4	5	6	6	5	4	3	2	1		
1 blacks→		1	2	3	4	5	6	7	7	6	5	4	3	2	1	
0 blanks →	1	2	3	4	5	6	7	8	8	7	6	5	4	3	2	1

```
for(x=7,i=1; i<=8; j++)
{       for(j=1; j<=x;j++)      // loop to add blanks before each row
              printf(“⊔“);
         for(j=1; j<=i; j++)     // loop to print first half of values  in a row
              printf(“%d “,j);
         for(j=i; j>=1; j--)     // loop to print second half of values in a row
              printf(“%d “,j);
         printf(“\n”);
        x--;
}
```

First j-loop prints the blanks, the second & third j-loop prints the 1st & 2nd halves.

Try without using ‘x’

5 blanks →						*					
4 blanks→					*	*	*				
3 blanks→				*	*	*	*	*			
2 blanks→			*	*	*	*	*	*	*		
1 blanks→		*	*	*	*	*	*	*	*	*	
0 blanks→	*	*	*	*	*	*	*	*	*	*	*

…6 rows

```
for(x=5,i=1; i<=6; j++)
{       for(j=1; j<=x; j++)     // loop to print blanks
                printf(“⊔“);
        for(j=1; j<2*i; j++)
                printf(“*“);
        printf(“\n”);
         x--;
}
```

The 1st inner j-loop prints the spaces before printing numbers

the second inner loop prints the stars.

Try without using ‘x’.

0 blanks →	*	*	*	*	*	*	*	*	*	*	*
1 blanks→		*	*	*	*	*	*	*	*	*	
2 blanks→			*	*	*	*	*	*	*		
3 blanks→				*	*	*	*	*			
4 blanks→					*	*	*				
5 blanks→						*					

```
N=6;
for(X=N, i=0; i<N; i++)
{       for(j=0;  j<i ; j++)  // loop to print blanks
              printf(“ “);     // single space
        for(j=0;  j<2*X;  j++)
              printf(“*“);
        printf(“\n”);
         X=X-2;
}
```

Printing multiplication tables from 5 to 12

```
void main()
{       int n,i;
        for(n=5; n<=12; n++)                    // loop to print tables from 5-12
        {
                for(i=1; i<=10; i++)            // loop to print 10 terms in the table
                    printf("\n %d * %d = %d", n, i, n*i);
                printf("\n\n");                 // inserting two extra new lines after each table
        }
}
```

Printing multiplication table(s) for a desired values

This program continuously accepts table numbers from k.B until zero as end of program. When zero is entered then loop stopped by break and program gets closed. For every input, related table is printed.

```
void main()
{   int n , i ;
    while(1)
    {       printf("\n\n enter table number (0 to end ):");
            scanf("%d", &n);
            if(n==0)  break;
            for(i=1; i<=10; i++)
                printf("\n %d * %d = %d", n, i, n*i);
    }
}
```

Printing prime numbers list between 50-100

```
void main()
{       int n,i,bool;
        for(n=50; n<=100; n++)      // loop to print prime numbers from 50-100
        {       bool=1;             // let us assume 'n' is prime, so bool=1
                for(i=2; i<=n/2; i++)  // loop to check prime-ness of 'n'
                {       if(n%i==0)
                        {       bool=0;
                                break;
                        }
                }
                if(bool==1)  printf("%d ", n);
         }
}
```

The outer loop is used for tracing all numbers between 50 to 100 whereas the inner loop is used to check whether each number (n) is prime or not. This inner loop checks prime-ness by dividing from 2 to n/2.

Printing twin prime numbers between 2-100

The twin prime numbers are 3-5, 5-7, 11-13, 17-19 …. the difference of their numbers are 2

Logic: initially, let us take first prime 2 (say previous=2), now start finding primes from 3 to 100 (N=3 to 100)
if N is prime then check out if N-previous==2 or not? If yes then print previous & N as twin primes.
Now take this current N value as 'previous' for next cycles.

```
void main()
{       int n, i, prev, bool;
        prev=2;                     // let us take firs prime 2 for 'prev'
        for(n=3; n<=100; n++)       // loop to print prime numbers from 3-100
        {       bool=1;             // let us assume 'n' is prime, so take bool=1
                for(i=2; i<=n/2; i++)   // loop to check prime-ness of 'n'
                    if(n%i==0)
                    {   bool=0;
                        break;
                    }
                if(bool==1)
                {    if(n-prev==2)
                         printf("\n  %d  -  %d ", prev, n);
                     prev=n;        // taking current prime as previous prime for next cycle
                }
        }
}
```

Adding sum of digits in a given number until it gets single digit

ip: 19999 op: 1

process: 1+9+9+9+9 → 37 → 3+7 → 10 → 1+0 → 1

```
void main()
{       int n, sum;
        printf("\n enter number :");
        scanf("%d", &n);
        while(n>9)          // loop to add repeatedly until n becomes single digit, if n>9 means more digits
        {       sum=0;
                while(n>0)  // loop to add all digits in 'n'
                {       sum=sum+n%10;
                        n=n/10;
                }
                n=sum;
        }
        printf("\n sum = %d", n);
}
```

Menu driven program to find given number is odd/even, Palindrome, etc

Menu driven program to find given number is odd/even, Palindrome, Prime, Armstrong, and perfect or not;

Even: The number is divisible by 2 (remainder is zero)

Palindrome: If 'n' and its reverse are equal then it is called palindrome

Prime: The number has no divisibles other than 1 and itself

Armstrong: sum of cubes of digits equal to given number (153 → 1^3+5^3+3^3 → 153)

Perfect: sum of factors equal to given number like 6 (1+2+3→6)

While executing the program, the menu appeared as given below

```
        Menu run
   ==========================
        1. Even/add
        2. Palindrome
        3. Prime or not
        4. Armstrong
        0. exit
           Enter choice [1,2,3,4,0]:
```

```
void main()
{       int N,i,t,r,rev, sum, choice;
        printf("\n enter N value:");
        scanf("%d", &N);
        while(1)
        {       printf("\n\n  Menu run    ");
                printf("\n ==========================");
                printf("\n 1.Even/Odd ");
                printf("\n 2.prime or not");
                printf("\n 3.Palindrome");
                printf("\n 4.Armstrong");
                printf("\n 0.exit");
                printf("\n Enter choice [1,2,3,4,5,0]: ");
                scanf("%d", &choice);
                printf("\n\n output is: ");
                switch( choice )
                {       case 1:  if( N%2==0 ) printf("Even");
                                 else printf("Even");
                                 break;

                        case 2: rev=0;  t=N;
                                while(t>0)
                                {    rev = rev*10 + t%10;
                                     t = t/10;
                                }
                                if(N==rev)  printf("palindrome");
                                else printf("not palindrome");
                                break;
```

```
                    case 3:  for(i=2; i<=N/2; i++)
                             {      if(N%i==0)
                                           break;
                             }
                             if(i>N/2) printf("prime");
                             else printf("not prime");
                             break;

                    case 4:  t=N; sum=0;
                             while(t>0)
                             {    r=t%10;
                                  sum=sum + r*r*r;
                                   t=t/10;
                             }
                            if(sum==N) printf("Armstrong");
                            else printf("not Armstrong");
                            break;

                    case 0:  exit(0);      // program closes
                   default:  printf("invalid input");
               }
               printf("\n\n");
          }
     }
```

Single Dimensional Arrays

In real world, we often need to maintain collection of data such as marks of all students in a class, items price at supermarket, matrices data, etc. To handle this collection of data, arrays are the alternative. The word array means, collection of homogenous items arranged sequentially one after other.

In computer science, an array is a collection of adjacent memory locations used to store and access several similar values using single name. It is a mechanism to handle collection of similar values with a single name. The new definition is, array is a collection-type data-type for handling collection of same type values. According to mathematics terminology, array is a **vector** and single variable is a **scalar.**
For example**, [13, 26, 7, 15, 10, 67, 984]** are array of integer values.

Space for all items of array is allocated in contiguous memory locations in the RAM and each item is accessed with their index value. For above values, the index of 13 is 0, 26 is 1, and 7 are 2 and so on.

Arrays can be extended to any number of dimensions, but most of the time, we use single or double dimensional arrays. Syntax to declare array is:

int X[5]; → single dimensional array int Y[5][6]; → two dimensional array

X[0]	X[1]	X[2]	X[3]	X[4]

Y[0][0]	Y[0][1]	Y[0][2]	Y[0][3]	Y[0][4]	Y[0][5]
Y[1][0]	Y[1][1]	…	…	…	Y[1][5]
Y[2][0]	…	…	…	…	Y[2][5]
Y[3][0]	…	…	…	…	Y[3][5]
Y[4][0]	…	…	…	…	Y[4][5]

Single dimensional arrays

Syntax to define one-dimensional array variable is: **data-type array-name[array-size];**

for example **int X[7];**

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]

Here, the **'X'** is an array name, which holds 7 **int**eger items. The declaration of array is same as any other normal variable except the **array-size.** The size should be mentioned with constant value and which represents count of elements that are collectively created as array.

Accessing array elements

Each value in the array is accessed independently using its index value with the array name. Especially in C, the index ranges from **0** to **arraysize**-1.

The syntax to access array element is: array-name[index]

13	**26**	**7**	**15**	**10**	**67**	**984**
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]

For example,
the expression x[0] accesses the 1st element in array. Here x[0] →13
the expression x[1] accesses the 2nd element in array. Here x[1] →26
the expression x[i] accesses the i+1th element in array.

initialization of array VS assignment of array

Assigning values at the time of array declaration is said to be **"array initialization",** whereas assigning values to specific element after array declaration is said to be assignment.

array initialization example	**array assignment example**

int a[5]={10, 20, 30, 40, 50 };	int a[5];
	a[0]=10;
	a[1]=60;
	a[4]=70;
	un-initialized cells contain garbage values.

More about array initialization: Syntax is: **data-type array-name[array-size]={ list of initial values };**
note: the list of initial values should be surrounded by pair of braces { }

int a[5]={10,20,30,40,50};

10	20	30	40	50
a[0]	a[1]	a[2]	a[3]	a[4]

int a[ ]={10,20,30,40,50};

10	20	30	40	50
a[0]	a[1]	a[2]	a[3]	a[4]

size of array is nothing but number of values assigned to array

int a[5]={10,20,30};

10	20	30	0	0
a[0]	a[1]	a[2]	a[3]	a[4]

zeroes filled by compiler for last two

int a[5]={9};

9	0	0	0	0
a[0]	a[1]	a[2]	a[3]	a[4]

zeroes filled by compiler

int a[3]={10,20,30,40,50};

10	20	30	40	50
a[0]	a[1]	a[2]	a[3]	a[4]

compiler shows error, because
the array size < count of values

About array size

At the time of compilation, the space of variables including arrays logically arranged in an order how they declared in the program. For example, **int a, b, c[5], d, e;**

A	B	c[0]	c[1]	c[2]	c[3]	c[4]	d	e
2byte	2byte	←		10 bytes		→	2byte	2byte

Here totally 18 bytes of memory is allocated for all these variables including arrays, generally, these occupy continuous locations as shown in picture. If array size "int c[5]" is not given then compiler can't allocate space for next variables (d,e). Because, at compile time all variable's space is arranged logically one after the other, if you don't mention array-size then compiler cannot allocate space for adjacent variables (here d,e). So array-size must be given while declaration and it must be constant, if not, we get compile time error.

Let us see some valid and invalid declarations

eg1) int a[5]; // valid declaration

eg2) int a[ ]; // error, size must be given at coding time (empty array not allowed)

eg3) int n=5;
int a[n]; // invalid declaration, the size must be constant, the 'n' value '5' is available at the time of running, but space is created at the time of compilation, so error)

```
eg4)  printf("enter n value :");
      scanf("%d", &n);
      int a[n];      // this is also a fatal error
```

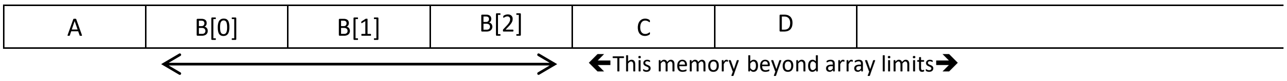
Note: These arrays are called static-sized-arrays(fixed size arrays), these are used when we know the size of array at coding time. For examples phone number with 12 bytes, name with 30 bytes, pincode with 6 bytes, and address with 50 bytes.

Sometimes, we can't expect the array-size(data-size) at coding time, then dynamic arrays are the alternative. We will see at end of chapters [malloc(), calloc(), realloc(), free()]

Checking array boundaries

The compiler does not check whether the array index is in valid range or not, because such provision not provided/available in C, therefore, it is the programmers responsibility to check array index before running a program. Ensure that, the index value must lie within the range of 0 to **arrSize**-1, otherwise program may crash due to accessing unauthorized memory which is out of range. For example,

int A, B[3], C, D;



```
A=10;  C=20;  D=40;
B[1] = 300;     // valid expression
B[2] = 400;     // valid expression
B[3] = 500;     // in-valid expression
B[4] = 600;     // in-valid expression
B[-1] = 700;    // in-valid expression
```

The last two expressions B[3]=500, B[4]=600 crosses the array limits and possibly they enter into (C,D)'s memory, the expression B[3]=500 indirectly puts 500 in 'C' and B[4]=600 indirectly puts 600 in 'D's memory. the B[-1] indirectly puts 700 in 'A' . In this way, some array indexes infiltrate into next memory cells and may corrupt their data or code. If program's code(instructions) is damaged and processor tries to execute such damaged instructions then it leads to program crash.

Scanning '5' values to array

```
int a[5], i;
for(i=0; i<5; i++)
{   printf("enter value of a[%d]:" , i );
    scanf("%d", &a[i] );
}
input:   enter value of a[0] : 7↵
         enter value of a[1] : 18↵
         enter value of a[2] : 12↵
         enter value of a[3] : 10↵
         enter value of a[4] : 15↵
```

Some people like this style

```
    int a[5], i;
    printf("enter 5 values :");
    for(i=0; i<5; i++)   // some people like this style
    {   scanf("%d", &a[i] );
    }

input: enter 5 values: 7   18   12  10   15 ↵
```

We can scan like scanf("%d%d%d%d%d", &a[0], &a[1], &a[2], &a[3], &a[4]); but this leads to complex(heavy)

Scanning ‘n’ values to array

Sometimes we can’t expect the size or count of input values, the count may vary day to day or time to time, in that case, first we scan the count of input values and then values of array. For example, in a college, the class room capacity is 60, but count of joining students may vary year to year, say 20-to-60. In this case, first we scan the count of joined students and then their marks data. Following example explains how to scan marks of ‘n’ students and printing average marks of a class.

```
void  main()
{   int a[60], i, n, sum=0;
    printf(“enter no.of students joined:”);
    scanf(“%d”, &n);
    if( n>60 )
    {   printf(“error, array size <60 ”);
        exit(0);        // closes the program
    }
    printf(“ enter marks of %d students :”,  n );
    for(i=0; i<n; i++)
        scanf(“%d”, &a[i] );
    for(i=0; i<n; i++)
        sum=sum+a[i];
    printf(“average marks is %f”,  (float) sum/n);
}
```

```
ip: enter no.of students joined: 6 ↲
     enter marks of 6 students: 79   56  67  90  89  78  ↲
op: average marks is 75.90
```

How arrays are useful?

To understand the benefits of arrays, consider the following problem of reading marks of 4 students and then displaying their total & average.

```
void main( )
{       int m1,m2,m3,m4, total, avg;
        printf(“enter marks 1:”);
        scanf(“%d”, &m1);
        printf(“enter marks 2:”);
        scanf(“%d”, &m2);
        printf(“enter marks 3:”);
        scanf(“%d”, &m3);
        printf(“enter marks 4:”);
        scanf(“%d”, &m4);
        total=m1+m2+m3+m4;
        avg=total/4;
        printf(“\n total = %d, average = %d”, total, avg );
}
```

The above program looks simple since we are taking only the marks of 4 students. If more students exist then it leads to hard coding and cumbersome in writing many **printf()** and **scanf()** functions. For this kind of problems, arrays are the alternative. Let us try the above program with arrays,

```
void main()
{       int m[4], total, avg, i;
        for( i=0; i<4;  i++)
        {   printf(“enter marks %d ::” , i );
            scanf(“%d”, &m[i]);
        }
        for(i =0; i<4; i++)
            total = total + m[i];
        avg = total/4;
        printf("\n total = %d, average = %d", total, avg);
    }
```

The size of program will remain same even if the number of students increases, also it leaves the code simple & readable.

Finding sum, big and small of N values in the array

input: enter how many input values : 6 ↲ (no.of input values to the array)

enter 6 values to array : 4 12 43 6 19 10 ↲

output: sum = 94, bigg = 43, small = 4

* to sum all values of array, take variable 'sum=0' and do **sum=sum+a[i],** where i=0 to n-1

* to find big value, take variable 'bigg=0' and compare if bigg<a[i] then take bigg=a[i].

```
    void main()
    {       int a[10], sum, bigg, smalll, i, n;      // bigg is to hold big-value, smalll is to hold small-value
            printf("enter how many input values to be scanned to array:");
            scanf("%d", &n);
            if( n>10)
            {   printf("error, insufficient array size, should be below 10 ");
                exit(0);        // closes the program
            }
            // now scanning each element one by one to the array
            printf("enter %d values to array :", n);
            for( i=0; i<n; i++)
                    scanf("%d", &a[i]);
            sum=bigg=smalll=0;      // to clean the garbage with zero
            for(i=0; i<n; i++)
            {       sum=sum+a[i];
                    if( bigg < a[i] )
                          bigg=a[i];
                    if( smalll > a[i] )
                          smalll=a[i];
            }
            printf("\n sum = %d, bigg = %d, smalll = %d", sum, bigg, smalll );
    }
```

In the above program, the **bigg** is meant to store the biggest value and **smalll** is to store the smallest value. In the for-loop, the value of **bigg** is compared with all values in the array. If any **a[i]** is found bigger than the **bigg,** then it is taken into **bigg**. In this way, finally the bigger is achieved.

If all inputs are –ve like -12, -45, -6, then the condition if(bigg<a[i]) always fails, because initial value of bigg is zero, here no array value is > zero, so output zero is printed as big value. So, to solve this problem, initialize **bigg & smalll** with any one element in the array before loop. It is better to initialize with a[0].

Finding how many 3 divisible exist in the array.

Logic: Divide each a[i] with 3 and count them when they divided perfectly, like if a[i]%3==0 then count++;

```
void main()
{       int a[50], i, n, count=0;
        printf("enter how many input values :");
        scanf("%d", &n);
        for( i=0; i<n; i++)
        {       printf("enter value of a[%d] :", i );
                scanf("%d", &a[i]);
        }
        for( i=0; i<n; i++)
        {       if( a[i]%3 ==0 )
                        count++;
        }
        printf("no.of 3 divisible are: %d", count);
}
```

input: enter how many input values : 4

```
                enter value of a[0] : 7
                enter value of a[1] : 18
                enter value of a[2] : 12
                enter value of a[3] : 10
```

output: no.of 3 divisible are: 2

Finding at least one value in the array is –ve or not?

input: enter how many input values:4

```
    enter value of a[0] : 7
    enter value of a[1] : -18
    enter value of a[2] : 12
    enter value of a[3] : 10
```

output: yes, –ve value exist

```
void main()
{       int a[50], i, bool=0, n;
        printf("enter how many input values :");
        scanf("%d", &n);
        for(i=0; i<n; i++)
        {       printf("enter value of a[%d] :", i );
                scanf("%d", &a[i]);
        }
        for( i=0; i<n; i++)
        {       if( a[i] < 0 )
                {       bool=1;
                        break;
                }
        }
        if(bool==1) printf("yes, –ve value exist");
        else  printf("no, -ve is not exist");
}
```

Finding no.of days in a given month

```
  ip: 2   2003                 ip: 4  2009
op: 28 days                  op: 30 days
    void main()
    {   int m;
        int  days[]={0,31,28,31,30,31,30,31,31,30,31,30,31};
        printf("enter month:");
        scanf("%d", &m);
        printf(" days in month is %d", days[m] );
    }
```

The array **"days[]"** holds the number of days in each month. The days in a month initialized to array is relative to the month-index, for example

a[1] → January,

a[2] → February,

a[0] → 0 → dummy value..

note: here leap-year in not considered.

Checking given date is valid or not?

```
  ip: 30  2   2001             ip: 30  4   2010         ip: 31  12   2021
op: invalid date             op: valid date           op: valid date
    void main()
    {       int  arr[]={0,31,28,31,30,31,30,31,31,30,31,30,31};
            int d, m, y;
            printf("enter a date :");
            scanf("%d%d%d" , &d , &m , &y);
            arr[2] = 28 + (y%4==0);      // February has 29 days in leap year,  if  y%4==0 is true then it gives 1.
            if( m<0 || m>12 || d<1 || d>arr[m] )
                printf(" invalid date");
            else   printf( " valid date");
    }
```

Finding ^NC_R of two numbers

Ip: 4 2 ip: 7 1

op: $^NC_R = 6$ op: $^NC_R = 1$

```
void main()
{     long int    a[10]={ 1,1,2,6,24,120,720,5040,… };
      int n, r, result;
      printf("enter n & r values :");
      scanf("%d%d", &n, &r);
      result = a[n] / (a[r]*a[n-r]);
      printf("\n NCR = %d", result);
}
```

Initially, the array filled with already calculated factorial values. Therefore, the $^N\mathbf{C}_R$ value is easily obtained by substituting in the equation from calculated values. Finally, the result is printed.

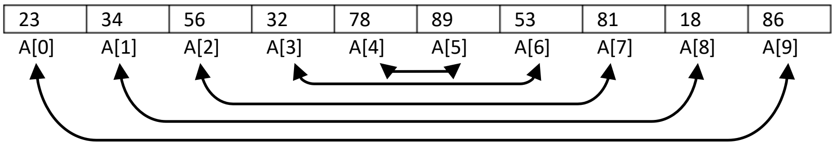
Reversing array elements

ip: 23 34 56 32 78 89 53 81 18 86

op: 86 18 81 53 89 78 32 56 34 23 (reversed array)

Program accepts **'n'** numbers from keyboard and then reverses the elements of the array.

To reverse the elements, swap them in opposite directions i.e., swap the first element with the last element, second element with the previous of last, and so on. Like shown picture



23	34	56	32	78	89	53	81	18	86
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

```
void main()
{       int a[20], i, j, n, temp;
        printf("enter no. of  input values n :");
        scanf("%d", &n);
        printf ("enter %d values to array :", n);
        for( i=0; i<n; i++)
            scanf("%d", &a[i]);
        for(i=0, j=n-1;  i<j; i++, j--)
        {   temp=a[i];
            a[i] = a[j];
            a[j] = temp;
        }
        printf("\n after reversing, the array elements are \n");
        for(i=0; i<n; i++)
            printf("%d  ",  a[i]);
}
```

Deleting kth position element in the array

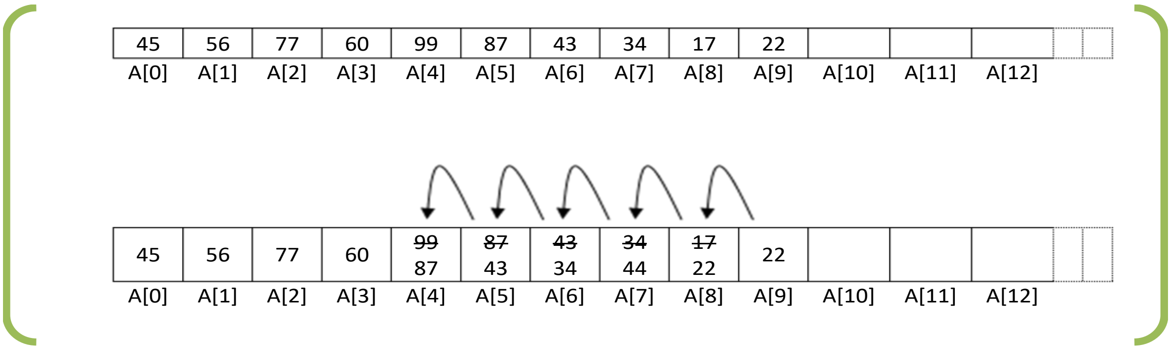
ip: 23 34 56 32 78 89 96

op: 23 34 56 32 89 96 //after deleting 5th element (78)

The following array contained 10 values, and it gives demo how to delete 5th element.

The code to delete 5th element is as follows

```
for(i=5; i<10; i++)
      a[i-1] = a[i];      //  it replaces  a[4] by a[5],   a[5] by a[6], ...etc.
```



45	56	77	60	99	87	43	34	17	22			
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]

45	56	77	60	~~99~~ 87	~~87~~ 43	~~43~~ 34	~~34~~ 44	~~17~~ 22	22			
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]

Now write a program yourself, where accept N values from KB and delete kth element in the array(k<N) later print all elements after deleting kth element.

Inserting a new element at kth position

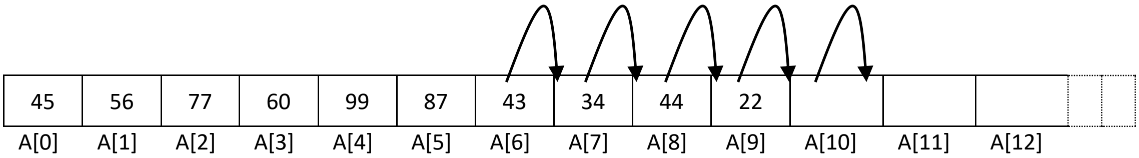
ip: 45 56 77 60 99 87 43 34 44 22

op: 45 56 77 60 99 **‘11’** 87 43 34 44 22 // after inserting 11 at 6th position

Program inserts a new element at kth position in an existing array of ‘n’ elements, where k<n.

To insert a new element at A[k], shift all existing elements of A[k], A[k+1], A[k+2]...a[N-1] to the right side by one position. So that we get a gap at A[k], where new element can be inserted.

For example, to insert a new element 11 at A[5], shift all elements 22,44,34, 43, 87 to the right side by one position, so that we get a gap after 87, where 11 can be inserted.



```
void main()
{       int a[20],i, k, new;
        printf(“enter no.of input values n :”);
        scanf(“%d”, &n);
        printf(“enter %d values to array : “, n);
        for( i=0; i<n; i++)
            scanf(“%d”, &a[i]);
        printf(“enter new element and position to insert :”);
        scanf(“%d%d”, &new, &k);
        for(i=n; i>=k; i--)       // shifting elements to right-side
             a[i] = a[i-1];
        a[k-1]=new;               // inserting new element
        n++;                      // now array contains n+1 elements
        printf(“\n after inserting elements, the array is :”);
        for( i=0; i<n; i++)
            printf(“%d  “, a[i] );
}
```

Printing factors of each number in given array of 5 values

ip: 14 16 13 11 24

```
op:   14 => 1, 2, 7, 14
      16 => 1, 2, 4, 8, 16
```

```
    for(i=0;  i<5;  i++)
    {       printf(“\n factors of %d => “, a[i] );
            for( j=0;  j<=a[i] ;  j++)
            {       if( a[i] % j == 0 )
                            printf(“%d  “,  j );
            }
    }
```

Printing list of primes in given array of values

```
Ip:  23   11   19  20  87  65  42
op:  11   19   87     // primes
void main()
{   int a[20], i, bool, j;
    printf("enter the no.of input values n :");
    scanf("%d", &n);
    printf("enter %d values to array :", n);
    for( i=0; i<n; i++)  scanf("%d", &a[i]);
    for(i=0; i<n; i++)                // loop to check each number in the array,  whether it is prime or not?
    {       bool=1;                   // let a[i] is prime, so set bool to 1
            for(j=2; j<=a[i]/2; j++)  // loop to check a[i] is prime or not
            {       if( a[i]%j==0)
                    {       bool=0;
                            break;
                    }
            }
            if(bool==1)
                    printf("%d   ", a[i] );          // if not divided anywhere, it is prime
       }
    }
```

i-loop is to check all numbers in the array,

j-loop is to check each a[i] is prime or not?, it is by dividing from 2 to a[i]/2

Printing common elements of two arrays

```
 ip:  23   34   45   32   78   44   72   85
      21   44   55   62   56   23
op: 23   44
```

Let us say, two arrays a[ ] & b[ ] contained n1 & n2 number of elements respectively.

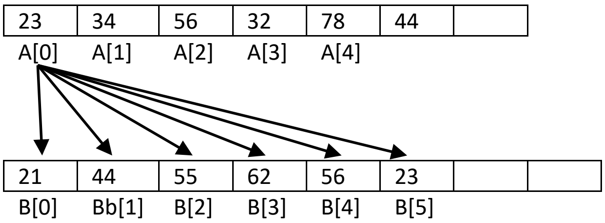
step1: compare a[0] with all the elements of b[ ], if found anywhere then print it.

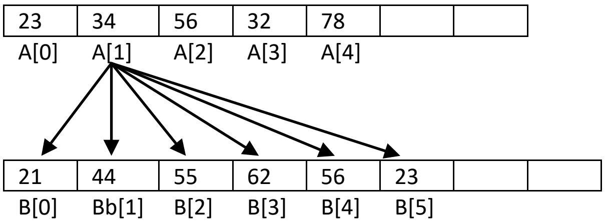
step2: compare a[1] with all the elements of b[ ], if found anywhere then print it.

In this way elements are compared and common elements are printed.

Note: Let each array contained distinct elements (no duplicate exist)

The following figure and code shows how the comparison is done for every element.





The code is,

```
for(i=0; i<n1; i++)
      for( j=0; j<n2; j++)
      {     if( a[i] == b[j] )
            {   printf( "%d  ", a[i]);
                break;
            }
      }
```

Printing all digits of N in ascending order

30) Let N value has non repeated digits like 982463 , now just print all digits in ascending order 234689

step1: take array with size 10, and initialize all cells with 0. This is like: int a[10]={0};

step2: now take digits one by one from N, if digit is 3 then set a[3]=1, in this way fill array cells with 1.

step3: now print all array index values of ‘i’ where a[i]==1 , here i=0,1,2,3,…,9

if N is 982463, let us see how array cells filled with 1’s

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
0	0	1	1	1	0	1	0	1	1

```
int   a[10]={0};
int   n;
scanf(“%d”, &n );
while( n>0 )
{       a [ n%10 ]=1 ;
        n=n/10;
}
for( i=0; i<10; i++)
{       if( a[i]==1 )
                printf(“%d  “, i );
}
```

Finding standard deviation from set of values

ip: A[] = { 7 , 1 , -6 , -2 , 5 }

op: result=4.69

The formula for standard deviation is sqrt $(\sum(A[i]–\text{mean } A[])^2/N)$

Let the average(mean) of all values are M, the formula as given below

sd = sqrt $((A_0-M)^2 + (A_1-M)^2 + (A_2-M)^2 + (A_3-M)^2 + \ldots + (A_n-M)^2 / N)$

```
void main()
{      int a[20], i, N, mean;
        float sum, sd;
       printf("enter the no.of input values N :");
       scanf("%d", &N);
       printf("enter %d values to array :", N);
       for( i=0; i<N; i++)
           scanf("%d", &a[i]);
       for(i=sum=0; i<N; i++)           // calculating sum of all values
       {    sum=sum+a[i];
       }
       mean=sum/N;                      // calculating mean value
       for(i=sum=0; i<N; i++)           // calculating sum of squares to convert –ve values to +ve
       {    sum = sum + (a[i]-mean)*(a[i]-mean);
       }
       sd=sqrt(sum/N);
       printf("standard deviation is %.2f", sd);
}
```

Functions

Function is a subprogram that performs a given sub-task in a program. It is an independent block of code with a name to perform a specific task. Functions mainly provide reusability and modularity. It breaks down a big program into several sub programs for easy handling. Thus, collection of functions makes a good C-program. In computer science, the function is also called routine, subroutine, subprogram, procedure, or method or module.

While developing a big application, it should be designed in such a way that, it is divided into several meaningful modules and each module again divided into several sub-modules called functions.
For example, let us consider student automation project in a school, it can be divided into several modules say admission, attendance, exam-test, fees, games, etc. Each of these modules is made up of several functions which accomplish individual tasks. Generally, every module is developed in separate files like "admission.c", "attendance.c", "exam-test.c", etc; all these files may developed by several people and finally integrated to build the whole project.

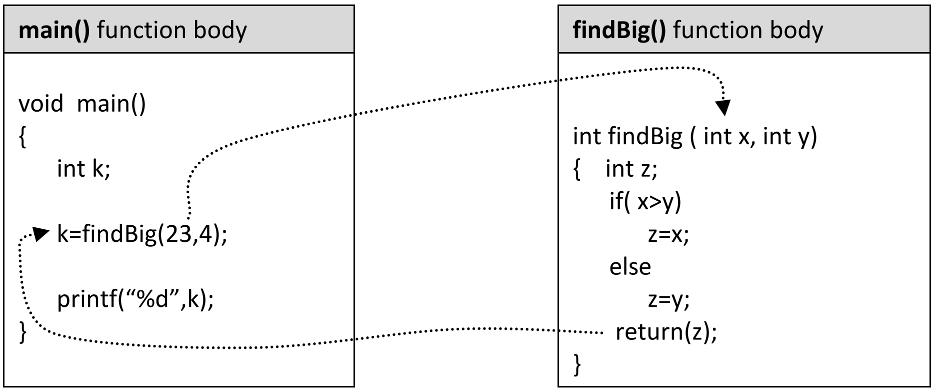
Types of functions

Functions are classified into two types: **1. Library functions 2. User-defined functions**

Library functions: These are ready-made functions, which are designed and written by the C manufactures to provide solutions for basic and routine tasks in the programming. For example I/O functions printf() & scanf(), mathematical functions pow() & sqrt(), etc. The vendor of C, supply these predefined functions in compiled format with C software. There is huge collection of functions available to meet all requirements in the programming, thus this collections are called functions Library.

User-defined functions: We can write our own functions as per our requirement just like any library function. There is no conceptual difference between user-defined and library functions, all works in similar manner, besides, our functions can be added to existing library and we can use like a library function.
If more collections found, then we can create our own library.

In C, the main() is also a user defined function, the word '**main'** reserved by the complier but its body has to be implemented by the programmer. The main() function works as start & end point of program, and remaining functions are executed by transferring(call) the control temporarily from main().There is no syntax difference between main() and other functions, all works in similar way.
Let us see the following example, which illustrates how the functions are executed in the program.



main() function body

```
void  main()
{
     int k;

     k=findBig(23,4);

     printf("%d",k);
}
```

findBig() function body

```
int findBig ( int x, int y)
{    int z;
     if( x>y)
          z=x;
     else
          z=y;
      return(z);
}
```

The instruction k=findBig(4,23) is a function-call statement in main(), when it gets executed, the control jumps to the findBig() fn body along with values (23,4), and these values are assigned to (x, y) variables, after calculating big value(z), it returns and assign back to k, this is as shown above.
In this way functions are executed.

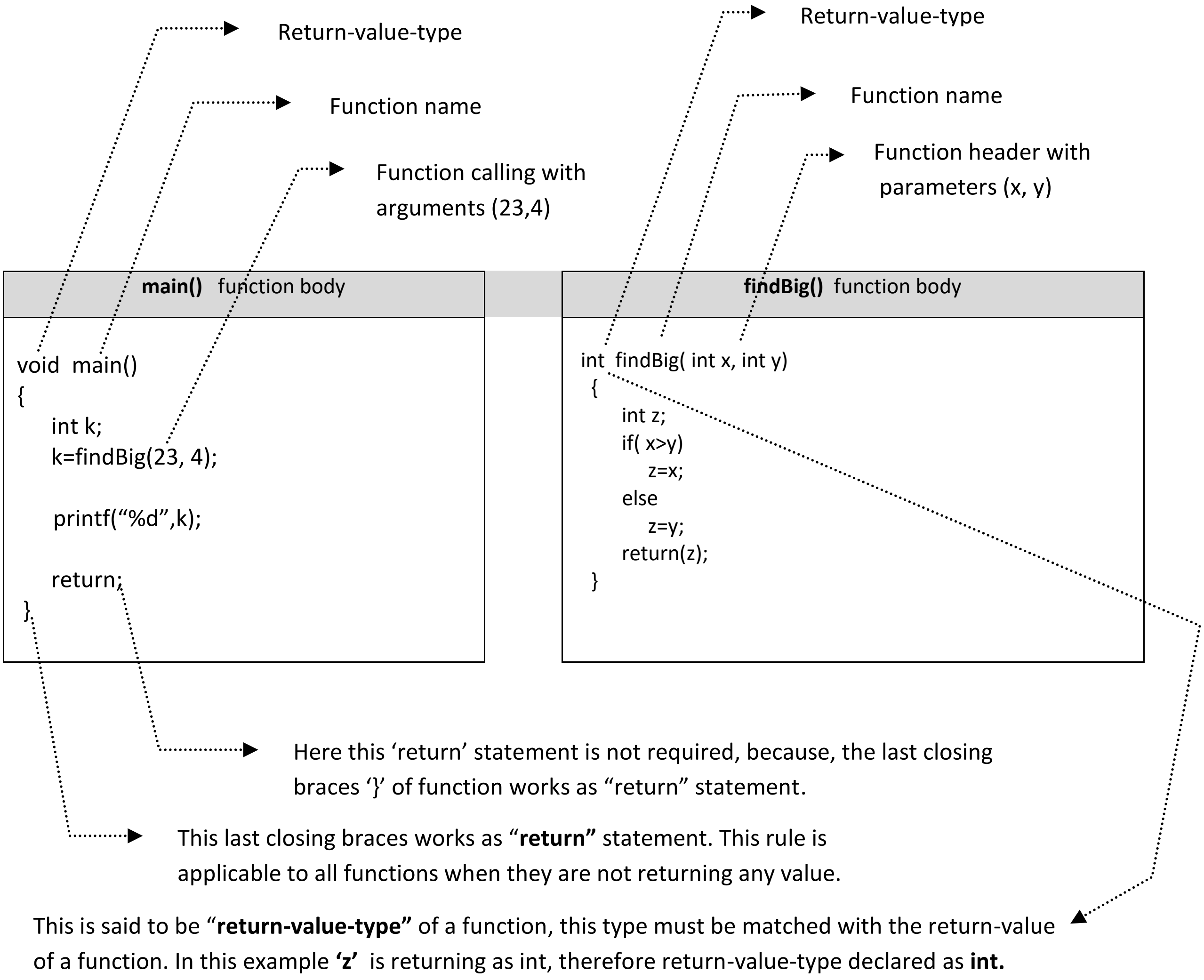
*the passing input values to function is said to be **arguments**, whereas the receiving variables of such arguments is said to be **parameters.** In this example, the values(23,4) are called arguments and (x, y) are called parameters. The arguments and parameters count must be matched, and type must be compatible.

*In this program, we have two functions **main()** and **findBig().** Here the main() is invoking(calling) the findBig(), so in this context main() is said to be **calling-function**, whereas findBig() is said to be **called-function**.

*Here the word "**findBig**" is said to be the function name and it follows the rules of variables name. We can give any name to the function just like a variable name in the program.

*The type **int** before the function name is said to be **return-value-type.** In this example, the function is returning 'z' value as int, therefore return-value-type declared as **int.** The return-type explicitly tells what type of value the function is returning. It is useful for compiler to check syntax errors as well as to the programmer for documentation.

*Thus, every function does something and returns a calculated value to the calling-function, all functions including main() follows same syntax rules with one or two optional statements. The following figure shows the each & every entity of function


Return-value-type
Function name
Function calling with arguments (23,4)
Return-value-type
Function name
Function header with parameters (x, y)
main() function body
findBig() function body

```
void  main()
{
     int k;
     k=findBig(23, 4);

     printf("%d",k);

     return;
 }
```

```
int  findBig( int x, int y)
  {
      int z;
      if( x>y)
          z=x;
      else
          z=y;
      return(z);
  }
```

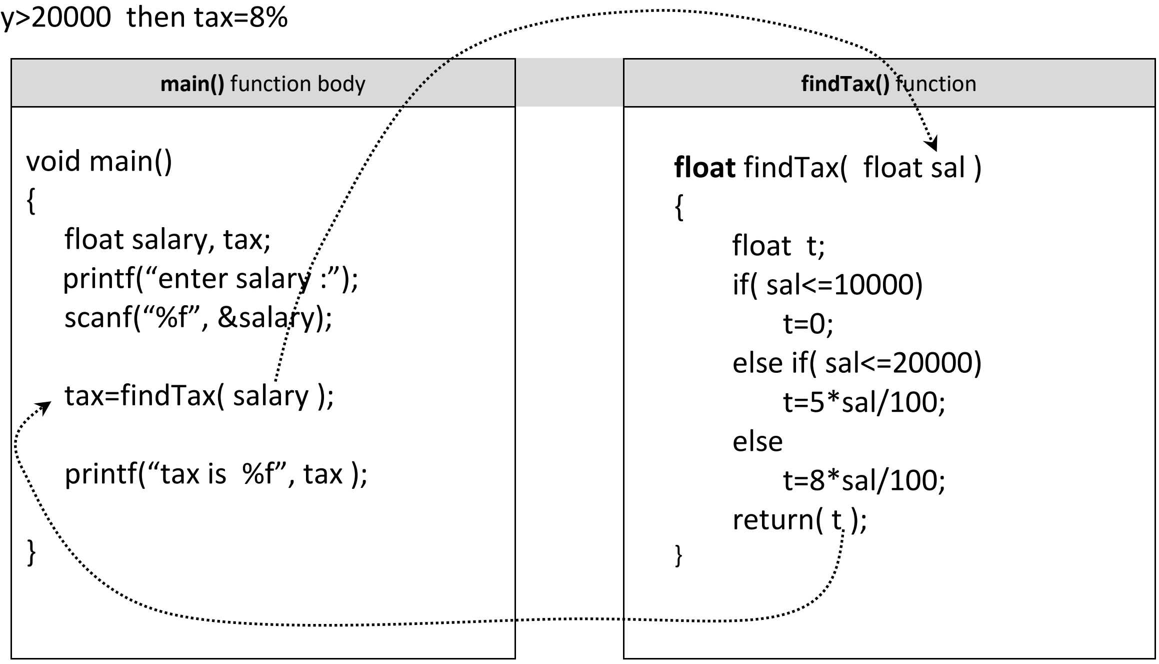
Here this 'return' statement is not required, because, the last closing braces '}' of function works as "return" statement.

This last closing braces works as "**return**" statement. This rule is applicable to all functions when they are not returning any value.

This is said to be "**return-value-type"** of a function, this type must be matched with the return-value of a function. In this example **'z'** is returning as int, therefore return-value-type declared as **int.**

Finding tax on employee salary using function

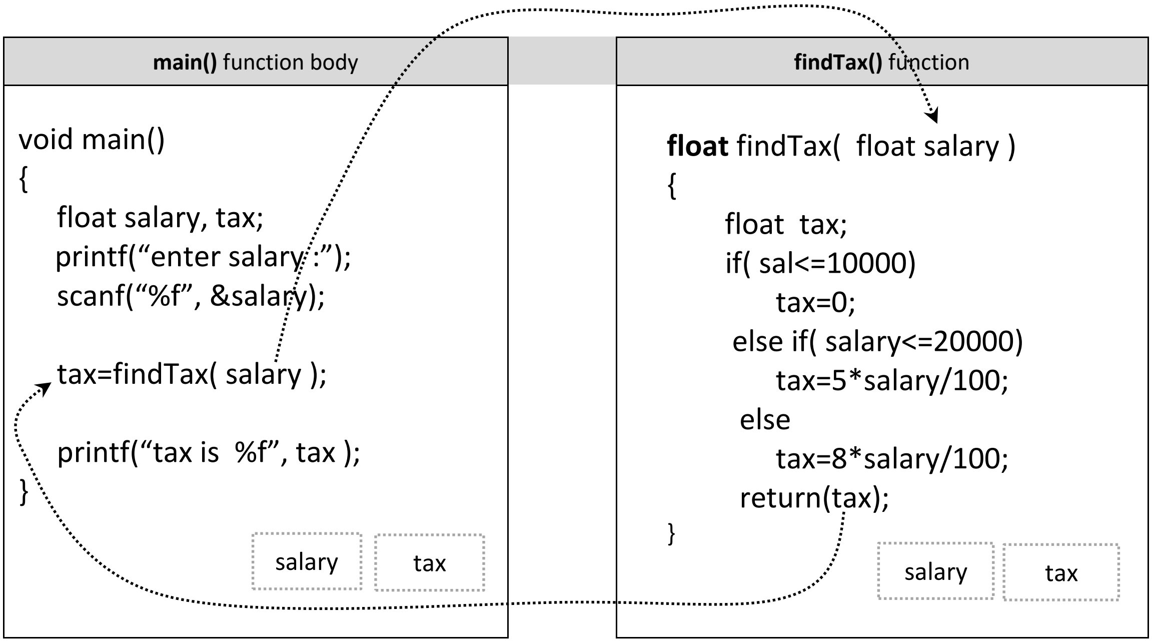
If salary<=10000 then tax=0%
if salary>10000 && salary<=20000 then tax=5%
if salary>20000 then tax=8%



```
void main()
{
    float salary, tax;
    printf("enter salary :");
    scanf("%f", &salary);

    tax=findTax( salary );

    printf("tax is  %f", tax );

}
```

```
float findTax(  float sal )
{
     float  t;
     if( sal<=10000)
          t=0;
     else if( sal<=20000)
          t=5*sal/100;
     else
          t=8*sal/100;
     return( t );
}
```

The function findTax() is taking employee salary as input(argument), and after calculating tax amount, it returns and assigns back to 'tax'. This is as shown in the above figure, here the variable 'salary' is said to be argument, whereas 'sal' is said to be parameter. In the above findBig() example, the arguments (23,4) are constants, but in this example, argument salary is a variable. Here value of salary is passed as argument to the function. Sometimes argument and parameter names can be same, but they are different copies, let us see following example



```
void main()
{
    float salary, tax;
    printf("enter salary :");
    scanf("%f", &salary);

    tax=findTax( salary );

    printf("tax is  %f", tax );
}
```

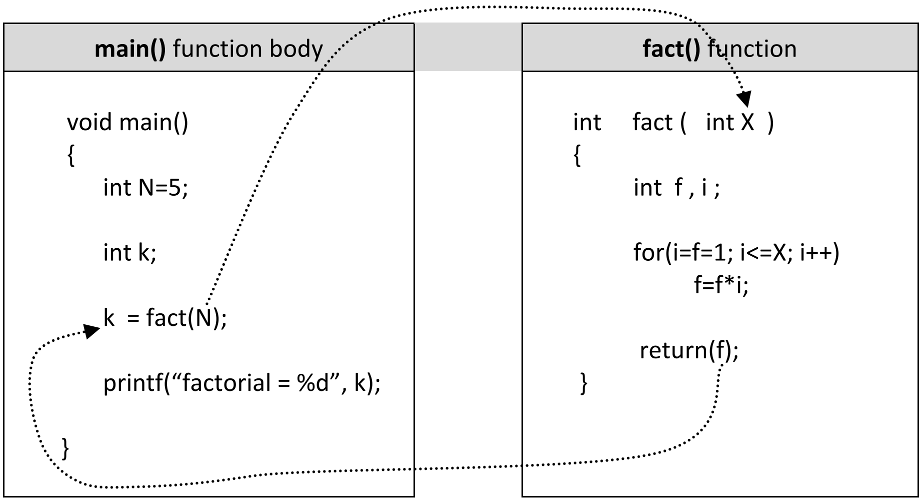
```
float findTax(  float salary )
{
     float  tax;
     if( sal<=10000)
          tax=0;
      else if( salary<=20000)
          tax=5*salary/100;
       else
          tax=8*salary/100;
       return(tax);
}
```

Here argument and parameter variable's name are same, but they are different copies, so in this total program two 'salary' variables and two 'tax' variables exist. One set belongs to main() fn and second set belongs findTax() fn (A separate memory space is created for each set like above shown picture).
So, argument variables are always different from parameter variables, parameters work like a copy of arguments. It can be imagined as parameters are place-holder of arguments. Generally, beginners get confusion when argument & parameters names are same.

Finding factorial of a given value using function

input: 5

output: 120



main() function body

```
void main()
{
     int N=5;

     int k;

     k  = fact(N);

     printf("factorial = %d", k);

}
```

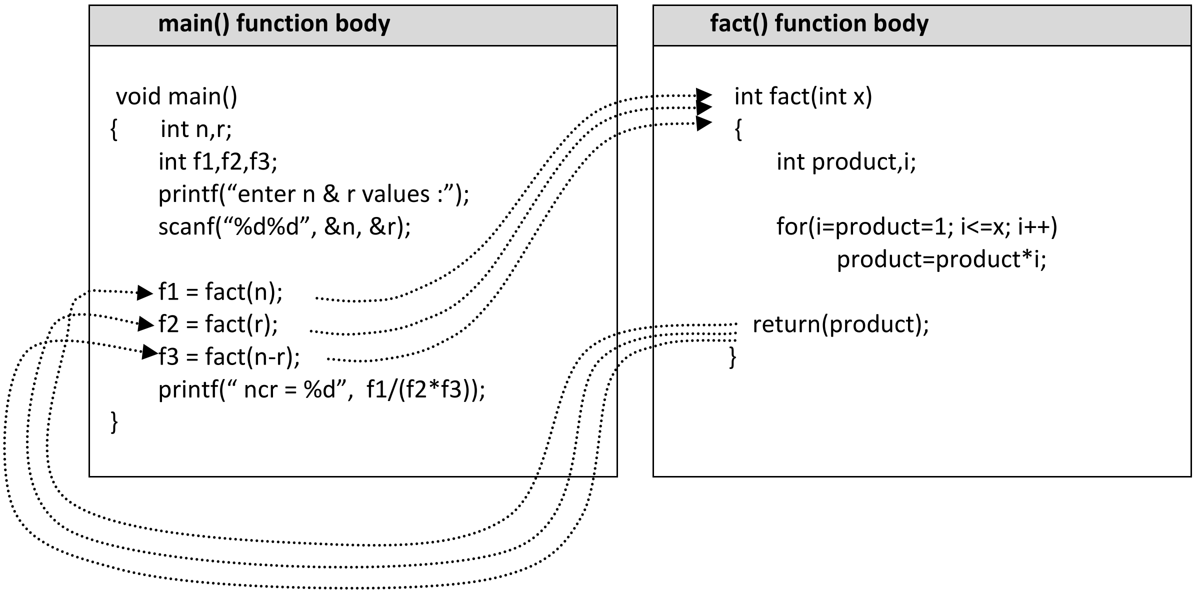
fact() function

```
int    fact (   int X  )
{
        int  f , i ;

        for(i=f=1; i<=X; i++)
                f=f*i;

         return(f);
 }
```

- The fact() function takes argument 'N=5' as input and stores into parameter **'X'**.
- After calculating factorial value, it returns and assigns back to 'k'.
- In this way, function can be called as many times as we want. Let us see in next example, how functions can be called more than once.

Finding $^{N}C_{R}$ of given N and R values

Logic: $^{N}C_{R}$ is a summation of n!, r!, and n-r!. Using fact() function, we can find these three factorials by calling 3 times. The control goes & returns three times to the fact() fn. This is as given below



main() function body

```
 void main()
{      int n,r;
       int f1,f2,f3;
       printf("enter n & r values :");
       scanf("%d%d", &n, &r);

       f1 = fact(n);
       f2 = fact(r);
       f3 = fact(n-r);
       printf(" ncr = %d",  f1/(f2*f3));
}
```

fact() function body

```
int fact(int x)
{
      int product,i;

      for(i=product=1; i<=x; i++)
              product=product*i;

   return(product);
}
```

Let input **n=7, r=3;** then function calls as given below

```
f1 = fact(7);           // here value 7 passes as argument to fact()
f2 = fact(3);           // here value 3  passes as argument to fact()
f3 = fact(7-3);         // here value 4 passes as argument to fact()
```

In first call, the control goes with argument '7' and returns 5040 (7!) and assigns to 'f1'.

In second call, the control goes with argument '3' and returns 6(3!) assigns to 'f2'.

In third call, the control goes with argument '4' and returns 24(4!) assigns to 'f3'.

In this way, reusability of code is possible with the functions whenever it is required.

Finding sum of 1 to N using function (1+2+3+4+5+ …..+N)

Following **"findSum()"** function calculates the sum of 1+2+3+…+N without using formula, this function takes '**N'** as argument and returns the sum of 1 to **N**. Later main() function checks the **"findSum()"** returned value is equal to formula N(N+1)/2 or not?. If equal then says "program is correct", or else says "program has some logical errors".

```
void main()
{   int N=5, result;
    result=findSum(N);   // function  call
    if( result==N*(N+1)/2 )
         printf("program is correct");
    else
         printf("program has some logical errors");
}
```

```
int findSum( int N)
{       int i , sum=0;
        for(i=1; i<=N; i++)
             sum=sum+i;
        return sum;
}
```

Finding summation of $x^1/1! + x^2/2! + x^3/3! + \ldots + x^N/N!$

Here we have written two functions, first one is to calculate X^i and second one is to calculate fact(i), and summation of these two functions return values gives the final result.

```
void main()
{    float  sum=0, v1, v2;
     int i;
     for(i=1;  i<=N;  i++)
     {    v1=power(x,i);
          v2=fact(i);
          sum=sum+v1/v2;
     }
     printf("sum is %f", sum);
}
```

```
int  fact( int  N)
{       int i , f;
        for(i=f=1; i<=N; i++)
             f=f*i;
        return f;
}
```

```
int  power( int x , int y )
{       int i , p;
        for(i=p=1; i<=y; i++)
             p=p*x;
        return p;
}
```

Syntax of function

Function has 3 syntaxes

1. **Function definition/body** (defines the code of function)
2. **Function calling/invoking** (executing the function task)
3. **Function proto-type** (function declaration like variable declaration, discussed at end of this chapter)

① Syntax of function-body

```
return-value-type function-name( parameters-list )
{       variable-declaration;
        instruction1;
        instruction2;
        ……..
        return(result);
}
```

The function body defines the task of function, ie, what it does. It defines function's task with input & output values. The body executes when it is called explicitly from other part of program. Every function follows this syntax with one or two optional statements.

② Syntax of function-call statement

```
variable = function-name( arguments-list );
```

* Calling a function means invoking the function task, ie, asking the function to do his respective job. When the above function-call statement gets executed, the control transfers to its body along with arguments and after executing all instruction within the body, the control returns to same point of call-statement.

* To understand the mechanism of function-call, let us consider a simple example. Suppose owner of house wants some groceries, so he calls his assistant and gives some money to him, later assistant brought and returns to him. In the first example, the owner is main() and assistant is findBig(). Here main() wants to find the big of two, so it called the findBig() and gave arguments (23,4) to him. In this way functions work.

*** Function name:** the function name should be relevant to function's task, it should express the purpose of function. For example, if a function finds square root of a value, then it is better to name it as 'sqrt()'. The body of function invokes by calling with this name.

*** Calling vs Called**: Here the main() is calling the findBig() function, so in this context, main() is said to be **calling-function** whereas findBig() is said to be **called-function**.

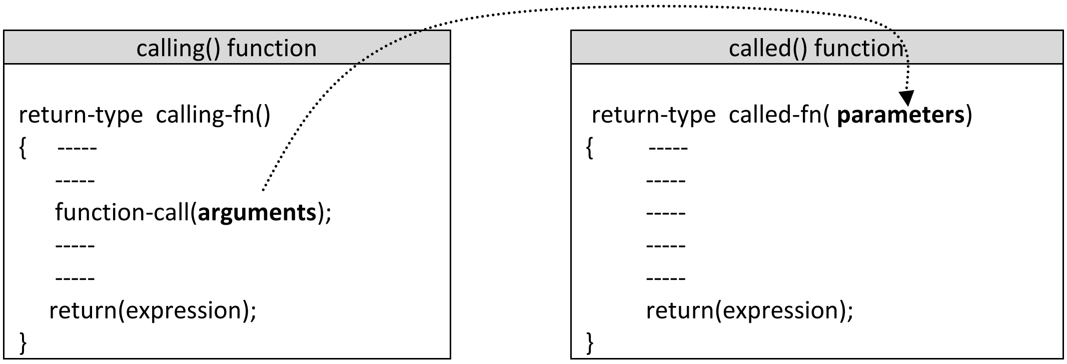
*** Arguments vs Parameters:** Arguments are nothing but input values of function. When a function is called, they are passed from **calling-function** to **called-function** along with the control. Whereas parameters are variables, which receives (hold) the argument values at called-function. (Arguments are **values** whereas parameters are **variables**)

* Arguments belong to **calling-function**, whereas parameters belong to **called-function**.

* Arguments and parameters must be matched or compatible.

* in old definitions, arguments were called actual-arguments or actual-parameters, whereas parameters were called formal-arguments or formal-parameters. (actual → versus → formal)

*** The following figure illustrates the relation between argument and parameters.**



calling() function

```
return-type  calling-fn()
{    -----
     -----
     function-call(arguments);
     -----
     -----
    return(expression);
}
```

called() function

```
return-type  called-fn( parameters)
{        -----
         -----
         -----
         -----
         -----
         return(expression);
}
```

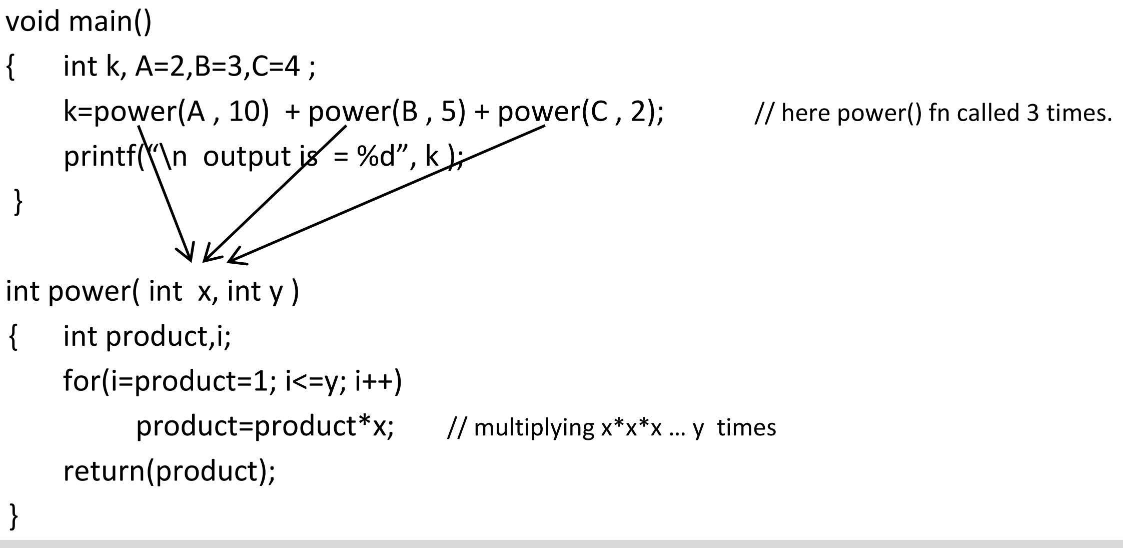
* **return** statement terminates the current task of function and returns the control to calling-function with/without a value.

* **function's-limitation:** The 'C' function can return only **one** or **none** value using 'return' statement, because the syntax provided in that way, to return more values pointers are used. (we will see later)

Note: we have a freedom to write functions in any order, that is, main() function can be written at bottom of big() function, but as the main() fn is starting point of program, the compiler automatically moves it to beginning in executable file. In this way programmer has a freedom to write functions in any order.

Finding value of equation $A^{10}+B^5+C^2$ using power() function

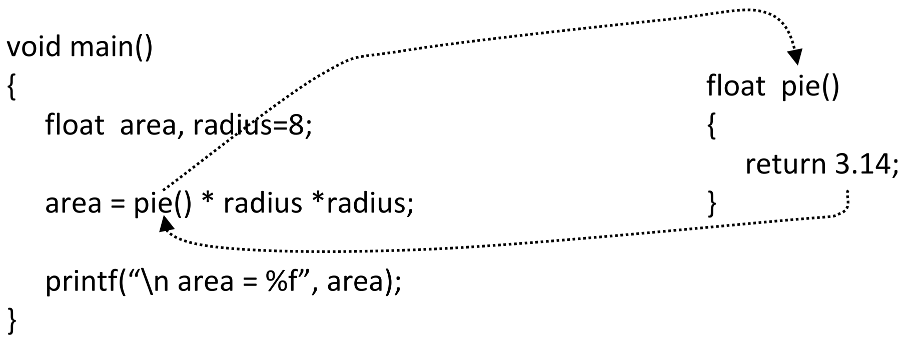
The power() function takes two arguments(x,y) as base and exponent and returns the power value.
Later the main() function calls power() to find value of equation: $A^{10}+B^5+C^2$



```
void main()
{    int k, A=2,B=3,C=4 ;
     k=power(A , 10)  + power(B , 5) + power(C , 2);          // here power() fn called 3 times.
     printf("\n  output is  = %d", k );
 }

int power( int  x, int y )
{    int product,i;
     for(i=product=1; i<=y; i++)
          product=product*x;        // multiplying x*x*x … y  times
     return(product);
}
```

A demo program how return-value substitutes at fn-call statement

If function is returning any value, then function-call statement can be used as variable or expression in the program, because the return-value substitutes in place of function-call as a variable. Let us see, how it is



```
void main()
{
    float  area, radius=8;

    area = pie() * radius *radius;

    printf("\n area = %f", area);
}
```

```
float  pie()
{
    return 3.14;
}
```

When the **pie()** function calls in the main(), it returns the float value **3.14** and substitutes at the function-call statement as "area=3.14*radius*radius"

Actually, the function's return-value is stored into a temporary variable (created by compiler) and such variable substitutes at call statement. Thus above function call can be imagined as

```
void main()
{     float  area, radius=8;
      temp=pie();         // here this 'temp' created by compiler
      area = temp * radius *radius;
      printf("\n area = %f", area);
}
```

here 'temp' is assumed name, actually it is a name-less variable created and used by compiler

Finding given number is palindrome or not?

The following function takes integer N as argument and returns the reverse of it, later main() function prints given number is palindrome or not?

```
int reverse(int n)
{       int rev=0;
        while(n>0)                  // loop to find reverse number
        {   rev = rev*10 + n%10;
            n=n/10;
        }
        return(rev);
}
void main()
{       int n;
        printf("enter n value :");
        scnaf("%d", &n);
        if( n==reverse(n) )  printf("palindrome");
        else   printf("not a palindrome");
}
```

As above told example, the function's return-value is stored into a temporary variable and such value substitutes in place of function-call. Thus, above function call can be imagined as given below

```
if( n == reverse(n) )
    printf("palindrome");
else
    printf("not a palindrome");
```

→

```
temp=reverse(n);
if( n == temp)
    printf("palindrome");
else
    printf("not a palindrome");
```

Finding big of three numbers using function

This big() function takes two arguments as input values and returns the biggest of them.
Later main() function calls this function to find biggest of 3 given numbers.

```
int big(int , int);           // fn declartion
void main()
{   int x,y,z,k;
    printf( “\n enter 3 values :”);
    scanf(“%d%d%d”, &x, &y, &z);
    k=big(x,y);                          //  k=big(x, big(y,z) );
    k=big(k,z);
    printf( “biggest = %d“, k);
}
int big(int a, int b)          // fn body
{   if(a > b) return a;
    else  return b;
}
```

The above two function-calls can be written in single line as k=big(x, big(y,z))
Let the x, y, z are 23, 42, 31, then function calls are k=big(23, big(42,31)) ➔ k=big(23 , 42) ➔ k=42.
First the inner-call **big(42,31)** will be made and it returns 42, and this value substitutes in outer-call as second argument. Finally outer-call will be made as **k=big(23,42);**

Finding given number is Armstrong or not?

This function takes integer ‘N’ as argument and returns Armstrong or not? If yes returns 1, otherwise 0.
Later the main() function scans ‘N’ as input and printing it is Armstrong or not?

```
int isArmstrong(int n)
{       int t=n, sum=0;
        while(n>0)
        {       r=n%10;
                sum=sum+r*r*r;
                n=n/10;
        }
        if( t==sum) return 1;
        else return 0;
}
void main()
{       printf(“enter n value :”);
        scanf(“%d”, &n);
        if( isArmstrong(n)==1)   printf(“yes, it is Armstrong“);
        else   printf(“no, not Armstrong”);
}
```

Printing list of Armstrong numbers between 1 to 1000 (using above fn)

Output: 1, 153, 370, 371, 407.

```
        void main()
        {       for( n=1;  n<=1000;  n++)
                if( isArmstrong(n)==1 )
                        printf("%d  ", n);
        }
```

Finding prime-ness of a given number using function

This function takes integer N as argument and returns the prime or not. If prime returns 1, otherwise, 0;

```
        int isPrime(int n)
        {     int i;
              for(i=2; i<=n/2; i++)      //checking 'n' by dividing from 2 to n/2
              {   if(n%i==0)
                      return(0);          // here N divided, so returning '0' as not-prime
              }
              return(1);                  // not at all divided,  so returning '1'  as  prime
        }
        void  main()
        {     int n;
              printf("enter n value :");
              scanf("%d", &n);
              if( isPrime(n) ==1) printf("prime number ");
              else   printf("not prime number");
        }
```

Printing prime numbers from 50 to 100 using above "isPrime()" fn.

```
    void main()
    {     int n;
          for(n=50; n<=100; n++)
          {     if( isPrime(n))             if(1) → true   ,   if(0) → false
                    printf("%d ", n);
          }
    }
```

Here isPrime() will be called 50 times with arguments 50, 51, 52, 53, 54, 55, 56, ….,100;

Printing twin primes 2 to 100 using above "isPrime()" fn.

The difference of two primes is 2 then they said to be twins. For example, 3→5, 5→7, 11→13, 17→19, etc.

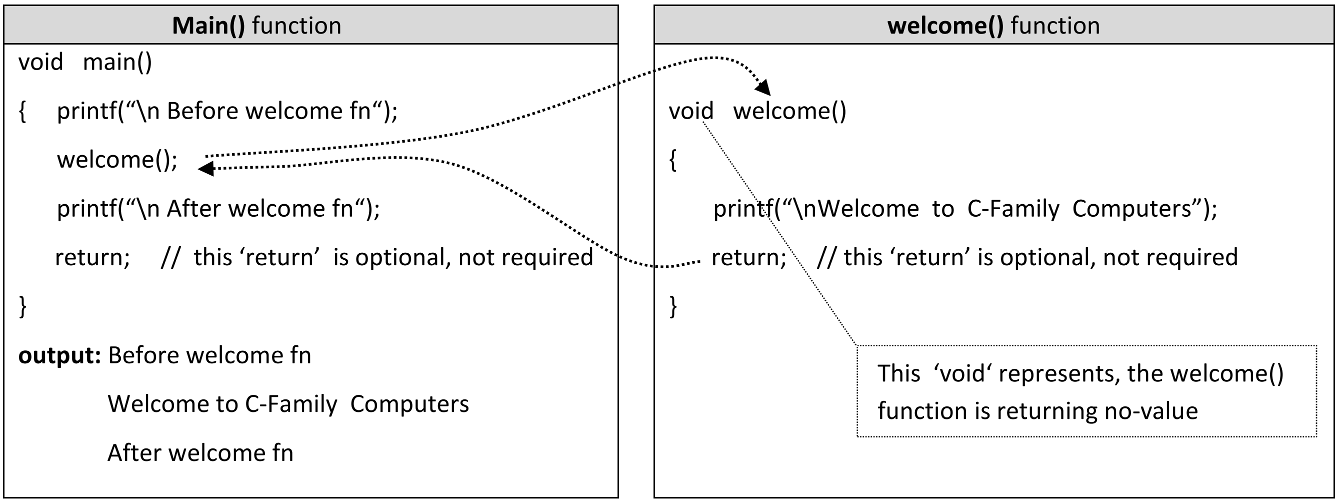
```
      for(n=2; n<100; n++)
      {     if( isPrime(n) && isPrime(n+2) )
                printf("\n  %d →  %d ", n, n+2 );
      }
```

Printing list of primes between given two limits (50 to 100)

```
void main()
{       printAllPrimes( 50 , 100 );
}
void printAllPrimes( int lower  ,  int upper )
{       int n;
        for( n=lower;  n<=upper;  n++)
        {       bool=1;
                for( i=2;  i<=n/2;  i++)
                {       if( n%i ==0 )
                        {       bool=0;
                                break;
                        }
                }
                if( bool==1)
                        printf("%d  ", n);
        }
    }
```

About "void" data-type

In case, if function is not returning any value, then the **return-value-type** should be '**void', t**he keyword **void** is also a data type like int, float, char. It represents empty-type/null-type. The following function welcome() is returning no-value, therefore we kept "void" as return-type.



Main() function

```
void   main()
{    printf("\n Before welcome fn");
     welcome();
     printf("\n After welcome fn");
     return;    //  this 'return'  is optional, not required
}
```

output: Before welcome fn

Welcome to C-Family Computers

After welcome fn

welcome() function

```
void   welcome()
{
     printf("\nWelcome  to  C-Family  Computers");
     return;    // this 'return' is optional, not required
}
```

This 'void' represents, the welcome() function is returning no-value

The above **"welcome()"** function is just displaying a message on the screen, it is taking no argument and returning no-value. Therefore, the return-value-type '**void'** has given in this example. The last closing braces of a function works as "return" statement, therefore "return" is an optional statement at the end of function. As we know, the main() function generally returns no-value, that is why we always write main() function body as **"void main(){…}"**

Printing multiplication table(s) up to 10 terms

This function takes table number N as argument and prints the 10 terms of a table and returns nothing. Because this is not calculating any value to return it. The main() calling for tables 3, 6 and 12.

```
void printTable( int n)
{       int i;
        if(n<0)
        {       printf("error, table Number should not be –ve");
                return;     // this 'return'  statement throws the control back to main() fn. (this is not an optional statement)
        }
        for(i=1; i<11; i++)      // loop to print 10 terms in a table
        {    printf("\n %d * %d = %d", n, i, n*i);
        }
        return;        // this return statement is optional
}
void main()
{      printTable(3);              // fn call to print 3rd  table
       printTable(6);              // fn call to print 6th table
       printTable(-7);             // fn call to print -7th table, but shows error as -ve
}
```

The **printTable()** function is not calculating any value, it is just printing table on the screen. Since, it is not returning any value, the return type "void" specified here.

Printing multiplication tables in given limits like 3 to 19

```
void main()
{      printAllTables( 3 , 19 );
}
void printTable( int lower, int upper )
{       int n;
        for( n=lower; n<=upper; n++)
        {       for(i=1; i<11; i++)     // loop to print 10 terms in a table
                      printf("\n %d * %d = %d", n, i, n*i);
        }
        return;        // this return statement is optional before last closing braces of fn
}
```

Program prints 3-digit number in English words

The function **printDigit()**, prints the given digit in English words, for example, if input argument is 4 then it prints "four" as English word. This function is returning nothing because it is not doing any calculations, it is just printing given digit on the screen.

Now using this fn, write a main() fn, where scan 3-digit number like 247 and print output as **two-four-seven.**

```
void main()
{       int n=247;                      // or scan from keyboard
        printDigit( n/100 );            // passing 1st digit  2
        printDigit( n/10%10 );          // passing 2nd digit  4
        printDigit( n%10 );             // passing 3rd digit  7
}
```

```
void printDigit(int n)
{       switch(n)
        {       case  0:  printf("zero");  break;
                case  1:  printf("one");   break;
                --------
                case 9:   printf("nine");  break;
        }
}
```

A demo program, how multi-level calls are made

Example, how the control jumps several levels of calls and returning to same point of call.

```
void main()
{       printf("\n  before x() function");
        x();
        printf("\n  after x() function");
}

void x()
{       printf("\n  before y() function");
         y();
        printf("\n  after y() function");
}

void y()
{       printf("\n On behalf of C-Family Computers");
}
```

```
before x() function
before y() function
On behalf of C-Family Computers
after y() function
after x() function
```

- Here x() function is said to be **called-function** as well as **calling-function**.
- The main() is calling the x(), and x() is calling the y(). Therefore, x() is said to be **called-function** as well as **calling-function** based on context.
- The order in which the functions are written, they may not be the same order to be called.
- Functions cannot be nested like control structures, because their body is independent.
- A program can have any number of functions.
- In programming, to accomplish one task, one function may take assistance of other functions and that functions may take assistance of some other functions. In this way, software designed and developed with the collection of functions and that are provided by several people in the industry.

* Scope of variables

If any variable is declared within a block then it becomes local to that block and cannot be used or accessed outside of that block

```
void main()
{
        {
            int  k;
            k=100;
         }
        printf("%d ", k);        // error, undefined symbol  'k'
    }
}
```

Here compiler shows an error message called **"undefined symbol k"** at printf() statement**,** because the variable 'k' declared in inner block and we are trying to access or print outside. Thus 'k' cannot be used other than that block. In this way, variables are block-scope. Let us see one more example

example2:

```
void main()
{       int k=100;
        test();
        printf("the k value =%d", k);
}
void test()
{       k++;            // error, undefined symbol 'k'
}
```

Here compiler again shows same error message **"undefined symbol k".** The variable 'k' declared in main() function block and we are trying to access in test() function block. In this way, function block is also a block and we can't access one block variables in another block.

example3:

```
void main()
{       {
                int k;
                k=100;
        }
        {       int k;
                k=200;
        }
}
```

Here two copies of 'k' variables exist in this program, both memory addresses are different. The first 'k' belongs to first inner block and second 'K' belongs to second inner block.

example4:

```
void main()
{       int k=90;
        test(k);
}
void test( int k )
{       printf("k value is %d ", k);
}
```

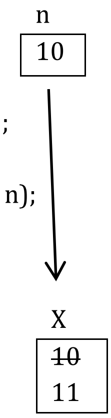
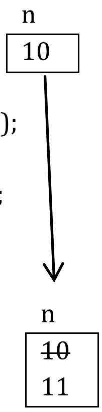
Here two copies of 'k' variables exist in this program, the first 'k' belongs to main() fn and second 'k' belongs to test() fn. Here the first 'k' served as argument whereas second 'k' served as parameter.

We will discuss this variable's scope and accessibility in storage classes chapter.

Demo program, how arguments VS parameters behave

If argument is a variable, then its value is passed and assigned to parameter, as we know, the parameter is a **copy** of argument, if any changes made in the **copy (parameter)** that doesn't affect the **original (**argument).
The following examples explain in two-ways

Example 1	**Example 2**

void main()
{ int n=10;
printf("\n before calling =d",);
test(n);
printf("\n after calling = %d ", n);
}
 	
void test(int x)
{
x++;
}
Output: before calling = 10
after calling = 10

 	
	void main()
	{ int n=10;
	printf("\n before calling = %d", n);
	test(n);
	printf("\n after calling = %d ", n);
	}
	
	void test(int n) // int n=5;
	{
	n++;
	}
	Output: before calling = 10
	after calling = 10
	

1) When test() calls, the argument **n** value 10 passes and assigns to parameter **x.** Now, **x** is said to be **copy** of **n.** (like above picture) If any changes made in **x,** it doesn't affect the **n.** if one wants to change the **n** through **x**, then pointer are required. We will see in next chapter.	Here names of argument & parameter are same; both variable names are **n,** but they are different copies, creates in separate memory area in the RAM. So, if any changes made in parameter **n,** it doesn't affect to argument **n.** **Both programs show same output.**

Swapping of two values

```
void swap( int , int);        //function proto-type
void main()
{     int x=5,y=7;
      printf("\n before swapping = %d, %d", x, y);
      swap(x, y);
      printf("\n after swapping = %d, %d", x, y);
}
void swap(int x, int y)
{     int temp;
      temp=x;
      x=y;
      y=temp;
}
```

```
before swapping  = 5, 7
  after swapping  = 5, 7
```

The swap() function cannot swap the x, y values of main() fn, instead, it swaps the x, y of its own function.
As we know, if parameter (copy) changes, it doesn't affect the argument (original) values.
To swap x, y values of main() using swap() function, pointers concept is used, we will see in the next chapter.

Passing array values one by one to function and printing on screen

```
void main()
{     int a[5]={ 55, 64, 67, 89, 32 };
      for(i=0; i<5; i++)
            display( a[i] );               //  calling 5 times using loop, and passing each value one after one.
}
void display( int  x )
{     printf("%d  ", x);
}
```

Here in this program, 5 elements are passed one by one in every call of function, this is just demo program.

Passing all elements of array at a time to a function

```
void main()
{     int a[3]={ 55, 64, 67 };
      display( a[0], a[1], a[2] );
}
void display( int  x, int y, int z )
{     printf("%d   %d   %d", x, y, z);
}
```

Here each element is passed separately as an argument to the function, so for 3 arguments, 3 parameters are required. Here we have taken x, y, z, for a[0], a[1], a[2].

Generally, this logic is not recommended when more elements found in array, for this problem, we have a solution in pointers concept.

More about functions

1. return and return-value-type
2. about main() function and returning value to O.S
3. types of data communications between functions (calling types)
4. function signature (proto-type)
5. local and global variables
6. rules and regulations applied to functions.

1) return & return-value-type

return: The job of 'return' statement is, terminating the execution of current function and transferring the control back to its calling-function with/without a value. The syntax is

```
syntax 1: return;              // returning the control without value
syntax 2: return(value);  // returning with value, here parenthesis are optional
```

Functions can have any number of return statements but executes only one, upon executing one statement, the control immediately transfers back to calling-function.

return-value-type: It is the data-type of a value which is being returned from a function, if return-type is not mentioned explicitly, then compiler takes default type, which is **int**.

The return-type explicitly tells what type of value the function is returning. It is useful for compiler to check syntax errors as well as to the programmer for documentation.

int test()
{ int f=56.78
return(f);
}
Here we are trying to return 56.78, but compiler returns only 56. Because, the return type is int.
	float test()
	{
	return(14);
	}
	Here, compiler returns 14 as 14.00
	Because the return-type is **'float'**

void test()
{
return 10;
}
If return-type is 'void' then we can't return a value.
It shows error.
	int test()
	{
	return; //observe no value
	}
	Here compiler shows an error/warning.
	Because return-type is int, but we are not returning any value;

test() // return-type is **int**
{
--- 	
--- 	
return 10;
}
Here the return-type is not mentioned, so compiler takes default type, which is **int**. In some compilers, the default type is **void**.
	int test(int k)
	{ if(k<5)
	return(100);
	else
	return;
	}
	Error, both 'return' statements are not same, first one is returning 100 but second one is not.
	The 'return' statements must be same type.

2) about main() function

One may get a doubt, if main() is a function, then whom is calling or from which point it is being called. Actually, it **is** called by O.S, which is in turn by our self by giving run command like F9 or typing exe file in OS shell, or in some other form. In some compilers, the main() must return a value, which goes to O.S and takes some actions like closing streams, files and other resources allocated for the program. It can be any of following ways

void main()
{
---- 	
---- 	
---- 	
}
Turbo C and other compilers allow this syntax.
	int main()
	{
	--- 	
	--- 	
	return 0; // here '0' indicates normal termination
	}
	return(0) : defines normal termination of program.
	return(errorCode) : defines some error found, here errorCode like return(1)/return(2),…
	This error codes defined in OS level
		int main()
		{
		--- 	
		if(condition)
		return 1; // equal to exit(1)
		----- 	
		----- 	
		return 0; //equal to exit(0);
		}

return(0) defines the normal termination of program and it closes all files, i/o streams and other resources before termination of program.

return(1),return(2), etc defines the abnormal termination of program, here OS records the error-code or error-type in log files for future reference. (Like history in web-browsers)

3) Types of data communications between functions

1) function with arguments and return value: most of the calculation functions take arguments and returns a value. For example, finding factorial value.

```
void main()
{   int k;
    k=fact(4);
    printf(“factorial is %d”, k);
}
```

```
int fact( int n )
{   int f=1;
    while(n>1) f=f*n--;
    return f;
}
```

2) function with arguments and no return value: some functions take arguments but returns nothing, eg printing multiplication table.

```
void main()
{
    printTable(7);
}
```

```
void printTable( int n )
{   for(i=1; i<11; i++)
        printf(“\n %d  *  %d =  %d”, n,i,n*i);
    return;
}
```

3) function with no arguments but returns a value: this task is less common in programming, takes no arguments but returns a value, like calling: rand() fn, getTime() fn, getDate() fn, getGPS() fn.

```
void main()
{       int m1,m2;
        m1=scanMarks();
        m2=scanMarks();
        if( m1>50 && m2>50)  printf(“passed”);
        else  printf(“failed”);
}
```

```
int  scanMarks()
{   int  m;
    printf(“enter marks:”);
    scanf(“%d”, &m);
    return  m;
}
```

4) function with no arguments and no return value: this is used for fixed jobs, the tasks like clearing screen, printing common messages.

```
void main()
{
    showAddress();
}
```

```
void showAddress()
{       printf(“C-Family Computers”);
        pritnf(“Vijayawada”);
        pritnf(“AP,India”);
        pritnf(“pin:520010”);
}
```

5) Function proto-type

This concept is very good for senior programmers when they want to create abstract data-types, but for beginners who learning C, it seems to be worst concept. It creates much confusion over function-call vs function-proto type. Good news is, in some modern compilers this concept is optional.

The English word proto-type means a model to an actual implementation, in C, it is used to introduce the function to the compiler to give an idea about the return-type, function-name, argument-type and count. Let us see, why it is required

```
void main()
{    ---
     k=test(20,30); // function-call
     ---
}
float test(float a, float b)    // function body
{     return(a+b);
}
```

In C, functions are compiled in an order in which they are written in the program. Here first main() fn, later test() fn will be compiled. While compiling main() function, at that time, compiler doesn't know about test() function(because its body appeared at bottom). When compiler faces the call statement "**k=test(20,30)",** it notices and assumes, the 'test()' is a function with arguments 20,30. Based on this call statement, compiler understands it as, the function is taking two int-type arguments by seeing (20,30) and returning int-type value. (because default return type is **int**)
But later, when compiling test() function-body, it will notices as, it is taking '**float-type'** arguments with return value float, therefore, this leads to data-type mismatch error between call and body.

To solve this problem, we have two options,

1. Declare the function proto-type before first call
2. Define the function body before first call. (called-function before calling-function)

```
float test(float , float);  // function proto-type
void main()
{      ---
       k=test();
       ---
}
float test(float a, float b)
{
       return(a+b);
}
```

proto-type declaration is nothing but, first function-header line terminated by semicolon. this is as shown above.

```
float  test(float a, float b)    // this body itself works as proto-type
{
       return a+b;
}

void main()
{       ---
        k=test();
        ---
}
```

**Note: if called-function body is provided before calling-function, then the body itself works as proto-type.
In this case, no special proto-type required**

6) Rules and regulations applied to functions

1. Argument and parameter must be compatible as per their count, types and order. If argument is **int-**type then the matching parameter should be int/long/float or higher types.

2. In C, all functions are global, so any function may call anywhere in the program. Sometimes self-calling is also possible, this is called recursion, we will see later.

3. Function can return only one value using **return** statement, to return more values **pointer** are required.

4. The default return type of function is **int.** But for some compilers it is **void.**

5. Some people think that, parameters are special kind of variables and cannot be used other than receiving arguments. Actually, these are like any other normal variable in the program.

6. Passing arguments and returning value makes the function independent, reusable, easy to write the logic and understand. Always divide the code as possible as independent blocks of functions.

7) Advantages of functions

Reduces repetition of code: the repeated code can be converted into a function and called as many times as we want in the programing.

Code reusability: Once, if function is made ready for use, it can be used several times in several programs where ever it is necessary. Thus, reuse of function makes the programming easier and speed. Generally, programmers develop applications from already existing code not from zero level. As we know pow(), sqrt(), printf(), scanf(),…etc are preexisting functions and they are using many times almost in every program.

Reduces complexity: when a big program is divided into several functions, it reduces complexity, improves readability, easy to modify and also debugging made easy.

Ease of debugging: debugging means finding syntax and logical errors. If any error occurs in a particular function then it doesn’t require checking in other functions. Therefore, it is easy to find errors.

Increases readability: By observing function call statements (function-name), one can understand what the function does, functions written by several people for several purposes. Generally, one doesn’t involve or try to know how other’s functions made it. For example, we are calling “sqrt()” without knowing how it works. So it is easier to understand the logic when program is made up with collection of functions. By observing only function call statements in the program one can easily understand the entire logic.

Easy to modify: if one function is modified, that does not affect the other functions as they are independent.

Portability: in computer science, C became a vital language, its usage extended in almost all fields in the real world. Therefore, today, the C software is available in most of the computer environments. If a function is developed on a particular operating system then it can be easily adapted to other operating systems without/negligible modifications. Such written functions are portable and generic.

Pointers

Pointers play an important role in C language; the excellent features of pointers have made ‘C’ a vital language in computers world. In some other languages like Pascal, Basic, etc where pointers are not programmers-friendly as they are in C. In C, pointers are simple to use and programs can be made efficiently and compactly. They give tremendous power to the programmer. A veteran programmer feels that, in the absence of pointers no application could be made as efficiently as he expected. However, pointers are dangerous too, for example a pointer variable containing an invalid address can cause the program to crash.

In programing, in all situations, we cannot access the data directly using variable name. The alternative is, accessing the data indirectly through its address; this is achieved through pointers. Pointers provide quick and dynamic access of arrays. Actually, array values implicitly managed through this indirect access technique only. Because, we can’t provide a name to each value in the array, where all values are accessed through single pointer. Pointers are preferred mostly in manipulation of arrays, strings, lists, tables, files, etc. They have well support of implementing dynamic data structures like array-lists, linked-lists, stacks, queues, sets, trees, graphs , etc.

Definition of pointer

Pointer is a special type of variable, which is used to store address of memory location, so that, such memory location can be accessed through pointer from any point in the program. In this way, pointers provide indirect memory accessing.

We know that, memory is a collection of bits, in which each eight-bits constitute one byte. Every byte has its own unique location number called **address**. This location number (address) is just a serial number of the byte sequence. For example, first byte address is 1, second byte address is 2, and so forth. To store this address, we need a special kind of variable called ‘pointer variable’.

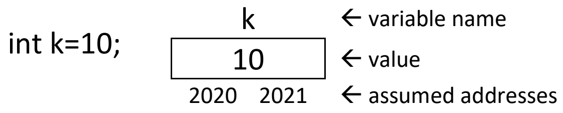
Address of memory location is just a serial number in the byte sequence, so, to store such address, a normal integer variable is enough. However, indirect accessing is difficult, because the compiler cannot determine whether the variable is holding an address or value. To work with pointers, we have two unary operators

- **Reference/address operator (&)**
- **De-reference/value operator (*)**

The ‘&’ and ‘*’ operators work together for referencing and de-referencing.

Reference operator (&)

This referencing operator is also called address operator, which gives the address of memory in which the variable is resided. Add symbol ‘&’ before variable name to get its address, for example



Syntax to get the address is:

&variable-name → gives address
&k → 2020

printf(“ value = %d , address = %u “ , k , &k) ; → value = 10 , address = 2020
// we know %d is used for signed int , %u is used for unsigned int (address is **unsigned int** value)

- The address of a variable may not be always a fixed location like 2020 as shown in above, and it is randomly given by the computer. Here we assumed k’s address is 2020.

- Here the expression '&k' is pronounced as **"address of k".** The space for 'k' at run time may allocate anywhere in the RAM, and to get such address, the reference operator(&) is used.
  The format string %u or %p is used to print the address. [%p→prints the address in hexadecimal format]

- We know that, the variable 'k' occupies two bytes in memory. Suppose, it has occupied memory locations 2020 & 2021, only the first byte address (2020) is considered to be the address of 'k' by the compiler. As compiler aware that 'k' occupies two bytes in memory, if the first byte address is 2020 then obviously the second byte address would be 2021. So, we always take ***first byte address as the address of a variable*** irrespective of variable size.
- Remember that, address is an **unsigned-int** type value.

De-Reference operator (*)

You may misunderstand this operator with multiplication operator, we use this operator for both aspects and you would not get any confusion between these two, both do different actions.

This de-referencing operator(*) is used to get the value at a given address, for example, *(2020) → 10; the value at address 2020 is 10. This is k's value.

Syntax to get 'value' at given address is: ***addressable_expression** // add symbol '*' before the address

For example: printf("\n the value of k = %d", *&k); // output = 10

*&k → *2020 →10

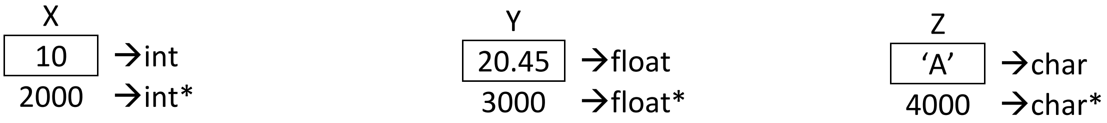
*&k → k → 10 // the operators "*** and &**" can be cancelled when they found in adjacent places

The reference operator (&) gives the address of 'k', whereas de-reference operator (*) gives the value in that address, so finally it is 'k' value. This operator is also called ***'value operator'*** as it gives the value at given address, but calling 'value operator' is informal.
Most of the time, this operator is called ***'indirection operator'*** as it changes the direction of control from one location to another location (pointer-variable to value-variable, from p to k) within the program's memory to access the required data. The action of 'de-reference' operator is just opposite to the action of 'reference' operator.

Data type of address

If variable type is "**int**" then its address type is "int*", similarly, if variable type is "float" then its address type is "float*". In this way, we can easily identify the address-type of a variable. Let us see,

int X=10; float Y=20.45; char Z='A';



Address types are derived types, by adding a symbol star (*) to the any existing data-type, it becomes as address-type. For example,

The data type **"int*"** derived from **"int"**
"int"** derived from **"int*"**
"int*"** derived from **"int**"**
……… in this way we can extend pointers up to any number of times (infinite times)

Similarly, we can define all other types. Notice, pointers are infinite types.

Declaration of pointer variable

The symbol star ('*') is a multipurpose operator, used in the following 3 cases

1. As a multiplication operator, to multiply two values
2. As a de-reference operator, to get value at a given address (as shown above)
3. **To declare variable as a pointer**

Now let us see how to declare a pointer variable

It is just like a normal variable declaration, but the symbol star (*) should be prefixed to the variable name. (The star symbol differentiates the normal and a pointer variable)

Syntax to declare pointer variable is: **data-type *pointer-name1, *pointer-name2, … ;**

```
For example: int *p;           // called integer pointer, it holds a integer variable address
             float *q;         // called float pointer, it holds a float variable address
```

Here **p** and **q** are pointer variables and they are capable of holding address of **int** & **float** variables respectively. Here **p** is also called a **pointer to int** and **q** is called a **pointer to float**. The integer pointer is used to access the integer value indirectly and a float pointer is used to access the float value indirectly.

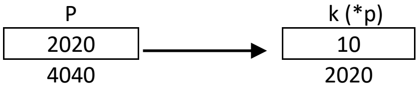
Note: Address is an integer type value, to store such address, pointers occupy 2bytes memory. Therefore, all type pointers occupy 2-bytes. Here p & q occupy 2bytes each. (we see later more clearly) Let us have some demo programs to get command over pointers and their applications.

Demo 1

```
void main()
{       int k=10;
        int *p;             // here '*' is used to declare 'p' as a pointer variable
        p=&k;
        printf("\n output1 =  %d",  *p);        //here '*'  is used to access  k's memory
        *p=20;                                  // here also  '*' is used to access  k's memory
        printf("\n output2 =  %d",  *p);
        printf("\n output3 =  %d",  k);
}
```

Output:

```
output1 = 10
output2 = 20
output3 = 20
```



Here the purpose of the instruction '**int *p'** is to declare **p** as a pointer variable. So here, the purpose of **'*'** is to specify the **p** as a pointer variable instead of normal variable.

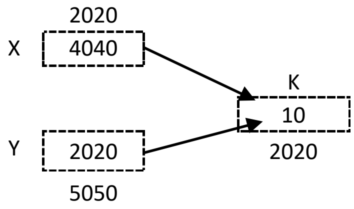
The expression **'*p'** in printf() statement accesses the **k**'s memory

When the **p** is assigned with **'&k'**, it can be called as "**the pointer p is pointing to k",** this is visualized as above figure.

Demo 2

This demo program tells how two pointers can points to same location.

```
void main()
{   int k=10, *x, *y;
    x=y=&k;              // both pointers  x, y is assigned with &k
    printf("\n output1 = %d   %d", *x, *y);
    *x=20;
    *y=30;
    k=40;
    printf("\n output2 = %d  %d  %d", *x, *y, k);
}
Output1: 10   10
Output2:  40  40   40
```



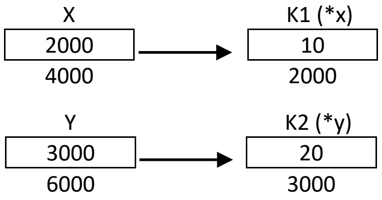
→Any number of pointers can points to one variable (one location), if the value is changed through one pointer then the other pointers also gains changed value.

→Here both pointers x,y pointing to the 'k'. Hence k's memory can be accessed in three ways using direct access k, using indirect access *x, and *y; here all expressions *x, *y, k → accesses same location(k).

Demo 3

This demo program swaps two values using pointers.

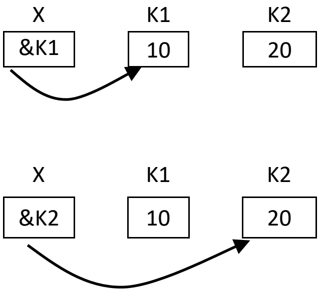
```
void main()
{       int k1=10, k2=20, *x, *y, t;
        x=&k1;
        y=&k2;
        t=*x;          // t=k1;
        *x=*y;         // k1=k2;
        *y=t;          // k2=t;
        printf("\n  output1 = %d  %d", *x, *y);
        printf(" \n output2=  %d  %d", k1, k2);
}
Output:
output1 = 20   10
output2 = 20   10
```



Demo 4

Expect the output of following program.

```
void main()
{      int k1=10 , k2=20 , *x;
       x=&k1;
       *x=30;
        x=&k2;
      *x=40;
      printf("\n %d    %d    %d    %d", *x, k1, k2);
}
```

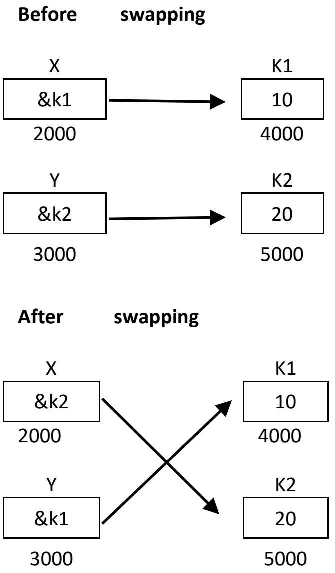


Here the pointer 'x' initially pointing to 'k1', later it is changed to 'k2';
so the expression "*x" accesses the memory which is currently pointing to.

Demo 5

Demo program for swapping two pointers and it is just like swapping two integer values.

```
void main()
{   int k1=10, k2=20, *x, *y, *t;
    x=&k1;
    y=&k2;
    t=x;
    x=y;
    y=t;
    printf("\n  output1 = %d  %d", *x, *y );
    printf(" \n output2=  %d  %d", k1, k2 );
}
Output:
    output1 = 20    10
    output2 = 10    20
```



Initially x,y are holding address of k1,k2; after swapping x & y, 'x' redirected to k2, 'y' to k1.

In this way, we can change the pointing address while running a program. Let us have one more example

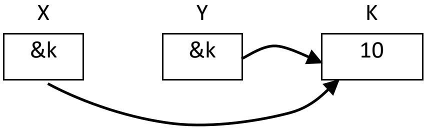
Demo 6

Expect the output of following program.

```
void main()
{   int k1=10, k2=20;
    x=&k1; y=&k2;
    *x=*x+*y;
    *y=*x+*y;
    printf(" \n output2=  %d  %d", k1, k2);
}
```

Demo 7

Expect the output of following program



```
void main()
{   int k=10, *x, *y;
    x=&k;
    y=x;
    *y=55;   *x=66;
    printf("\n %d    %d    %d    %d", *x, *y, k);
}
```

Demo 8

```
void main()
{    int k1=10, k2=20 ,   *x , *y;
     x=&k1;    y=&k2;
     printf("\n enter any two values ");
     scanf("%d%d", x, y);  //  passing x,y values to scanf(), which are nothing but  &k1, &k2;
                           //  this call can be taken as:  scanf("%d%d", &k1, &k2);
     printf("\n  %d  +  %d  =  %d",  k1, k2, *x+*y);
}
```

Types of pointers (address types)

Some people confused over meaning of pointer and an address, actually, it is something like a variable and its holding value. Similarly, pointer variable holds an address, therefore, we can interchangeably use these words while telling to others. Already we discussed before about address and its types, now we deal with more detail. Pointers are derived types as they can be derived from any existing data-type. By adding a symbol star (*) to the any existing data-type, it becomes as pointer-type.

```
 The data type:  "int*"     derives from  "int"
                 "int**"    derives from  "int*"
                 "int***"   derives from  "int**"   // in this way, we can go up to any levels (infinity)
```

Similarly all other types are defined and these are infinite types.

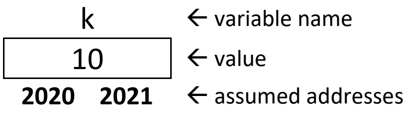
```
For example:  int   *p;      // the data-type of  'p'  is 'int*'
              float  *q;     // the data-type of  'q'  is 'float*'
              int    **x;    // the data-type of 'x' is  ' int**'
              int   **y;     // the data-type of 'y' is  ' int***'
```

Here 'p' and 'q' are pointers of type 'int*' and 'float*' types respectively.
If there is any data type like '**XYZ**' then '**XYZ***' will become a pointer type.

- In first version of C, the syntax to declare a pointer was
  int*   X,Y, Z;   // here X, Y, Z are '**int***' types
- In next version, it was modified as
  int *X, *Y, *Z;   // the '*****' shifted to before each variable name
- Now if you declare carelessly like: **int*   X** (or) **int   *   X;**
  then compiler takes as : **int   *X;**   // the symbol '*' shifts to before 'X'
- **int*   X, Y;**   here **X** is a pointer, but **Y** is a normal int. because it is: **int   *X, Y;**

How much memory occupied by a pointer?

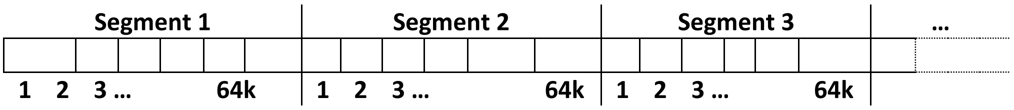
- As we know, address is like an integer number; hence, to store such address, 2 bytes is enough for the pointer variables. In windows/unix based C-compilers where pointers occupy 4-byte memory.
- In other point of view, only the first byte's address is considered to be the address of a variable, so to store such 2byte integer value, pointer requires 2byte.
- For example, here we take below **k's** address as 2020 (we ignore 2021)
- In other words, irrespective of type/size of a variable, only the first byte address assigns to the respective pointer, therefore, 2bytes enough for any type of pointer.



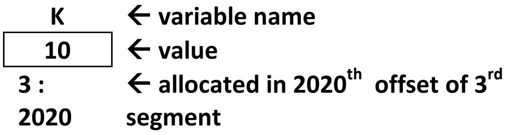
```
void main()
{      int  *p;   float *q;     char *x;    int k;    long int v;
       printf("sizeof(p) = %d", sizeof(p) );        // sizeof(p) = 2
       printf("sizeof(q) = %d", sizeof(q) );        // sizeof(q) = 2
       printf("sizeof(x)  = %d", sizeof(x) );       // sizeof(x) = 2
       printf("sizeof(&k) = %d", sizeof(&k) );      // sizeof(&k) = 2
       printf("sizeof(&v) = %d", sizeof(&v) );      // sizeof(&v) = 2
       printf("sizeof(v)    = %d", sizeof( v ) );   // sizeof(v) = 4
}
```

sizeof() : is an operator, which gives number of bytes occupied by a variable or expression or value.

Let us see, how main memory(RAM) managed in the computer: Memory is a large collection of bytes, in which it is divided into several logical blocks called segments, and each segment contains 65,535 bytes (64K) called offsets. The size of segment may vary from one to another operating system. This is as given below



For example: int k=10; // Let us say, the variable 'k' allocated in 2020th offset of 3rd segment, then k address would be 3:2020, but our programs print this address as a single number like 32020.



Conclusion: based on compiler and operating system, the pointer takes 2/4 byte.

Difference between one to another pointer

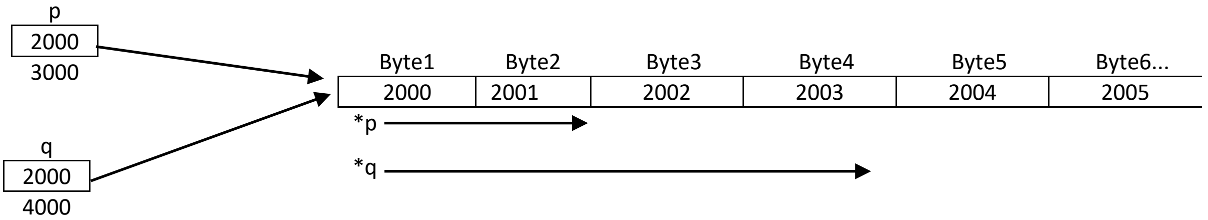
Even though all pointers occupy same size of 2 bytes to store address, they differ in two cases.

1. While accessing pointing value
2. While incrementing/decrementing the pointer (when accessing array elements through pointer)

Difference is, **int*** points to **int**,
float* points to **float** // Similarly, all pointers behave in this way

For example: int *p;
float *q;

Let us say, here p & q has assigned with same address 2000, then see how they access pointing memory



the expression *p access only the first 2byte of pointing memory as int-type, like above picture(2000+2001)
the expression *q access the first 4byte of pointing memory as float-type (2000 to 2003).
As **p** is int* pointer, it access only the first 2byte memory as 'int' type.
As **q** is float* pointer, it access only the first 4byte memory as 'float' type.
in this way, pointer access pointing memory. Let us see following example,

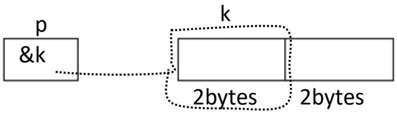
```
printf("sizeof(p) = %d", sizeof(p));       //  sizeof(p) = 2
printf("sizeof(*p) = %d", sizeof(*p));     //  sizeof(*p) = 2
printf("sizeof(q) = %d", sizeof(q));       //  sizeof(q) = 2
printf("sizeof(*q) = %d", sizeof(*q));     //  sizeof(*q) = 4
printf("\n before incrementing, the address is =%u  %u", p , q);   // say output 2000, 4000
p++; q++;    // p increments by 2,  q increments by 4
printf("\n after incrementing, the address is =%u  %u", p , q);    // output will be 2002, 4004
```

✓ While accessing array elements using pointer, the incrementing/decrementing is required, where '**int***' increments by 2 to access next element in the array**,** similarly '**float***' increments by 4. We will see this incrementing concept next topics.

**Precautions with pointers

Variable and its pointer must be matched, if not, then the pointer may access less or more bytes on pointing value, resulting, the program may behave strangely such as showing garbage output or crashing of program. program crash means hanging or closing a program abnormally in the middle of execution.

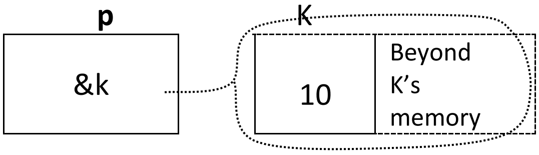
```
void main()
{   int *p;
    long int k=10;
    p=&k;
    printf("\n output = %ld", *p);
}
```



Did you observe? here **p** is int*, but **k** is long-int, even though **k** is **long-int** type, its address can be assigned to int* pointer of **p**. At compile time, it shows a warning and not an error. But ***p** access only the first 2byte of **k** out of 4byte, since it is an **int*** pointer. So, pointers blindly access pointing value as its base type.

Example 2:

```
void main()
{       long int *p;
        int k=10;
        p=&k;
        printf("\n output = %ld", *p);
}
```



output = unknown value (garbage)

Here the expression *__p__ access **4** bytes of target pointing memory, first 2bytes belonging to **k** and rest of 2 bytes unknown memory beyond **k**. it is said to be illegal accessing, and often called memory leak error, or pointer leak error. We don't run this type of programs otherwise may crashed (may closed in the middle of execution).

Example 3:

```
void main()
{   int *p;
    *p=20;       // stores value 20 in unknown location, because 'p' is not pointing to any variable.
    ----
}
```

Here '**p'** has not been initialized with any valid address, so by default it contains garbage (unknown) address. In the above example, the *p accesses the K's memory. In this program, '&K' has not been assigned to 'p'. So, the expression *p=20 puts 20 in unknown location in the RAM, this leads to illegal access, for security reasons, the operating system stops our program immediately. This is often called program crash.

Printing address of a variable

Sometimes, we need to check the address of a variable while debugging a program, whether the pointer is pointing properly or not? To print an address, use format string either %u or %p;

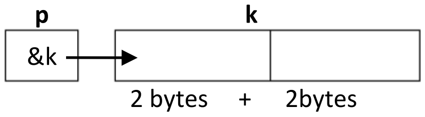
%u is unsigned-int format , %p is hexa-decimal format

```
void main()
{    int k=10, *p;
     p=&k;
     printf("\n  &k=%u  , p =  %u " ,  &k , p );      //  &k = 65524  ,  p =  65524
     printf("\n  &k=%p  , p =  %p " ,   &k , p  );     //  &k  =  fff4   , p =  fff4
}
```

Pointer conversion

In C programming, sometimes, we need to access one of type data from another type of pointer, where pointer conversion is needed. Here pointer needed to be type-casted as per pointing value. The following example explains how ‘long int’ value can be accessed through ‘int*’ pointer.

```
void main()
{   long int k=100;
    int *p;
    p=&k;
    printf("\n output 1 = %ld", *p);
    printf("\n output 2 = %ld ", *(long int*)p);
}
Output:
output 1  = garbage
output 2  = 100
```



Converting data-type **‘p’** from int* to long int*

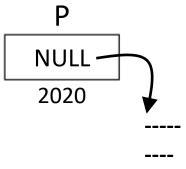
As we know, the expression ***p** access only first 2 bytes of **k** memory as **p** is int* pointer. For accessing total value of **k** then **p** should be type-casted to **long int***. This is shown in second printf() statement.

NULL pointer value

If pointer contained NULL value, we can say that, currently the pointer is not pointing any memory location. The symbol ‘NULL’ defined in ‘stdio.h’ file with value zero, at compile time, the symbol NULL replaces by zero wherever is used in the program (replaced in machine code file not in our source code file).

The usage of NULL is as follows

```
 #include<stdio.h>       // this file must be included
    void main()
    {   int  *p=NULL;      //  int *p=0;
        float *q;
           ----
        if(p==NULL)
        {   ---
        }
        q=NULL;
         ---
     }
```



Misunderstanding in pointer initialization

int k=10;

int *P=&k; → this is initialization of **P** with **&k.** So, here **&k** is assigned to **‘p’** but not to ***p**. But many people wrongly identify as, the **&k** is assigned to ***p** (because of its appearance). But actually, it is two actions like given below picture

int * P = &k;   first box is: **int *p;** and second box is **p = &k;**

Above initialization can be considered as:

```
int  *p;
p = &k;
```

void* pointer (generic pointer)

it is also called generic pointer, or pointer of no particular type, it can hold any type of address, but cannot be de-referenced without explicit type casting, because, the compiler cannot determine what type of address holding by the pointer. The following example explains how the ‘void*’ pointer can be used in the programming.

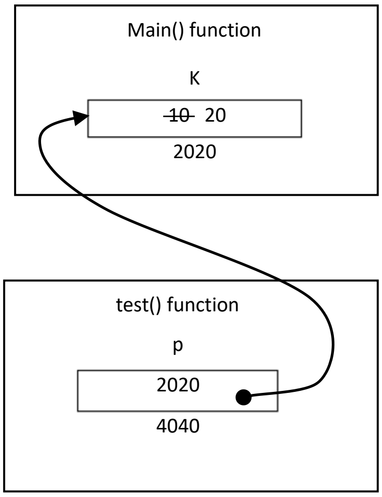
```
void main()
{     int k=10;    float f=20.45;    void *p;
      p=&k;
      printf( “\n value of k = %d”,   *p );              // error
      printf( “\n value of k = %d”,   *(int*)p );        // converting  p from “void*”   to   “int*”
      p=&f;
      printf( “\n value of f = %f”,   *p );              // error
      printf( “\n value of f = %f”,   *(float*)p );      // converting ‘q’ to “float*”
}
```

If p is ‘**int*’** pointer then it blindly accesses the pointing memory as **int,** and if p is ‘**float*’** then it blindly accesses the pointing memory as **float**. But, void* cannot determine how many bytes to be accessed at pointing location, because, it can be assigned with any type of address and compiler does not know what type of address currently it is holding, as a result, the expression *p gives an error at first printf(). So we need to use type-casting.

Passing address to function

Passing address to a function is same as passing value to a function, but the receiving parameter must be a pointer of same kind. Using **pointer parameter**, we can read/write/modify the argument’s value from called function. Passing address to a function mechanism is often called call-by-reference.

```
void test( int *p);
void main()
{   int k=10;
    printf("\n before call = %d" , k);
    test(&k);
    printf("\n after call = %d ", k);
}
void test( int *p)       // int *p=&k;
{
    *p=20;
}
before call = 10
  after call = 20
```

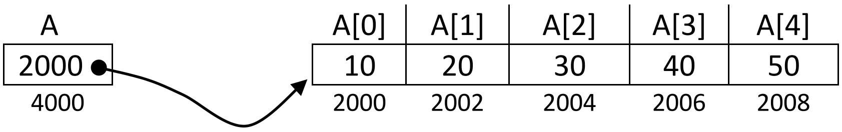


- When the test() function calls in main(), the ‘**&k’** passes and assigns to parameter **p**, thereby using **p**, we can read/write/modify the value of **k** from the test() function.
- Here the expression ***p=20** assigns 20 to **k**.
- this calling mechanism is known as **call-by-reference**

Arrays VS pointers

Arrays implicitly managed as pointers by the compiler, here all array expressions are converted into pointer expressions by the compiler. For example, the expression A[i] is converted into *(A+i) and we can also interchangeably use array expressions as pointer expression and vice-versa. Here array name works as constant pointer variable, which holds the address of first element. This is as given below

int A[5]={10, 20, 30, 40, 50};

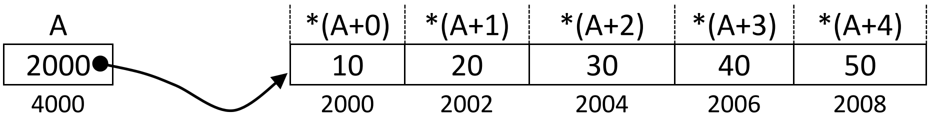


→ Array expressions can be converted and used as pointer expressions in C-Language.

→ The array expression A[i] can be taken in pointer expression as *(A+i),

→ So the expression A[i] is said to be **array-notation,** whereas *(A+i) is said to be **pointer-notation**

→ The above array elements can be accessed in pointer-style as



The expression A[0] → *(A+0) → *A

&A[0] → &*(A+0) → (A+0) → A → 2000 **(Let us assume 2000 is base address of array A[])**

so the expression ‘&A[0]’ can be taken as ‘A’

let us see, how array elements are accessed

A[0] → *(A+0) → *A → *2000 → 10

A[1] → *(A+1) → *(2000+1) → *(2002) →20 (integer-address increments by 2)

A[2] → *(A+2) → *(2000+2) → *(2004) →30

A[i] → *(A+i) → *(2000+i) ,

Note: compiler implicitly takes **A[i] as *(A+ i*sizeof(int)) → *(A + i*2)**

Making a pointer to an array

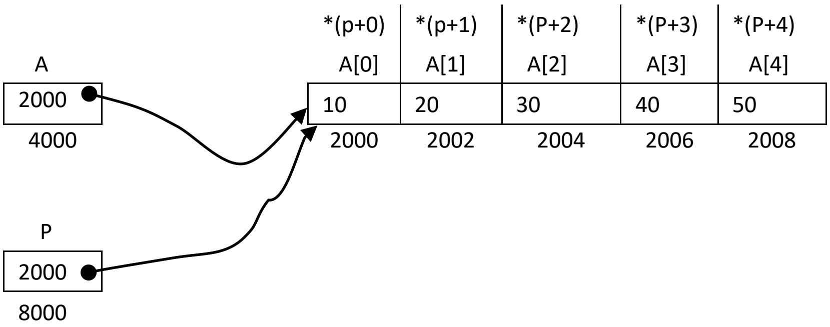
the C manufacturers also provided same facility to access such arrays using our own pointers.

Here a single pointer *cannot* point to all elements at once, instead, it can point to only one element at a time. By incrementing a pointer or by adding index value to pointer, the remaining elements are accessed.

Let us see one example,

```
int A[5] = { 10, 20, 30, 40, 50 };
int *p;
p=A;        // p=&A[0];    here &A[0] is an integer address, so our pointer should be int *p;
```

Let us see, how the pointer ‘p’ is pointing to an array



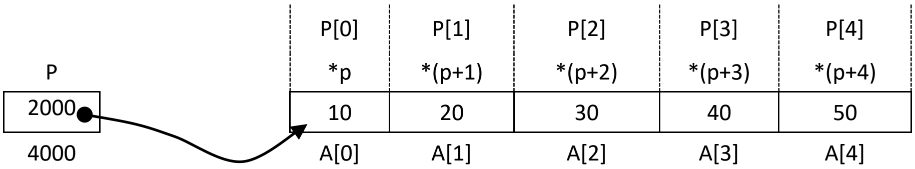
the above pointer expression *(p+0) , *(p+1) , *(p+2)… can be written as p[0] , p[1] , p[2] , p[3]
Since *(p+i) can be written in array style as p[i]

the expressions *(p+0) → p[0] → *p accesses the A[0] // now p[0] and A[0] access same location
*(p+0) → *(2000+0) → *2000 → 10

*(p+1) → p[1] accesses the A[1]
*(p+1) → *(2000+1) → *2002 → 20

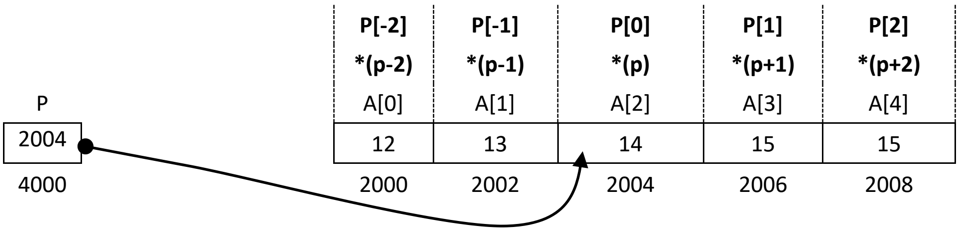
*(p+2) → p[2] accesses the A[2]
*(p+2) → *(2000+2) → *2004 → 30

The above figure can be imagined with the pointer "p" as



Example2: if the pointer 'p' is set to A[2] then the expressions are as follows

int A[5] = {12,13,14,15,16};
p = &A[2];

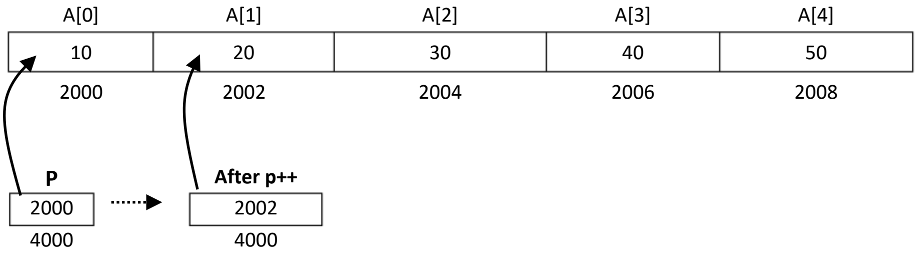


here p[0] is A[2] , p[1] is A[3] , p[2] is A[4] , p[-1] is A[1], p[-2] is A[0]

Demo program for printing array elements using pointer

```
void main()
{     int A[5] = {10, 20 , 30, 40, 50};
      int *p;
      p=A;      // p=&A[0];
      for(i=0; i<5; i++)
         printf("%d  ", *(p+i)  or  p[i]  );
      for(i=0; i<5; i++)
      {  printf("%d  ",  *p );
         p++;                         // increments by 2
      }
}
```

- The above two loops print the same output.
- In the 2^{nd} for loop, the **p** advances to next element after every **p++** operation, observe following figure



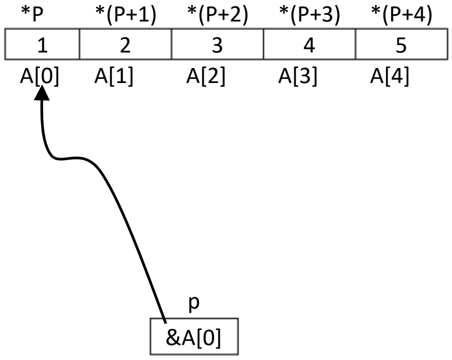
Example for printing address of each element in the array

```
        void main()
        {     int x[5] = {1,2,3,4,5};
              printf(“\n address of all locations are \n”);
              for(i=0; i<5; i++)
                  printf(“\n %u    %u “, &x[i]  or   x+ i  );
        }
Output: 2000    2002    2004    2006    2008   // address of each element in the array
```

Passing one dimensional array to function

We cannot pass entire array of elements to a function, that is, we cannot send all elements at once, there is no such provision in C, solution is, passing base address and accessing through pointer. This is simple & fast.

```
        void main()
        {     int A[5]={11, 22, 33, 44, 55};
              display(A);  // display(&A[0]);
        }
        void display( int  *p  (or)  int p[] )
        {     int i;
              for( i=0; i<5; i++)
                  printf(“%d  “,  p[i]  (or)   *(p+i)  );
              OR
              for( i=0; i<5; i++, p++)
                  printf(“%d “, *p  );
        }
```



In the above program, when display() function gets called, the argument ‘&A[0]’ will be passed and stored into pointer ‘**p’**. This is same as passing integer address to a function. The parameter ‘p’ can be declared in two ways like: int *p (or) int p[ ] (these two declarations are same)

The first declaration ‘int *p’ makes some confusion whether the pointer ‘p’ is pointing to single-integer or array-of-integers, so one cannot be determined immediately. Therefore, C providers gave 2nd declaration. But by seeing int p[], one might wrongly think, it is an empty-array, but actually it is also same as int *p. This declaration gives a clear idea that, ‘p’ is receiving an array-address instead of single integer address.

While developing big size applications, there exist many functions and all are messed up in the program. The called-function may not be found at bottom of calling-function, it may exist in some other file, that is why, they recommend this array-like-pointer declaration **“int p[]”.** It gives clear idea that, p is pointer to an array instead of single-value.

Let us have one more example, filling values to array at called-function.

```
    void main()
    {       int a[3];
            fill(a);
            printf(“\n after filling, the values are \n”);
            printf(‘%d  %d    %d“, a[0], a[1], a[2] );            //  output: 10  16   9
    }
    void fill( int p[]  or int *p )
    {       p[0]=10;     p[1]=16;     p[2]=9;
    }
```

Example2: following function takes array-address and number of elements, returns the sum of all values.

```
int findSum ( int p[] , int n )
{     int i, sum=0;
      for( i=0; i<n; i++)
      {     sum=sum+a[i];
      }
      return( sum);
}
void main()
{     int  a[5], n, i, k;
      printf("enter 5 values to array: ");
      for( i=0; i<5; i++)
          scanf("%d", &a[i]);
      k = findSum( a , 5 );
      printf("\n the sum of all values = %d",  k );
}
```

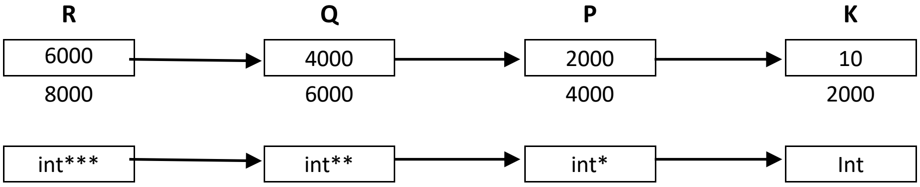
```
  input:  enter 5 values: 4   7   8   2    3
output:  sum is 24
```

> The ability to access calling-function's array elements at the called-function using its base-address provides convenience for moving multiple data items back and forth between functions shows the excellence of pointers in C.

Pointer to pointer (multiple indirections)

A pointer which points to another pointer variable is known as "pointer to pointer". For example, to store the address of **int** variable, the **int*** pointer is required, similarly, to store the address of **int*** variable, the **int**** pointer is required. In this way, we can extend up to desirable number of indirections. But most of the times we use single or double pointers. Pointer to pointer mainly used in multi-dimensional arrays.

```
void main()
{   int K=10;
    int *P;         // single indirection
    int  **Q;       // double indirection
    int  ***R;      // triple indirection
    P=&K;  Q=&p;  R=&Q;
    printf("\n output = %d  %d  %d ", K , *P , **Q , ***R );
}
```



The expression '*R' accesses the 'Q',   the '**R' accesses 'P',   the '***R' accesses the 'K' value.

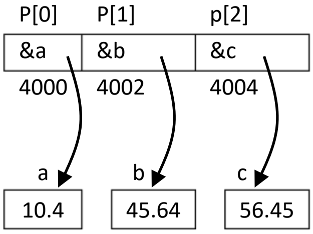
*R → Q	**R → *Q → P	***R → **Q → *P → K

Array of pointers

In computer science, array of pointers vastly used in the programming. I recommend you to practice more on this kind of applications. Array, which can hold the collection of addresses is known as array of pointers. For example, we can declare array of pointers as **int *p[3]** instead of **int *p1,*p2,*p3.**

Using this concept, we handle 2D array data such as matrices and tables with flexible and compact code. **"float *p[3].** Here p[0], p[1], p[2] are pointers of type 'float*' each. (array of 3 float pointers)

Demo program for array of pointers

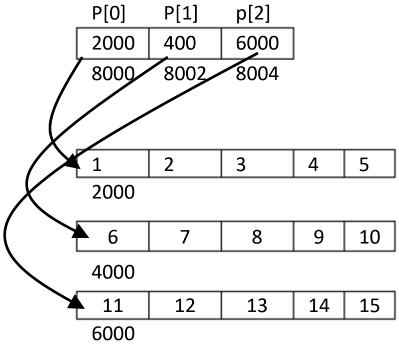


```
void main()
{      float a=10.4, b=45.64, c=56.45;
       float *p[3];
       p[0]=&a;   p[1]=&b;  p[2]=&c;
       printf(" %f  %f    %f", *p[0],  *p[1],  *p[2] );
}
```

```
output: 10.4    45.64    56.45
```

Let us see one more example

```
void main()
{    int *p[3],  a[5]={1,2,3,4,5}, b[5]={6,7,8,9,10}, c[5]={11,12,13,14,15};
     p[0]=a;   p[1]=b;   p[2]=c;   // p[0]=&a[0];  p[1]=&b[0];  p[2]=&c[0];
        for( i=0; i<3; i++)
        {   for(j=0; j<5; j++)
                printf("%d   ", *(p[i]+j)  or  p[i][j] );
            printf("\n");
        }
}
```



```
output:  1  2  3   4  5
     6  7  8  9  10
     11  12  13  14  15  16
```

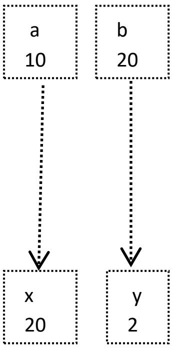
Call by value vs Call by reference

In **call by value** method, the values of arguments are passed and stored into parameters. That is, a copy of argument’s data is created in parameters at the called-function. Now, the arguments are said to be **original copy** and parameters are said to be **duplicate copy**. So, if any changes made in parameters (duplicate) that doesn’t affect to arguments(original). For example calling pow(), sqrt(), printf(), …etc. these functions follows the call by value method.

In **call by reference** method, the address of arguments is passed and stored into corresponding pointer-parameters. Now, using these parameters, we can read/write/modify the argument values.
For example, calling swap() function, incrementDate() function…etc. Let us see an example

Call by value does not work

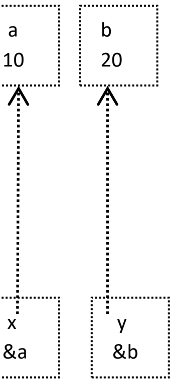
```
void main()
{
   int a=10, b=20;
   printf(“\n before swap = %d %d”, a, b);
   swap( a , b);
   printf(“\n after swap= %d  %d”, a, b);
}
void swap( int x , int y)    // x=a, y=b
{   int t;
   t=x;
   x=y;
   y=t;
}
Before swapping = 10    20
   after swapping = 10   20
```



The (a,b) values(10,20) are passed and stored in (x,y) so if any changes made in (x, y) that does not affects the values of (a, b)

Call by reference works

```
void main()
{
    int a=10, b=20;
    printf(“\n before swap = %d    %d”, a  ,b);
    swap( &a, &b);
    printf(“\n after swap= %d  %d”,  a , b  ) ;
}
void swap( int *x , int *y)     // x=&a, y=&b
{    int t;
    t=*x;
    *x=*y;
    *y=t;
}
Before swapping = 10   20
  after swapping = 20   10
```



The swap function pointers(x, y) are pointing to main function’s (a, b). So the expressions (*x, *y) indirectly swaps the values of (a, b) .

As we know, using ‘return’ statement, we can return only one value from a function, because most of the time, we return only single value, therefore the syntax was provided in that way. So, to return more values, pointers are used. Here we can return indirectly through pointers. The above swap() function indirectly returned the swapped values to main() function through pointers.

Let us see one example

Following function takes **radius** as argument and returns **area & perimeter** of a circle.

```
void   find( float radius, float *p, float *q)
{      *p= 3.14 * radius *radius;       // assigning indirectly to ‘area’ in main() fn.
       *q=2*3.14*radius;                // assigning indirectly to ‘circum’ in main() fn.
}
void main()
{      float radius, area, circum;
       scanf(“%f”, &radius);
       find( radius, &area, &circum);
       printf(“ area = %f”, area);
       printf(“\ncircumference = %f”, circum);
}
```

Here find() function is taking radius as input argument and calculating area & circum. These two calculated values indirectly assigning to main() function variables area & circum through p&q pointers.

Returning sum & product of 1+2+3…+N, 1*2*3…*N using single function.

This example explains how the function **find()** returns the sum & product of 1 to N to the main() function. This function returns sum and product indirectly through pointers.

Here '**find()**' fn takes arguments N along with **address of variables which receives the returning values**.

```
void main()
{       int N=12, sum, product;
        find(N, &sum, &product );
        printf("\n sum = %d , product = %d ", sum, product );
}
void find( int  N,  int  *x,  int  *y)
{       int i, s=0 ,p=1;          // 'i' for looping, 's' for sum,  'p' for product.
        for(i=1; i<=N; i++)
        {   s=s+i;      p=p*i;
        }
        // following assignments, returns s, p values to sum, product in main() fn through pointers x , y.
        *x=s;       // sum=s;
        *y=p;       //  product=p;
}
```

Following function returns big & small in array of 10 values.

The function findBigSmall(), which takes array base address & number of values in the array and to return the big & small of them. Fill the body of following fn.

```
void main()
{       int  a[10]={11, 34, 88, 49, 15, 66,  45, 23, 32, 17 };      // 10 values
        int big , small;
        findBigSmall( a, 10, &big , &small );
        printf("\n big = %d,  small = %d", big, small);
}
void findBigSmall( int a[] , int n , int  *pBig , int  *pSmall )
{       int s, b, i;
        s=b=a[0];
        for(i=0; i<n; i++)
        {   if( b<a[i] )  b=a[i];
            if( s>a[i] )  s=a[i];
        }
        *pBig=b;   *pSmall=s;   // returning two values to main() fn
}
```

Advantages of pointers

1. Call by reference
   - We can return more values from a function
   - Passing array to function is possible only through call-by-reference
   - If data is big like strings, structure, file then call by reference saves the time and space.
2. Dynamic memory allocation is possible only with the help of pointers.
3. Direct interaction with hardware and O/S is possible.
4. Pointers are the backbone to implement dynamic data structures.
5. Pointers make the code flexible, compact and fast execution
6. Through pointers it is easy to handle large collection of data.

More about pointers

1. Some common ways of pointer pointing to
   - Pointer to normal data variable
   - Pointer to pointer
   - Pointer to one-dimensional array
   - Pointer to two-dimensional array
   - Pointer to three-dimensional array…etc
   - Pointer to function
   - Pointer to structure
   - Array of pointers

2. All pointers occupy equal number of bytes to hold address

   int* pointer occupies 2 bytes
   int** pointer also occupies 2 bytes
   float* pointer occupies 2 bytes
   char* pointer also occupies 2 bytes

3. The type difference between one to another pointer helps to determine the data type of target location which the pointer pointing to

   int* pointer points to **int** memory
   float* pointer points to **float** memory
   Similarly, all pointers occupy in that way.

4. Incrementing/decrementing changes the pointer from one to next location in the array

   int* pointer increments by 2
   float* pointer increments by 4

5. Allowed and not allowed arithmetical operations on pointers
   - ➢ Adding or subtracting integral value to address is allowed like p+1, p-1, p+10 …etc
   - ➢ Multiplying or dividing integral value to address gives no meaning
   - ➢ Adding or multiplying or dividing two addresses has no meaning
   - ➢ Subtracting two addresses has a meaning (If two addresses belong to same array)
     For example, &x[6]-&x[2] givens the number of elements between locations.
   - ➢ Comparison of two addresses is allowed

   Note: If any operation has meaning on pointers then compiler allows such operation.

Pointer to function

Pointer not only points to a data, but also can points to a code. That is, a pointer can be made to points to a function. So that using pointer, we can make function calls also. In C, every function is loaded in separate memory location as it appears in C program. This location address is the entry point of a function, ie, it is the first instruction address of a function. Using this address, we can shift the control from caller-function to the called-function.

This approach seems to be unnecessary in general programming. Because in C, all functions are global, any function can be called from anywhere in the program. However, it is possible to pass addresses of different functions in different times to a function thereby making function-calls more flexible and abstract.

Syntax to declare a pointer to function

return-value-type (*pointer-variable-name) (list-of-parameters-type);

Let us see some examples

① int (*p)(int,int) → here the pointer 'p' can pointing to a function of type 'int fn-name(int,int)'
this is, the fn which takes two integer arguments and returns an integer value.

② void (*p)( ) → here the pointer 'p' can pointing to a function of type 'void fn-name( )'
this is, the fn which takes no arguments and returns no value.

③ float (*p) (float,float,float) → it can pointing to a function of type 'float fn-name(float, float, float);

To get the address of a function, use the function name without parenthesis and arguments that gives starting address of a function. The following program gives a demo.

```
void main()
{     void  (*p)();     // pointer to function, which takes no arguments and returns no value.
      p=test;           // assigning test() function address to 'p'
      p();              // calling test() function through 'p'
      (*p)();           // this is another style of calling the test() fn, this is old style.
}
void test()
{     printf("\n Hello World");
}
```

Example for calling a function based on even/odd input

```
void main()
{    int n , k;
     int (*p) (int , int);            //this is pointer to a function, which takes 2-int  arguments and returns int value

     printf("enter N value :");
     scanf("%d", &N);
     if( N%2==0) p=add;
     else  p=subtract;

     result = p(10,20 );              // calling the function based even or odd
     printf("output = %d", result);
}
int  add(int x, int y)  { return a+b; }
int  subtract(int x, int y)  { return a-b; }
```

Demo, passing one function address to another function

```
void y()
{
    printf("hello");
}
```

```
Void  x ( void (*p)()  )
{
    p();    // calling y() fn through 'p'
}
```

```
void main()
{
    x(y);   // passing y() fn ddress to x() fn
}
```

Sorting and Searching

The word sorting refers, arranging the data either in ascending or descending order. The collection of data may be an integers, real numbers, strings, etc. In computer science, the task of sorting is major process in everyday life. Therefore, there are several efficient techniques were invented, so programmer may relax from how to code and sort elements efficiently. U may get one doubt, what is the use of sorting? If data is not sorted order then searching takes much time. If data is in sorted order then searching will be easy, here the search operation takes less than $\log_2 N$ time. We have 3 elementary techniques available

1. Bubble sort.
2. Selection sort.
3. Insertion sort.

We have several advanced alternative sorting techniques available (not covered in this book)

Bubble sort

In bubble sort, the adjacent pair of elements are compared one with next element, and placed into sorted order by swapping disorder elements. ie, if a[i]>a[i+1] then swapping takes place. In this way bubble sort works. While comparing elements, each adjacent pair imagined as one bubble, therefore it was called as bubble sort. The process will be as follows

Step1: In first phase of loop, it compares a[i] with a[i+1], if any a[i]>a[i+1] then swapping takes place. In this way it compares and swaps from first element to last element. After this process, the biggest element will be moved to last position and it is said to be sorted.

Step2: In second phase of loop, it takes only first N-1 elements of array and repeats as said in step1, (here do not take last element, because it was already sorted in previous step1). In this step, the second big will be moved to N-1th position and it is said to be sorted. In this way, bubble sort works.

Suppose, the values are 90 10 70 30 20 and in every cycle of loop is as follows

phase1:	Before 1st iteration	90	10	70	30	20	
	Before 2nd iteration	10	90	70	30	20	
	Before 3rd iteration	10	70	90	30	20	
	Before 4th iteration	10	70	30	90	20	
	after 4th iteration	10	70	30	20	90	(90 sorted here)
phase2:	Before 1st iteration	10	70	30	20	90	
	Before 2nd iteration	10	70	30	20	90	
	Before 3rd iteration	10	30	70	20	90	
	after 3rd iteration	10	30	20	70	90	(70 sorted here)

In this way all elements gets sorted

```
int  bubbleSort( int a[] , int n )
{      int  i, j, temp;
       for(i=n-1; i>0; i--)
          for(j=0; j<i; j++)
              if( a[j] > a[j+1] )
              {      temp=a[j];
                     a[j]=a[j+1];
                     a[j+1]=temp;      // swapping elements
              }
 }
```

The main() function to test the bubble sort

```
        void main()
        {       int  a[]={12 , 45 , 22 , 34 , 15 , 67, 19}, i;     // 7 elements initialized
                printf("\n before sorting, the elements are :");
                for( i=0; i<7; i++)
                    printf("%d  ", a[i] );
                bubbleSort(a,7);
                printf("\n after sorting, the elements are :");
                for( i=0; i<7; i++)
                    printf("%d  ", a[i] );
        }
```

```
output: before sorting, the elements are: 12 , 45 , 22 , 34 , 15 , 67, 19
         after sorting, the elements are: 12 , 15 , 19 , 22 , 34 , 45 , 67
```

Analysis of bubble sort

to sort 1st big element, the inner loop take n-1 comparisons
to sort 2nd big element, the inner loop take n-2 comparisons
thus all iterations take

n-1 comparisons in 1st iteration.
n-2 comparisons in 2nd iteration.
---
2 comparisons in n-2th iteration.
1 comparison in n-1th iteration.

then total number of comparisons taken by if(a[j]>a[j+1]) is :

$n-1+n-2+.....+2+1 \rightarrow n*(n-1)/2 \rightarrow n^2/2$ (approximately)

but swapping takes extra time. If a[j]>a[j+1] is true then swapping occurs.

Then total no.of swaps are $n^2/2$.

But each swapping takes 3 operations, then the total no. of operations are $3*n^2/2$.
Consider all the time, the condition a[j]>a[j+1] may not be true, if we take average → $3*n^2/4$
Total no. of operation required for bubble sort = no. of comparisons + no. of swapping

$t(n) = n^2/2 + 3*n^2/4 \rightarrow 5*n^2/4$

$t(n) = n^2$ (approximately)

So, bubble sort takes n^2 operations (n^2 instructions need to be executed).

Bubble sort 2'nd method

Sometimes, the input array may be almost in sorted order except one or two elements. In this case, the above algorithm takes unnecessary comparison in all iterations.
For example, let us take 5 elements 12 18 39 55 40. Here by swapping last two elements, the total array gets sorted. So, after completion of 1st iteration of outer loop, the total array gets sorted and successive iterations would not be required. Using following algorithm, we can achieve this solution, here it stops the outer-loop when no swapping takes place in inner-loop. However, this algorithm is suitable when we know the input data is almost in sorted order.

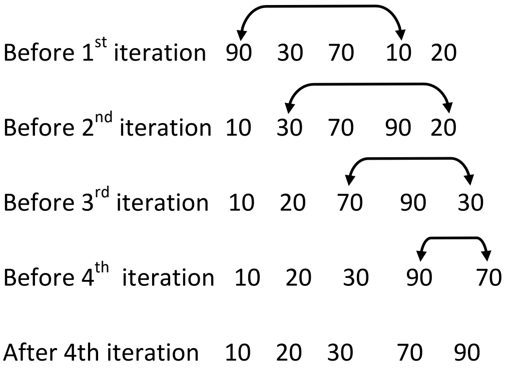
```
    void  BubbleSort2( int a[] , int n )
    {     int flag;
          for(i=n; i>0; i--)
          {       flag=0;
                  for(j=0; j<i; j++)
                  {       if( a[j] > a[j+1] )
```

```
                    {    temp=a[j];   a[j]=a[j+1]; a[j+1]=temp;   // swapping
                         flag=1;
                    }
               }
               if(flag==0) break;  //if no swapping occurred, then terminate the loop.
          }
     }
```

Selection sort

In selection sort, first it finds the smallest element in the array, and exchanges it with the first position element. Next it finds the second smallest element and exchanges it with the second position element. In this way selection sort works. Let us take 5 elements: 90, 30, 70, 10, 20

After every pass of outer loop, the elements is as follows



Before 1st iteration 90 30 70 10 20

Before 2nd iteration 10 30 70 90 20

Before 3rd iteration 10 20 70 90 30

Before 4th iteration 10 20 30 90 70

After 4th iteration 10 20 30 70 90

```
     void selectionSort( int a[ ] , int n )
     {  int i, j, smallPos;
        for(i=0; i<n; i++)
        {       smallPos=i;                 //  assume  a[i] is the ith small, so take its position to 'smallPos'
                for(j=i+1; j<n; j++)        //  loop to find small in remaining elements in the array
                {       if( a[smallPos] > a[j] )
                                smallPos=j;  //  if a[j] smaller than a[i] , then take its position to 'smallPos'
                }
                swap(&a[smallPos] , &a[i] );
        }
     }
```

Time complexity

- In the inner loop, the total no. of comparisons in all iteration takes as
  n-1 comparisons in 1st iteration
  n-2 comparisons in 2nd iteration
  ---
  2 comparisons in n-2'th iteration
  1 comparisons in n-1'th iteration
  Then total no. of comparisons taken by if(a[smallPos] > a[j]) are
  n-1+n-2+n-3+...+3+2+1→(n-1*n)/2→this is proportional to $n^2/2$.

- But extra operations are required for finding smallest position.
  That is, if a[pos]>a[j] condition is true then"smallPos=j" assignment takes place
  So total no. of assignments are $n^2/2$.

Consider all the time the condition a[smallPos]>a[j] may not be true ,then if we take average assignments then it is $n^2/4$.

- swapping elements in every outer-loop takes 'n' operations.
  But swapping takes three operations, so it takes 3*n.

- Now total no. of operations = total comparisons + total assignments + total swapping
  $= n^2/2 + n^2/4 + 3 * n$ ➔ this is proportional to n^2

Difference between Bubble sort and Selection sort

In this two sorting techniques, the selection sort is always better than bubble, because in selection sort, the no.of swapping are only N.

No.of operations in both sorting takes n^2. But selection sort is the chocie for sorting files with huge records and small keys(primary keys). For such application, the cost# moving the data dominates the cost of making comparisons, and no algorithm can sort a file with substantially less data movement than selection sort.

Let us say file contained 1000 records, and each record* contains 20 fields then frequent swapping in the inner loop of bubble sort takes lot of time. for example say, each field size is 45 bytes then 20*45 bytes of data has to exchange in each record.

Where as in Selection sort, we are finding smallest record position and finally exchanging it with first position. In this way the total no. of swapping is 'N', that is, swapping records occurs only N times. where as in bubble, the average swapping occurs $N^2/4$ times. So, selection sort is always better than to Bubble.

*Record : records is nothing but a collection of fields. for example customer record in a bank: Account holder name account no account type sex age address balance

#Cost: cost means time, here the time is calculated in terms of cost.

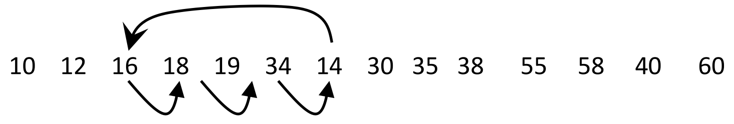
Insertion sort

Insertion sort inserts elements in proper places among those already in sorted order (those are sorted by previous iterations of loop). It shifts all previous bigger elements to right hand side, so that we get a gap to insert new element in its proper position.

How insertion sort works?

Let us take first 'k-1' elements in the array (sub array) and assume these are already in sorted order (sorted by previous iterations of loop). Now take k^th element, compare it with previous position elements and shift all previous bigger elements to right hand side so that we get a gap, where insert k^th element. These inserted positions may not be a final position, as they have to be moved again to make a room for further smaller elements, which may be encountered later.

Let us observe following elements:



For easy understating, let us take two elements 14 & 40 have to be sorted. Now take 14 and compare with previous elements one by one and move all bigger elements (34, 19, 19, 18, 16) to right side, so that we get a gap at 16 element place, where insert 14. In this way, insertion sort works. But this process starts from 2nd element.

Step by step procedure to sort elements (algorithm)

Step1: Pickup a[1], compare it with previous element a[0]. If a[0] > a[1], then shift a[0] to right side and insert a[1] in place of a[0].

Step2: Now first two elements themselves in sorted order, pickup a[2] and compare it with previous elements a[1] & a[0], shift bigger to right side and then insert a[2] in the gap.

Step3. Repeat this process for all remaining elements.

```
void  insertionSort ( int a[] , int n)
{       int temp;
        for(i=1; i<n; i++)
        {   temp=a[i];                  // now sorting a[i]
            for(j=i-1; j>=0; j--)       // loop to compare with previous elements
            {     if(a[j] > temp)       // if previous element is bigger then
                        a[j]=a[j+1];    //  moving a[j] to a[j+1] to right side
                  else
                        break;          // if previous element is small then stop the loop
            }
            a[j+1]=temp;                // insert element in rightful (gap) place
        }
}
```

Analysis of insertion sort

Best case: If input data already in sorted order then insertion sort takes only O(N) time only, because the inner loop each time terminates immediately by break. So the total No. of operations time is O(N).

Average case: If input data is randomly distributed then insertion sort takes approximately $N^2/2$.
(The inner loop terminates in the middle by break).
Total no. of comparisons are $n^2/2$, if we take average then it is $n^2/4$ (because loop may stopped in middle) so It uses about $N^2/4$ comparisons and $N^2/4$ moves on the average.
So total no.of operations is $N^2/4 + N^2/4 \cong N^2/2$.

Worst case: If input data is in reverse order (descending order) then insertion sort takes N^2 operations. Because the inner loop will not be terminated in the middle
Then no.of comparisons are $N^2/2$ + no. of moves of data are $N^2/2$. $T(N) = O(N^2)$.
Finally, the running time of insertion sort depends on the total number of inversions (reverse order of elements) in the file. But this takes less time than Bubble and Selection sort.

Comparison with other sorting techniques

Insertion sort is the best, when compared with selection and bubble sorts, because, in the average case insertion sort takes only $N^2/2$ units of time. It is difficult to prove theoretically, but practically insertion sort is the best. But bubble and selection takes n^2 units of time, if input is in any order.

In Other Cases: When the records are large size in file, then amount of time to move the data one place to another place takes a lot of time, in this case selection sort is better than insertion sort, because selection sort takes only N swaps.
Suppose in case of keys are strings then comparison of strings takes a lot of time than moving data one place to another place. In selection sort no. of comparison greater than insertion sort, so in this case, insertion sort is the best even though records are large size. (best reference book: algorithms by Robert Sedgwick)

Searching techniques

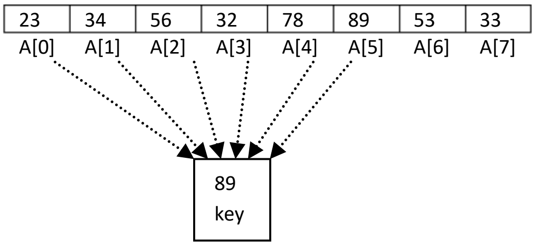
Searching is the process of finding required key element in the group of elements.

We haves several **techniques** in computer science, the two basic techniques are

1. Sequential or linear search
2. Binary or two-way search

Sequential or linear search

Sequential means one after one, in this search, the elements are compared one by one in the array, if the search-key is found, it returns the index position of such element, otherwise returns -1. **Here array elements may or may not be in sorted order**. For example, searching for 89 in the following array at 6th position



```
int linearSearch(  int a[],  int n, int key )
{       int i;
        for(i=0; i<n; i++)
        {    if( a[i]==key )
                  return i;
        }
        return -1;
}
void main()
{       int  a[200], n, i;
        printf(“enter number of input values to array :”);
        scanf(“%d”, &n);
        printf(“enter %d values to array:”,n);
        for( i=0; i<n; i++)
              scanf(“%d”, &a[i]);
        printf(“\n enter element to search :”);
        scanf(“%d”, &key);
        k=linearSearch(a, n, key);
        if(k ==-1)   printf(“\n not found”);
        else  printf(“\n element found at %d position”, k+1);
}
```

Binary search

The linear search is used when input file is small or elements are not in sorted order, it takes 0(n) time.

If elements are in sorted order, the binary search is the best choice, it takes <log$_2$(n) time.

Step1: In binary search, the array is divided into three parts at middle position, the left sub-array, middle element, right sub-array.

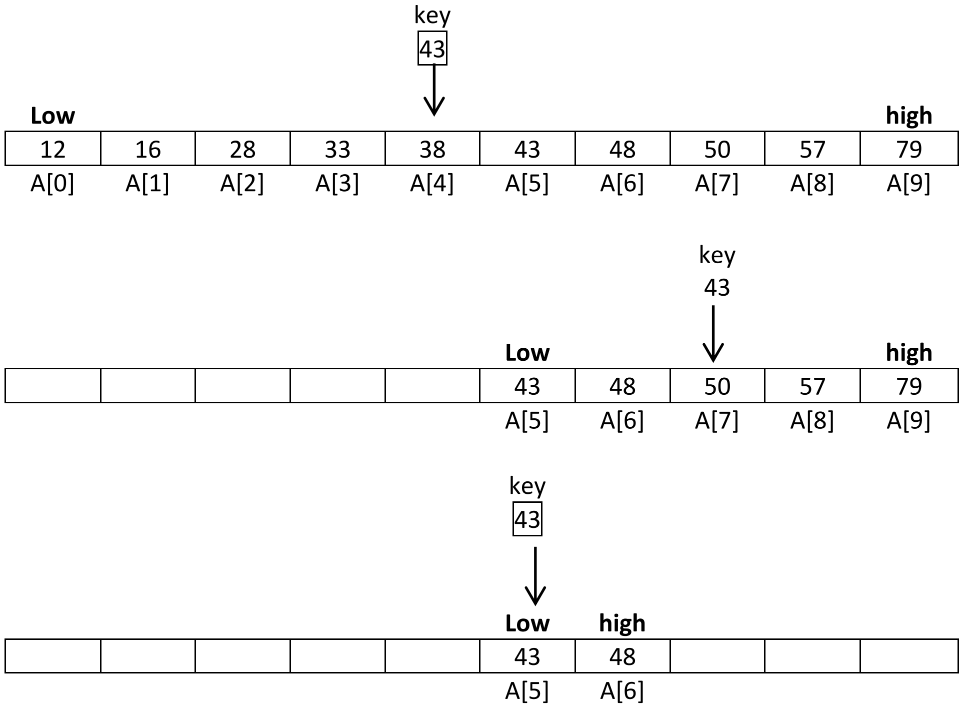
Step2: If middle-element==searchKey, it returns index of middle-element and stops the process.

Step3: If middle-element < key, it continues search in left-sub-array as said in step-1.

Step4: If middle-element > key, it continues search in right-sub-array as said in step-1.

step5: Repeat step1 to step4 until left/right sub-array partition becomes<1 element or element found.

Following picture demonstrates searching for 43.



key
43

Low									high
12	16	28	33	38	43	48	50	57	79
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

key
43

					Low				high
					43	48	50	57	79
					A[5]	A[6]	A[7]	A[8]	A[9]

key
43

					Low	high			
					43	48			
					A[5]	A[6]			

Time Complexity of binary-search,

Let array has N elements, and assume the N is almost equal to 2^K ($N \approx 2^k$)

At cycle1, the array size is N → (2^K)

At cycle2, the array size is N/2 → (2^{K-1}) → because at cycle2, we search in left/right sub-array.

At cycle3, the array size is N/4 → (2^{K-2})

After cycle K, the array size is 1 → 2^0

so it takes totally 'K' cycles, the total comparisons are: 1+1+1+ …k times (K comparisons)

But we have to express total comparison in known value 'N' (not in K), by applying log in both sizes

$N \approx 2^k$ → $k=Log_2(N)$, so finally binary search takes Log_2N comparisons in worst-case.

Source code for binary search

```
void main()
{      int  a[200], n, i;
       printf("enter number of  values to array :");
       scanf("%d", &n);
       printf("enter %d values to array:",n);
       for( i=0; i<n; i++)
             scanf("%d", &a[i]);
       bubbleSort( a , n );  // sort values before searching
       printf("\n enter element to search :");
       scanf("%d", &key);
       k=binarySearch(a, n, key);
       if(k ==-1)
             printf("\n element not found");
       else
             printf("\n element found at %d position", k+1 );
}
```

```
int binarySearch( int a[], int N, int key)
{      int mid, low, high;
       // the lower limit of array is 0, and higher limit  is N-1
       low=0; high=N-1;
       while( low<=high)  // repeat until partition size <= 1
       {      mid=(low+high)/2;
              if( key==a[mid] )
                     return mid;    // element found
              else  if( key < a[mid] )
                     high=mid-1;  // search in left sub array
              else
                     low=mid+1;  // search in right sub array
       }
       return -1;  // element not found
}
```

Scope, Accessibility & Storage Classes

Scope of variable: scope is the region, where the variable is available and accessible. First we discuss about local scope which we define using braces {}. Some space in the program area which is surrounded by pair of braces{} is said to be local-scope. In C, variables can be declared in any block in the program but they must be declared in first-line of block open; such variables are called local variables and can't be accessed other than that block. Let us see some examples.

Example1:
```
void main()
{
        {       int K;          // here memory space creates for 'K'
                k=100;
        }
        K=200;                  // here we get compile time error called  "undefined symbol K"
}
```

Here we get an error called **"un-defined symbol k"** at k=200, because, the **'k'** has declared inside the inner block, so it belongs to inner block and cannot be accessed/used outside. Now the scope of '**K'** is said to be this inner block and cannot be used outside. These variables are called local variables and scope belongs to same block in which they are declared. Let us have one more example,

Example2:
```
void main()
{       {       int k;          // memory space for 'k' creates here and scope belongs to this block only
                k=100;
        }

        {       int k;          // this 'k' is different from above 'k', here one more 'k' creates in the memory
                k=200;
        }
}
```

Here two copies of 'k' variables will be created, both memory addresses are different. The first 'k' belongs to first inner block and second 'K' belongs to second inner block.

Example3:
```
void main()
{       int  X;
        {   int  Y;
            Y=100;
            X=100;   // outer-block variable can be accessed in inner-block
        }
        X=200        // here we don't get any error, because, it is accessible
        Y=200;       // here we get an error, because Y belongs to inner block and can't be accessed outside.
}
```

Example4:
```
int  X=100;
void main()
{       int Y=200;
        show();
}
void show()
{       printf("%d  %d" , X, Y );          // error, undefined symbol 'Y'
}
```

* Here the variable ‘Y’ has declared inside main() fn block, so it **cannot be** accessed in show() fn block, as it is local to main() fn block. In this way one function’s local variables can’t be accessed in another function.

* Here the variable ‘X’ has declared at the beginning of program (outside all functions), now its scope is said to be **global** and it can be accessed anywhere in the program (in all functions). So, ‘X’ can be shared in all functions in the program.

Thus **Scope** is the region, where the variable is available and accessible. In C, scope is of three types: **local-scope, global-scope and file-scope.** The entire program’s area is said to be global scope, whereas a smaller region surrounded by a pair of braces {} is known as local scope. (file scope discussed later)

Local scope is the scope, surrounded by pair of braces{}, in C program, we can declare local variables anywhere in the program, but they must be declared in first line of block-open as shown in above examples. This block may belong to if-else, while-loop, for-loop, or function-block. This variable’s scope belongs to that block and cannot be accessed outside. These are often called local/private variables.

Global scope is said to be entire program’s area. If any variable declared at the beginning of program (outside all functions), then it is said to be global/public variable. Let see one more example,

```
int  X=100;                    //  X  is called global variable (now X is in global-scope)
void main()
{       int Y=200;             //  Y is local variable,  and belongs this main() fn block
        if( X < Y )
        {       int Z;         //  Z is local variable, and belongs to this if-block only
                Z=300;
        }
        X=200;
        show( 1000 );
}
void show( int A )           // A is local variable, belongs to this fn block, and also works as parameter
{       int B=100;           //  B is also local variable, and belongs to this show() fn block
        printf( “%d  %d  %d  %d”,  X, Y, Z, A,  B ) ;      //error,  here Y, Z are unknown to this fn block
}
```

Overriding of variables

If any two variables are declared in **inner** and **outer** block with the same name, then two copies will be created in the program, here the outer-block variable will be overridden by the inner-block variable when the control presents in the inner-block. That is, when the control is in inner-block, then the preference is given to inner-block variable.

```
void main()
{       int k=10;
        {       int k=20;
                {       int k=30;
                        printf(“ k = %d “, k);      // output:  k=30
                }
                printf(“ k = %d “, k);        // output:  k=20
        }
        printf(“ k = %d “, k);      // output:  k=10
}
```

Storage classes

We have 4 storage classes 1.auto, 2.static, 3.global, 4.register

auto & register are temporary variables, whereas static & global are permanent variables.

A variable that defined in the program possesses some characteristics such as data type and storage class. Data type tells the memory size and value type, whereas storage class tells the behavior of a variable, such as default initial value, scope, life-span and storage-place.
* Scope means region in which the variable is available and accessible. We have already discussed before.
* The default initial value is zero/garbage, temporary variables default value is garbage whereas permanent variables default value is zero.
* The life span specifies when the variable comes into existence and when it goes off, that is, when the variable's space gets created in the RAM(life starts) & when such space gets deleted from Ram(life ends).

```
auto int  x , y , z;        // here the storage class  of  x, y, z  is auto
static int   p , q;         // here the storage class of p, q is static
```

auto (automatic)

It is the default storage class for all local variables inside a block, that is, if any variable declared inside a block, then by default it is auto storage class. So, the keyword 'auto' is optional for auto variables. Of course, from the beginning of this book, we have been using 'auto' variables only.

example:
```
auto int  x, y, z;      // this is equal to  int x, y, z;
auto float p, q;        // this is equal to float p, q;
```

initial value: garbage is the default initial value. (if not initialized with any value)
scope: this variable can be accessed only within the block in which it is declared (already discussed above)
life: life is within a block as long as the control presence in that block. ie, when the control enters into a block, all of its auto variable's space is created (life starts) and when the control leaves the block, all such variable's space gets destroyed (life ends).

```
void main()
{       test();             // calling 3 times
        test();
        test();
}
void test()
{       auto int k=10;     //   memory space of k is created here, when the control comes here
        printf(" %d  ", k);
        k++;               //  no use of incrementing before dying  ( just for demo this is given )
}                          //  memory space of k will be destroyed here,  before returning to main() function
output: 10  10  10
```

The variable 'k' will be created & destroyed 3 times upon calling test() function for 3 times.
When the control enters into the test() function's body, the 'k' will be created & initialized with 10 and when the control leaves the function, the 'k' will be destroyed. Hence the output is 10 10 10.

Of course, it is meaningless to survive the 'k' in the memory after finishing the function task. Therefore, all auto variables will be destroyed before closing the function. But rarely some variables are needed to be survived, where the 'static' storage class is the choice.

static

This is also a local variable like auto, but space creates just before start of program (before main() fn starts) and survived till the end of program, used to access/share previous function call values for next subsequent calls. Unlike auto, the static is used to preserve the variable values even after closing the function. It will not be recreated & reinitialized in every call of function. Only once the variable is created and exists till the end of program. Scope belongs to local as auto, but initial value is zero.

```
void main()                         void test()
{       test();                     {       static int k=10;
        test();                             printf(" %d  ", k);
        test();                             k++;
}                                   }
                                    output: 10  11  12
```

The instruction **static int k=10** executes only once just before main() fn starts (this instruction shifts to before main() fn in machine code file), whereas **printf(" %d ", k); k++;** executes 3 times for 3 function calls.

Example2

```
void main()
{       for(i=0; i<5; i++)
        {       auto int  X=10;
                static int Y=10;
                printf("\n %d   %d", X , Y );          output:  10   10
                X++;  Y++;                                      10   11
        }                                                       10   12  … 5 times

}
```

The 'X' creates and destroys 5 times and every time it initializes with 10, so output is 10 10 10 10 10. Whereas 'Y' creates only once and initializes only once with 10, so output is 10 11 12 13 14.

register

Registers are small memory units available in the processor (ALU) itself, these are inbuilt memory units in the processor's ALU. Before doing any arithmetic operation, the input values are moved from RAM to registers and then given operation takes place. So, arithmetic operations are performed with the help of registers.

We know, only 4 or 5 registers are available in the processor and always 1 or 2 are busy. So other unused registers can be used for our variables, so that, they can have faster access. It saves the time to transfer variable values between RAM & registers. This is very much suitable for frequently used variables like loop variables. Only local integral types (char, int, long) are allowed as registers. The floating point types, arrays, pointers and static/global types are not allowed. Anyway, the keyword register is not a command, is just a request to the compiler, because, if registers are not available then space allocates in the RAM.
The other behavior like scope, life, and default-initial value is same as auto type.

```
Example:  register  int   a, b, c, d, e;
          register  float  x,  y,  z;         // error, float  are not allowed
```

In the above declaration, all (a, b, c, d, e) may not be allocated in the registers, because, we have limited number of registers. The x, y, z cannot be taken as register as they are float type. See following picture, here the register names are: AX, BX, CX, DX, SX, etc

RAM

Byte1	Byte2	Byte3	….

→

Registers in Processor		Circuits of Processor ALU
AX register		Addition circuit
BX register		Subtraction circuit
CX register		Multiplication circuit
DX register		……

global

Already we have discussed about these variable while discussing in global scope. Let us see more detailed. As we know, one function's local variable can't be accessed in another function, but sometimes, some variables need to be accessed throughout the program (in several functions) then we declare in global scope(outside of all blocks). Global scope is the region that doesn't belongs to any specific function or block, it belongs to entire program. Global variables preferably declared at the beginning of program or in a separate file. These variables life span is till the end program (same as static). The word **global** is not a keyword and hence it is not required while declaration of variable.

```
int k;          //now the 'k' is global and default value is zero
void main()
{       printf(" k = %d", k);
        test();
        printf(", k = %d", k);
}
void test()
{       k++;
}
```

Output: k=0, k=1, the variable 'k' is now global and can be accessed (shared) in both functions.

The static & global variable's life-scan & initial-values are same but scope is different.

The auto & register variable's scope and initial-value are same, but storage place may not be same.

storage place of variables (stack area and heap area)

The program's memory divide into **code & data** area, the code area contains instructions whereas data area contains variables, string-constants, etc. The data area is again classified into **stack & heap** area. The stack is the place where temporary variables are stored & managed, and in the heap area, the permanent variables are managed. (The stack & heap are predefined Data Structures in OS level).

the program's data area is as follows

Global variables	Static variables	String constants…	Auto variables	Register variables	Function return-address
Heap Area (permanent variable's area)			**Stack Area (temporary variable's area)**		

By default, stack variables contain garbage values, whereas heap variables contain zeros. For example:

```
auto int  x;      // 'x' contain garbage  value
static int y;     // 'y' contain  zero value
```

extern (proto-type)

If program is very big, obviously its code may distribute into several files and each file may contain several functions. Here, each file can be compiled independently and produced object file, later all such produced object files can be linked together. In this compilation process, we get some problems. If one file uses global variable but it is declared in another file then we get undefined-symbol errors. To avoid this type of errors, we need to specify the proto-type of global variable using keyword **extern.**

Program file1: “dis.c”

```
extern int num;   // not declaration, it is a proto-type

void display()
{   printf(“ experience =%d”, num);

}
```

Program file2: “main.c”

```
int num=14;    //declaration global of variable

void main()
{   display();

}
```

After compiling the 1st and 2nd program files separately, the object code for those files will be created with the file names “dis.obj” and “main.obj”. Suppose, if we are using Turbo C, then the command line usage of the compiler will be as follows

```
        tcc –IC:\tc\include –c  dis.c
        tcc –IC:\tc\include –c  main.c
```

Here “–I” specifies the include path of library functions, and “-c” is for only compilation.
Suppose if we are using C on UNIX system, then the method for using the compiler ‘cc’ is

```
        cc –c dis.c     //-c is for only compilation and not creating machine code file
        cc –c main.c
```

the above two commands will create the object files “dis.obj” and “main.obj” respectively. (in Unix it is dis.o and main.o) Now link these object files using the linker command. (Generally, the compiler itself can handle the linking job). The command line usage of the TCC will be as shown in the below example:

```
        TCC –Lc:\tc\lib   main.obj    dis.obj
```

here –L specifies the library files path. To do the same with UNIX’s CC

```
        CC   main.o  dis.o   final.exe
```

The above command will create the executable binary file with name “final.exe”

This extern keyword can also be used with function proto-types. This is especially when we want to access a function that is defined in other files or external pre-compiled modules. For example

Program file1: “dis.c”

```
void display()
{
       printf(“ experience =%d”, num);

}
```

Program file2: “main.c”

```
extern void display();  // declaration of function

void main()
{   display();

}
```

File scope

These variables are declared using **static** keyword in global scope, usually, file scope variables are declared at the beginning of file, so that, they can be accessed all functions within that file. File scope variables are local to one file and they cannot be accessed outside of file.

Program file1: “dis.c”

```
extern int exp; // not declaration , it is a proto-type

void display()
{    printf(“ experience =%d”, exp);

}
```

Program file2: “main.c”

```
extern void display();

static int exp=14; //global to this file only

void main()
{    display();

}
```

If we compile these two files separately, we don’t get any error, but at linking time, we get an error called “undefined symbol exp” in display() function. Because the variable “exp” is local to “main.c” file, since it is declared as static, therefore it is not accessible other than this file.

Preprocessor Directives

Before knowing about preprocessor directive, let us have one example, following program calculates the area & perimeter of circle.

```
void main()
{       float area, perimeter, radius;
        -----------
        area = 3.14 * radius * radius;
        perimeter = 2 * 3.14 * radius;
        -----------
}
```

For example, if we wrote a big application, where we used pie value 3.14 in many times, say more than 50 times, after some days, if this value needs to be changed 3.14 to 3.145 then it takes lot of effort to change in all places. If we missed in one or two places then it gives wrong output.

In such situations we think for a solution, it would be good, if it modified in one place, will affect to the total program. For that, preprocessor directive is the alternative, here we use **#define** statement like given below

```
#define PIE    3.14            // do not give semicolon at end
void main()
{       float area, perimeter, radius;
        -------------
        area = PIE *radius*radius;
        perimeter = 2*PIE*radius;
        -------------
}
```

At compile time, the symbol '**PIE'** replaces by 3.14 in every usage in the program, this replacement effects in executable file not in our source code file. If any changes needed in future then we simply modify the code at macro and recompile it, the compiler will generate a new 'exe' file by replacing old 'exe' file. This new exe file updated to new value. In this way, we automate repeated constants using #define directive.

It is mandatory to symbolize the repeating constants, so that they can be easily remember and modify later time. In this way wherever the pie value 3.14 is required, we can use **PIE** symbol.

What is the preprocessor?

The preprocessor, as its name implies, is a program, which performs some kind of text manipulations just before compilation begins. It provides some facilities like automating repeated constants, simplifies complex math equations, providing conditional compilation, combining two or more files…etc. Actually, this is not an **inbuilt** feature of c-compiler, this is an additional tool kit which helps the programmer.

Note: All preprocessor directives start with **#(hash)** symbol, and we can use these statements anywhere in the program. The directives are **#define , #include , #if , #else , #endif , #elif , #ifdef , #ifndef , #undef , etc**

Remember, the pre-processor directive generates a new temporary file and this file is given to the compiler as given below picture. It do not modify our source file, it creates new temporary file where it replaces PIE with 3.14. Thus compiler does not know about #define, #include statements.

file-name " sample.c "

```
#include<stdio.h>
#define PIE  3.14
void main()
{     float area, peri, radius;
       -------------
       area=PIE *radius*radius;
      peri=2*PIE*radius;
      -------------
}
```

After pre-processing

file-name " sample.temp "

```
void main()
{    float area, peri, radius;
      -------------
      area=3.14 *radius*radius;
     peri=2* 3.14 *radius;
     -------------
}
```

observe, here there is no #define or #include statements.
this file is given to the compiler

More about macro directive #define

#define is called macro directive, defines symbolic constants and equations.

The syntax is: **#define macro-name replacement-expression**

This macro directive defines a macro-name for a given replacement-expression. The replacement-expression is substituted in every occurrence of macro-name in the source file. The replacement-expression can be anything like constant, equation, string, etc. Let us have one more example.

```
#define   ONE          1
#define   TWO      ONE+ONE
#define   THREE    TWO+ONE
#define   show      printf
void main()
{    show("\n %d ",ONE);
     show("\n %d ", TWO);
     show(\n %d ",THREE);
}
```

⇒

After pre-processing, the program is

```
void main()
{    printf("\n %d ",1);
     printf("\n %d ",1+1);
     printf("\n %d ",1+1+1);
}
```

Observe that, here macro statements vanished by the preprocessor, so the compiler does not know about these directives

Note: It is good programming practice giving macro-names in full capital letters, so that, one can easily identify the macros in the programming. Actually, in software industry everybody follows this convention. I strongly recommend you to follow this rule for convenience.

Defining macro like function

The **#define** directive has another powerful feature, which can be used as simple function. Generally, it is used for simple mathematical equations, complex expressions and for simple functions. The macro function can have parameters and these are substituted at the time of pre-processing.

Syntax: **#define   macro-name(parameters)   replacement-expression-with-parameters**

```
#define  sum(x,y)    (3*x + 4*y )
void main()
{
     printf("\n %d", sum(7,8)  );
}
```

Code after pre-processing

```
void main()
{
      printf("\n %d", ( 3*7+4*8) );
}
```

Advantages of macro over function

- Macros does not make any function calls, instead it paste the replacement-expression in every use of macro. Hence we can avoid over-head of function call and return. As we know, the function call consumes extra space and time for jumping & returning the control, creating & destroying of local variable's space...etc. For simple functions, this time dominates function task. In this case, it is better to convert into macro, so that, we can save time & space. Simple function means having one to two lines of code.

- No need of specifying data types for the parameters, hence, we can use same macro with different data items. For example, the above macro sum(x, y) works with all data types like sum(4.5, 5.4) of float type.

Disadvantages with macros

- If macro replacement-expression is bigger, containing more lines of code and used many times then the size of machine code will be increased, because, in every usage of macro the replacement-expression will be pasted. In this case, normal function is the choice.
- There is no **syntax error** checking for macros, so it blindly pasts the replacement-expression.

```
#define   sum(x)    (x$x)
void main()
{   printf(" output = %d",  sum(5) );   // here macro pasts as  ( 5$5 )
}
```

Here by mistake, the user has given the **'$'** instead of **'+'**, the macro blinds pastes without checking any error. As a result, the compiler shows an error at printf() statement.

Let us see another example,

```
#define  sum(x)   x+x;          // this semicolon creates a problem
void main()
{       int k;
        k=sum(4)*sum(5);         // the compiler shows error at this line
        printf("%d  ", k);
}
```

Here the expression **k=sum(4)*sum(5);** replaces as **k=4+4;*5+5;;** // so this semicolon creates a problem.

#include directive

Generally, we break down a big application into several functions and we write in several files, later these files are combined using **#include** directive. This directive combines two or more files into a single file.
It is used to join the contents of one file into another file, we have 3 syntaxes

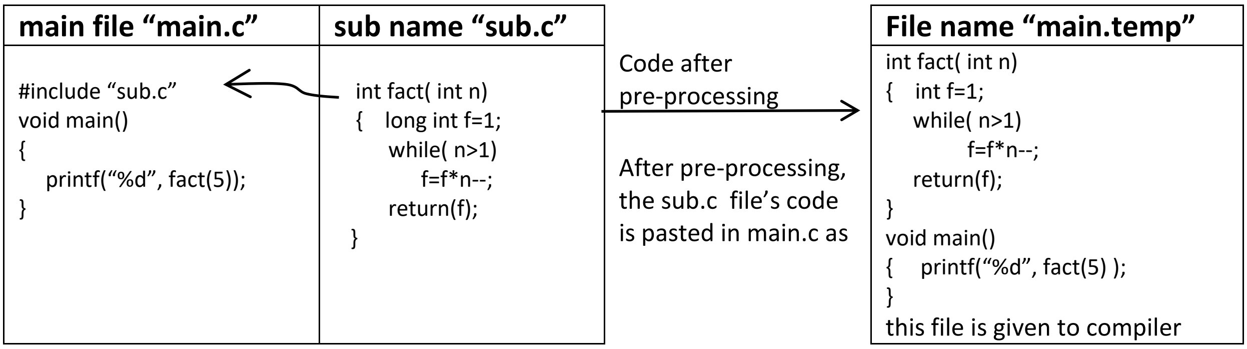
syntax1: #include "file name"

syntax2: #include "path + filename"

syntax3: #include <file name>

The first syntax is used, when the include file is in current working directory.
The second syntax is used, when the include file is in some other directory.
The third one, when the include file is in C-software directory.
For example:

here the file “main.c” contained main-program, “sub.c” file contained sub-program and the sub-file should be included as



main file “main.c”

```
#include "sub.c"
void main()
{
    printf("%d", fact(5));
}
```

sub name “sub.c”

```
int fact( int n)
{   long int f=1;
     while( n>1)
          f=f*n--;
     return(f);
}
```

Code after pre-processing

After pre-processing, the sub.c file’s code is pasted in main.c as

File name “main.temp”

```
int fact( int n)
{   int f=1;
    while( n>1)
           f=f*n--;
    return(f);
}
void main()
{    printf("%d", fact(5) );
}
```

this file is given to compiler

These two files will be clubbed into single and then given to the compiler (as shown above). In this way, we can attach as many files as we want in the program. The sub file (included file) can have many functions but required (called) functions only be attached to the executable file. In this manner, the C-header files contained proto type of all functions and we must include these files for avoiding errors.

Conditional Compilation

It is possible to compile selected portions of your program’s source code. This process is called conditional compilation and it is used widely by commercial software houses that provide and maintain many customized versions of one program.

As we know, these days, all commercial software designed keeping in mind all the aspects of related organizations in nature. For example, one bank’s software can be utilized in another bank with a few modifications. We cannot determine and finalize all the details of an application at the time of program development, for example, regarding of OS, hardware, services,…etc. Therefore, while developing software, the programmer must predict all possibilities that occurs later stages, thereby, it is needed to write the code for all possibilities so that we can add, remove, or modify the code according to the organizations required by the customer(client) before installation of software. In this way, the generic program can be changed according before delivering software with help of preprocessor. **Let us see one example**

```
#define COUNTRY   1       // 1-India, 2-US, 3-France, 4-England,…
#if COUNTRY==1
  char curr[]="Rupees";
#elif CUNTRY==2
  char curr[]="Dollor";
#else
  char curr[]="No currency defined";
#endif
void main()
{      printf("our currency = %s", curr);
}
```

By changing country code in first line, the corresponding currency name is applied to the ‘curr[]’

#ifdef, #ifndef and #undef

These is another method of conditional compilation, used to check whether a given symbol is already defined with **#define** or not. Based on this decision, we can add/subtract some code while compiling.

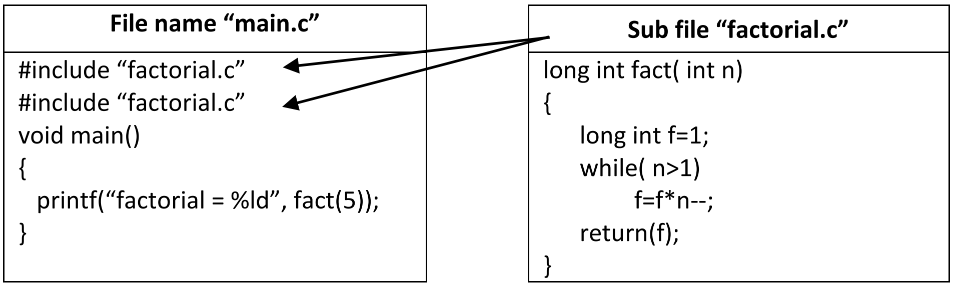
The Syntax is as follows

```
#ifdef   maco-name
    statement1
    statement2
     ...
#endif
```

example1	example2

#define SIZE 100
 	
#ifndef SIZE
    #define SIZE 200
#endif
 	
void main()
{
    printf("\n the size = %d", SIZE);
}
Output: the size = 100
	
	#ifndef SIZE
	    #define SIZE 200
	#endif
	
	void main()
	{
	    printf("\n the size = %d", SIZE);
	}
	
	Output: the size = 200

Example 3: Let us see one more practical example



File name "main.c"

```
#include "factorial.c"
#include "factorial.c"
void main()
{
  printf("factorial = %ld", fact(5));
}
```

Sub file "factorial.c"

```
long int fact( int n)
{
    long int f=1;
    while( n>1)
         f=f*n--;
    return(f);
}
```

In the above program, unfortunately the **"factorial.c"** file is included two times. At compile time, the preprocessor includes two copies of **"factorial.c"** file code in the program**.** As a result, we get an error at compile time. To avoid this type of errors, it is better to attach the macro check condition. This is as …

Sub file "factorial.c"

```
#ifndef  FACT
 #define FACT  1
    long int fact( int n)
    {
        long int f=1;
        while( n>1)
             f=f*n--;
        return(f);
    }
#endif
```

In the above code, at 1st time of inclusion of file, the symbol FACT is not yet defined, therefore the file "factorial.c" includes to the "main.c" file, and same time the FACT defines to 1.

In 2nd time inclusion, the FACT has already been defined; therefore it will be skipped by the #ifndef directive.

#undef

This directive removes a previously defined definition of the macro-name, for example

```
void main()
{
     #define   X  20
           printf(“%d “, X);
     #undef   X

     printf(“%d “, X);
}
```

after preprocessing

```
void main()
{
       printf(“%d “, 20);
       printf(“%d “, X);
}

Error :undefined symbol “X”
```

#error

Suppose programmer wants to show an error message if specific macro was not defined previously in the program. In those cases, we may use this #error to produce an error and to stop the compilation. This #error stops the compilation based on given condition. For example

```
      #ifdef  PIE
          #define    Area(r)      (PIE*r*r)
          #define    Perimeter(r)   ( 2*PIE*r)
     #else
          #error  “the symbol PIE not defined”
#endif
```

If we compile the above program, we get an error message, because the symbol “PIE” has not been defined in the program.

Two Dimensional arrays

Often there is a need to store and access two or three dimensional array data structures especially while dealing with matrices, tables, string, files... We can declare & access as many dimensions as we need in the programming.

Syntax: **data-type array-name [size1][size2][size3]….;**

```
For example:   int  x[3][5];           // two dimensional array
               int  x[3][4][5];        // three dimensional array
               int  x[3][4][5][6];     // four dimensional array
```

2D arrays: int x[3][5]; Here the value 3 represents **row-size**, 5 represents **column-size**.

Then the 2D array is as follows

	Columns				
Rows	x[0][0]	x[0][1]	x[0][2]	x[0][3]	x[0][4]
	x[1][0]	x[1][1]	x[1][2]	x[1][3]	x[1][4]
	x[2][0]	x[2][1]	x[2][2]	x[2][3]	x[2][4]

Scanning 2x4 matrix data and displaying on the screen

```
void main()
{       int  x[2][4], i, j;
        // scanning  data to 2D array
        for(i=0; i<2; i++)
        {       for(j=0;  j<4; j++)
                {       printf("enter value of x[%d][%d] :", i, j);
                        scanf("%d", &x[i][j] );
                }
        }
        // printing 2D array data
        for(i=0; i<2; i++)
        {       for(j=0;  j<4; j++)
                    printf("%d   ", x[i][j] );    // printing each element
                printf("\n");                     // inserting new line after each row
        }
}
```

Nested loops are required for scanning & printing 2D array data. Outer loop is for rows and inner loop is for columns. Here ‘i’ loop is to scan 2-rows, whereas ‘j’ is to scan 4-columns in each row.

If input numbers are 25, 61, 92, 53, 45, 67, 90, 55, then let us see how the program asks the user and stores into given array

```
enter value of x[0][0] : 25
enter value of x[0][1] : 61
enter value of x[0][2] : 92
enter value of x[0][3] : 53
enter value of x[2][0] : 45
enter value of x[2][1] : 67
enter value of x[2][2] : 90
enter value of x[2][3] : 55
```

25	61	92	53
45	67	90	55

Initialization of two dimensional arrays

We can initialize in two ways, first one is like 1D-array, second one is by enclosing each row with { }. Second method is clear and easy to understand. This method is recommended when the initializing values are more.

```
int   a[2][3]={ 51, 35, 67, 45, 23, 21 };
int   a[2][3]={   {51, 35, 67},  {45, 23, 21}  };
int   a[2][3]={ 51, 35, 67 };                  // only first row is initialized, and remaining is zero.
int   a[ ][3]={ 51, 35, 67, 45, 23, 21 };      // the row-size is 2, automatically calculated by compiler based on values.
int   a[ ][3]={ {51, 35, 67},  {45, 23, 21}, {44,55,67} };    // here the row-size is 3
```

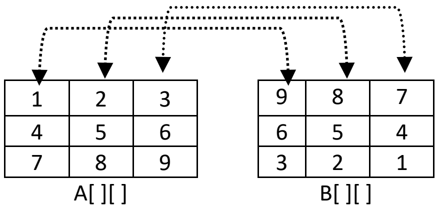
In the last two declarations, the row size automatically calculates by the compiler. Here, the row-size equal to (number-of-elements)/(column-size). The column size is mandatory while declaration of array.

```
inta[2][]={  {1,2,3}, {4,5,6}  }    // error, the column size must be required
```

Adding elements of 3x3 matrices and printing on the screen

This program adds two matrices of 3x3 size and prints the result. Consider the source matrices A,B already initialized with some assumed values.

```
void main()
{       int a[2][3]={     {1,2,3},  {4,5,6}, {7,8,9}  };
        int b[2][3]={     {9,8,7},  {6,5,4}, {3,2,1}  };
        int  i, j;
        for( i=0; i<3;  i++)
        {     for( j=0; j<3; j++)
                  c[i][j]=a[i][j]+b[i][j];
        }
         // displaying result of the matrix
        for( i=0; i<3;  i++)
        {     for( j=0; j<3; j++)
                  printf(" %d  ", c[i][j]);
              printf("\n");
        }
}
```



1	2	3
4	5	6
7	8	9

A[ ][ ]

9	8	7
6	5	4
3	2	1

B[ ][ ]

```
output matrix c[][] is:
        10   10   10
        10   10   10
        10   10   10
```

Adding two matrices of given input size

This demo program adds two matrices with a given input values instead of initialized values unlike above program. Here the size of matrices (order of matrix) and its data chosen by the user at run time. Therefore, these values should be accepted from keyboard.

```
void main()
{   int a[10][10], b[10][10], c[10][10] ,  r1, r2, c1, c2, i, j;
    printf("\n enter sizes of  1st matrix :");
    scanf("%d%d", &r1, &c1);
    printf("\n enter sizes of 2nd matrix :");
    scanf("%d%d", &r2, &c2);
    if( r1 != r2 ||  c1 != c2)
    {   printf("sizes not equal, cannot be added");
        exit(0);
    }
```

```
        printf("\n enter the elements of 1st matrix");
        for(i=0;  i<r1;  i++)
        {   for(j=0; j<c1;  j++)
            {    printf(" enter element of a[%d][%d] :", i, j);
                scanf("%d", &a[i][j]);
            }
        }
        printf("\n enter the elements of 2nd matrix");
        for(i=0; i<r2; i++)
        {   for(j=0;j<c2;j++)
            {    printf(" enter element of b[%d][%d] :", i, j);
                scanf("%d", &b[i][j]);
            }
        }
        // adding two matrices
        for( i=0; i<r1;  i++)
            for( j=0; j<c1; j++)
                c[i][j]=a[i][j]+b[i][j];
        printf("\n output matrix is \n:");
        for(i=0; i<r1; i++)
        {   for(j=0;j<c1;j++)
                printf(" %d  ",c[ i][j]);
            printf("\n");
        }
    }
```

```
Input:
        Enter the size of 1st matrix : 2  3
        Enter the size of 2nd matrix : 2 3

        enter the elements of 1st matrix
        enter element of a[0][0] : 7
        enter element of a[0][1] : 3
        enter element of a[0][2] : 2
        enter element of a[1][0] : 4
        enter element of a[1][1] : 6
        enter element of a[1][2] : 2

            enter the elements of 2nd matrix
        enter element of a[0][0] : 2
        enter element of a[0][1] : 4
        enter element of a[0][2] : 5
        enter element of a[1][0] : 6
        enter element of a[1][1] : 8
        enter element of a[1][2] : 1

output:  9     7     7
         10    14    3
```

Finding biggest of each row & column of 2D array data **(R X C size matrix)**

```
Input:
        23    55    66     7
        31    12    61    17
        5     14    98    47
        2     24    8     97
```

```
Output:
        row1 big = 66
        row2 big = 61
        row3 big = 98
        row4 big = 97
```

```
    void main()
    {   int a[10][20], r, c, i, j;
        printf("enter row and column size of input matrix :");
        scanf("%d%d", &r, &c);
        for(i=0; i<r; i++)
        {       for(j=0; j<c; j++)
                {   printf(" enter element of [%d][%d] :", i, j);
                    scanf("%d", &a[i][j]);
                }
        }
```

```
printf("\n each row output \n");
for(i=0; i<r; i++)
{       big=a[i][0];                // let us assume 1st column element is big
        for(j=1; j<c; j++)
        {   if( big < a[i][j] )     // comparing big with rest of elements in the row
                  big=a[i][j];
        }
        printf("\n biggest of  row %d  = %d ", i+1, big);
}
printf("\n each column output \n");
for(j=0; j<c; j++)
{       big=a[0][j];                   // let us assume 1st row element is big
        for(i=1; i<r; i++)
        {   if( big < a[i][j] )            // comparing big with rest of elements in the column
                 big=a[i][j];
        }
        printf("\n biggest of  column %d = %d ", j+1, big);
}
}
```

Transposing MxN matrix

Transposing is nothing but changing values from rows to columns and columns to rows

```
#define M  3
#define N  4
void  main()
{       int  A[M][N] = { {1, 2, 3, 4},
                         {5, 6, 7, 8},
                         {9, 10, 11, 12},
                       };
        int  B[M][N], i, j;
        // transposing elements from A[][] to B[][]
        for(i=0; i<M; i++)
              for(j=0; j<N; j++)
                   B[i][j] = A[j][i];
        printf("Result matrix is \n");
        for(i = 0; i < N; i++)
        {       for(j=0; j< M; j++)
                     printf("%d ", B[i][j] );
                printf("\n");
        }
}
```

```
ip:
     1    2     3     4
     5    6     7     8
     9   10    11    12

op:  1       5         9
     2       6        10
     3       7        11
     4       8        12
```

Multiplying two matrices and displaying the result.

Consider the source matrices A, B with the input sizes r1*c1 and r2*c2.

Before multiplying, the program checks the order of matrices; the size c1 must be equal to r2. If sizes are not equal then shows an error message "cannot be multiplied" and closes the program.

The output matrix size will be m1xn2.

```
void main()
{   int a[10][10], b[10][10], c[10][10];
    int r1, r2, c1, c2, i, j, sum;
    printf("\n enter size of input matrix :");
    scanf("%d%d", &r1, &c1);
    printf("\n enter size of second matrix :");
    scanf("%d%d", &r2, &c2);
    if( c1 != r2)
    {   printf("sizes not equal, cannot be multiplied");
        exit(0);
    }
    printf("\n enter elements of first matrix :");
    for(i=0; i<r1; i++)                       // loop to multiply 1st matrix of each row
    {       for(j=0; j<c2; j++)               // loop to multiply 2nd matrix of each column
            {       sum=0;
                    for(k=0;k<c1;k++)         // loop to multiply to get element of output matrix
                            sum = sum + a[i][k] * b[k][j];
                    c[i][j]=sum;
            }
    }
    printf("\n output matrix is \n:");
    for(i=0; i<r1; i++)
    {   for(j=0;j<c2;j++)
            printf(" %d  ",c[i][j] );
        printf("\n");
    }
}
```

Observe the multiplication stages carefully, it contained 3 loops, 1st loop is for maintaining the row index of 1st matrix, and 2nd loop is for column index of 2nd matrix. The 3rd loop is to keep the track of sum of products of row wise of 1st matrix with the columns wise of 2nd matrix. Finally this sum is assigned to the target matrix of relevant position.

Accepting marks details of 'N' students and printing result

This demo program accepts 'N' number of student details and prints the result.

Suppose if each student has 4 subjects then result will be displayed using following conditions.

If student obtained <40 in any subject, then print "failed", otherwise
If student obtained avg>=80 then print "passed in A-Grade"
If student obtained avg 80 to 60 then print "passed in B-Grade"
If student obtained avg<60 then print "passed in C-Grade"

```
input:  enter number of students: 3
        60   49   40   50
        10   20   30   40
        70   50   90   80
```

```
output:   idno   marks1   marks2   marks3    marks4      total    average      result
         =================================================================
             1         60         49         40             50          199       49         C-Grade
             2         10         20         30             40          100       25          Failed
             3         70         50         90             80          290       72          B-Grade
```

```
void main()
{   int  a[100][4];                          // let the maximum number of student are 100, and subjects are 4.
    int  i, j, total[100], avg[100], n;
    char result[100][20];
    printf("enter number of students :");
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {   printf("enter student %d marks of sub1, sub2, sub3, sub4  :");
        scanf("%d%d%d%d", &a[i][0],  &a[i][1],  &a[i][2],  &a[i][3]);
    }
    for(i=0; i<n; i++)                       // calculating total, avg and the result.
    {   for(j=0; j<4; j++)
            total[i]=a[i][j]+a[i][j]+a[i][j]+a[i][j]);
        avg[i]=total[i]/4;
        if( a[i][0]<40 ||  a[i][1]<40 || a[i][2]<40 || a[i][3]<40 )
            strcpy( result[i], "Failed");
        else  if ( avg[i] >= 80 )
            strcpy( result[i], "A-Grade");
        else  if ( avg[i] >= 60 )
            strcpy( result[i], "B-Grade");
        else
            strcpy( result[i], "C-Grade");
    }
    // printing result
    printf(" student idno   marks1  marks2  marks3 marks4   total avg result ");
    printf("\n==================================================");
    for(i=0; i<n; i++)
        printf("\n %d  %d  %d  %d  %d  %d  %s",  a[i][0],a[i][1],a[i][2],a[i][3] );
  }
```

2D arrays and pointers

Before knowing how to make a pointer to a multi-dimensional array, let us know, how a multi-dimensional array is stored in the memory? And how elements are accessed in different ways?
The two dimensional array data is also stored like single dimensional array, because the RAM is constituted as order collection of bytes. It is not like table, or matrix or in other form, it is like an array of continuous bytes. So the 2D and other dimensional arrays also occupy as 1D-array in the memory, this is as …

For example: int x[3][5] = { {1,2,3,4,5}, {6,7,8,9,10}, {11,12,13,14,15} };

1st row					2nd row					3rd row				
X[0][0]	X[0][1]	X[0][2]	X[0][3]	X[0][4]	X[1][0]	X[1][1]	X[1][2]	X[1][3]	X[1][4]	X[2][0]	X[2][1]	X[2][2]	X[2][3]	X[2][4]
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2000	2002	2004	2006	2008	2010	2012	2014	2016	2018	2020	2022	2024	2026	2028

In one dimensional arrays, the expression x[i] can be written in pointer form as *(x+i)
lly, In two dimensional arrays, the x[i][j] can be written in pointer form as *(x[i]+j) or *(*(x+i)+j)
But implicitly the expression x[i][j] is converted as ***(x+i*column-size+j).** This equation is called **row-major order** formula. This method used by compiler implicitly to access elements in the 2D-array, because, the row elements are expanded in column-wise. (Some other languages use **column-major** order formula)
Let us see how elements are accessed in 2D array

x[i][j] → *(x+i*column-size+j).
x[0][0] →*(x+0*5+0) → *(2000+0) → *2000 →1
x[0][1] →*(x+0*5+1) → *(2000+1) → *2002 → 2 // integer address increments by 2
x[1][0] → *(x+1*5+0) → *(2000+5) → *20010 → 6
x[1][1] →*(x+1*5+1) → *(2000+6) → *2012 → 7
x[2][0] → *(x+2*5+0)→ *(2000+10) → *2020 → 11
x[2][1] →*(x+2*5+1) → *(2000+11) → *2022 →12

Here the expression 'x' gives array base address.
That is, the expression x→&x[0][0]; // &x[0][0] →&*(x+0*5+0) → x→2000

&X[0][0] → X → gives array base-address → 2000

In one dimensional arrays, the expression x[i] → *(x+i);
In two dimensional arrays, the expression x[i] → (x+i*column-size) → gives ith row address.

x[0]→ (x+0*5) → x→2000 → gives 1st row address
x[1]→ (x+1*5) → x+5→2010 → gives 2nd row address
x[2]→ (x+2*5) → x+10 → 2020 → gives 3rd row address

Here special types of pointers are used to access the 2D array elements. For example
int (*p)[3]; → pointer to 2d-array where the column size is 3. The row size can be anything.
int (*p)[3][4]; → pointer to 3d-array where the row-column sizes are 3-4, table size can be anything.

Let us see one example

```
void main()
{   int a[2][3]={{1,2,3}, {4,5,6},{7,8,9} };
    int  (*p)[3];          // pointer 'p'  to 2d-array with the columns size 3.
    p=a;                   // p=&a[0][0];  making pointer to 2D array
    for(i=0; i<2; i++)
    {   for(j=0; j<3; j++)
            printf("%d", p[i][j]);      // p[i][j] → expands to → *(p+i*3+j)
        printf("\n");"
    }
}
```

Passing 2D array to function

it is just like passing 1D array to function, here array base address is passed and accessed through pointer.

```
void main()
{   int a[3][4]={ {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
    display(a);              // display( &a[0][0]);
}
void display( int (*p)[4]  or  int p[][4]  )
{   int  i, j ;
    for(i=0; i<3; i++)
    {   for(j=0; j<4; j++)
            printf("%d  ", p[i][j] );      // p[i][j] → *(p+i*4+j)
        printf("\n");"
    }
}
```

The second declaration for the parameter p[][4] is also a valid and this type of declaration is allowed only for the parameters while receiving 2D array. This is also implicitly a pointer of type "int (*p)[4]".

Passing row by row values of 2D array to a function

```
void main()
{       int a[3][4]={ {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
        for(i=0;  i<3; i++)
            display(a[i]);     // display (&a[i][0]) →  passing each row address like 1D array address
}
void display( int *p  or    int  p[] )    // here 'p' pointer to 1D array(row)
{       int i;
        for(i=0; i<4; i++)
            printf("%d  ", p[i] );    // p[i] → *(p+i)
        printf("\n");"
}
```

```
output: 1     2    3     4
        5    6    7     8
        9   10    11   12
```

Here each row address passed as 1D array to the function, for each call of **display()** function prints one row on the screen. It is called 3 times for 3 rows.

Adding two matrices using functions (custom sized input values)

```
void main()
{       int x[10][10], y[10][10], z[10][10] ,  r1, r2, c1, c2, r3, c3;
        scanMatrix( x, &r1, &c1 );
        scanMatrix( y, &r2, &c2 );
        k=addMatrix( z, &r3, &c3,  x, r1, c1, y, r2, c2);
        if(k==0)
            printf("addition is not possible, size not matched");
        else
            printMatrix( z, r3, c3);
}
void scanMatrix( int p[][10],  int *pr,  int *pc)
{       int  i,j;
        printf("\n enter the size of matrix :");
        scanf("%d%d",  pr, pc );
        for(i=0;  i<*pr;  i++)
        {       for(j=0; j<*pc;  j++)
                {       printf(" enter element of [%d][%d] :", i, j);
                        scanf("%d", &p[i][j]);
                }
        }
}
int addMatrix( int z[][10], int *pr, int *pc, int x[][10], int r1,int c1,  int y[][10], int r2, int c2)
{       int i,j;
        if( r1 != r2  ||  c1 != c2)
                return 0;     // addition failed
        // adding two matrices
        for( i=0; i<r1;  i++)
            for( j=0; j<c1; j++)
                z[i][j]=x[i][j]+y[i][j];
        *pr=r1;  *pc=c1;
         return 1;            // addition success
}
void printMatrix( int a[][10], int r, int c)
{       int i,j;
        printf("\n output matrix is \n:");
        for(i=0; i<r; i++)
        {       for(j=0;j<c;j++)
                    printf(" %d  ", a[i][j] );
                printf("\n");
        }
}
```

Character Handling

In programming, it is often need to handle single character data items such as employee sex(M|F), marks grade of a student(A|B|C), pass or fail (P|F), etc. These characters may be an alphabet, digit, or any other symbol, which is to be handled as a single character item in the program.

In computer's memory, the characters are stored in terms of their ASCII values but not in its picture form as visible on the screen. For example, the ASCII value of 'A' is 65, 'B' is 66. Remember that, these ASCII values are stored in its binary form as computer understands it. ASCII values are standard serial numbers given to each alphabet, digit, and all symbols in the keyboard. This standard is called **American Standard Code for Information Interchange** proposed by international computer association. Here the word "information interchange" means, interchanging the data from one device to another device or one computer to another in terms of ASCII values.

Hence every character internally represented in binary format and when it is to be displayed on the screen, it converts into symbolic picture form (picture in pixel form). Internally, special software continuously performs this graphical conversion job while displaying contents on the monitor. The hardware manufacturers provide this kind of software. The following table shows ASCII codes of some characters set

Characters	**ASCII values**
A to Z	65 to 90
a to z	97 to 122
0 to 9	48 to 57
' ' (blank space)	32
'\n' (enter key)	13

Character constant: If any single character enclosed within a pair of single quotes is known as char constant. Unlike integer or float constants, the char constants must be enclosed within a pair of single quotes.(not double quotes) . Remember, the char constant value is nothing but ASCII value. For example

- 19 , 34 , 10, -10 , 567 are valid integer constants.
- 19.78 , 10.00, -10.94, -45.98 are valid float/double constants.
- **'A', 'B', 'a' , '9', '+' …etc are valid character constants.**
- 'abc', '123', '+12', "A", "5" … are not valid character constants.

Character variable: The characters ASCII values lie in between 1 to 255, so to store such values, the C-developer Mr. Dennis Ritchie has given a suitable data type called 'char'. The data type 'char' takes 1byte memory, where we can store values of range 0 to 255.

```
char variable1, variable2,…variableN;         // this is the syntax to declare char variable
char  gender , marksGrade, accountType;       // here gender , marksGrade  are char type variables
```

Initialization of char variable:

```
char gender='M';          // male
char accountType='S';     // savings account
char color='Y';           // yellow
int   gender='M';         //  this is also valid but informal
```

char gender='M' or char gender=77; these both declarations are same, since 'M' ascii code is 77, but this 77 is difficult to remember/recognize as ascii code, so symbolic representation is always best. The last declaration **int gender='M'** is also valid but one byte is enough to store ascii, here 1byte gets unused in 2bytes of int memory.

Character library functions for I/O

scanf(), getchar(), getch(), getche()
printf(), putchar(), putch().

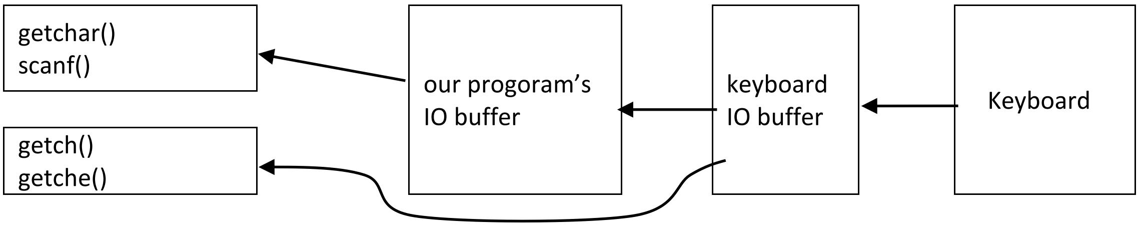
We know that, printf() & scanf() are generic i/o functions used to print & scan any type of data. Whereas other functions like getchar(), getch(), putchar(), putch() are specialized to handle only characters data. These are light weight functions to handle characters data.

getchar(): This is light-weight alternative to scanf() function, it accepts a single character from keyboard.

```
char ch;
ch=getchar();    // this is equal to scanf("%c", &ch );
```

getch(): the scanf() and getchar() does not support to read all keys like up-arrow, down-arrow, pg-up, pg-dn, home, end, F1, F2, etc. But the getch() supports to read all keys in the keyboard.
This reads values directly from keyboard buffer not from program's io buffer.

Buffer can be imagined as stock maintenance go-down building in a factory, first the manufactured items are kept for a while before shifting to destination. In computer, buffer is a small capacity memory, works as mediator between device & program, used to store input values for a while before reading by our program. Whatever you entered in the keyboard, the software of the keyboard inserts into its buffer, later our program reads such input values from the buffer. Thus buffer works as mediator between device & our program.



the getch() can accept any key code directly from keyboard buffer like above shown picture. It does not show input values on the screen. Even it doesn't wait for enter-key for input confirmation. That is, all other io functions waits until enter-key pressed by the user. When user presses the enter-key, then io functions starts scanning input. But getch(), getche() does not wait for enter-key for confirmation.
Note: The getch(), putch(), getche() are non-standard function, so it may not be available in all versions of C. It works only on DOS based C-Software like Turbo-C.

getche(): it is just as getch(), but it shows the input character on the screen.
putchar(): this is similar to printf() function for showing single character on the screen.
putch():it sends the input character directly to the monitor buffer.

Displaying all ASCII symbols

This program displays ASCII codes of A-Z, a-z, 0-9, etc.

```
vodi main()
{   char  ch;
    // printing all upper case alphabets
    for( ch='A'; ch<='Z'; ch++)          //  for( ch=65;  ch<=90;  ch++ )
        printf("\n %c = %d", ch, ch);

    // printing all lower case alphabets
    for( ch='a'; ch<='z'; k++)           //  for( ch=97;  ch<=122;  ch++ )
        printf("\n %c = %d", ch, ch);

    // printing all digits
```

```
        for(ch=’0’; ch<=’9’; ch++)            // for( ch=48; ch<=57;  ch++)
            printf(“\n %c = %d”, ch, ch);
    }
```

In the printf() statement, ‘%c’ is used to print character in picture format, whereas ‘%d’ is used to print its ASCII code. In the above program, the expression ‘ch++’ is a valid operation; even though the variable ‘ch’ is ‘char-type’. We can do all arithmetic and logical operations on characters just as integers.

Library functions for Character manipulations

isalpha(): This function checks whether the given character is alphabet or not? returns Boolean value.
If it is alphabet then returns 1, otherwise 0.

isdigit(): checks the given character is digit or not? Returns boolean value

islower(): checks the given character is lower case alphabet or not? Returns boolean value

isupper(): this is opposite above function.

tolower(): this coverts upper case alphabet to lower case and finally returns it.

toupper(): this is opposite above function.

Demo:

```
if( isupper( ‘A’ ) )  printf(“\n upper case alphabet”);    √
else   printf(“\n lower case alphabet”);   x

if( isupper( ‘a’ ) )  printf(“\n upper case alphabet”);    x
else   printf(“\n lower case alphabet”);   √

if( isdigit( ‘9’ ) )  printf(“\n digit”);        √
else   printf(“not a digit”);  x

ch=toupper( ‘a’ );
printf(“output is %d “, ch);  →  ‘A’

ch=tolower( ‘A’ );
printf(“output is %d “, ch);  →  ‘a’
```

Printing opposite case of a given character alphabet

This program accepts a single character alphabet from keyboard and prints opposite case.

```
  input:A↲        input: a          ASCII code of  A-Z  → 65 to 90 ,   a-z  → 97 to 122,   difference is  →  32
  output:a↲      output:A                              ‘A’+32  → ‘a’ ;  ‘B’+32  → ‘b’ ;  ‘a’-32  → ‘A’
```

below three programs do same job, but last one is good choice.

void main()
{   char ch;
    printf(“enter a alphabet :”);
    ch=getchar();
    if(ch>=’A’ &&ch<=’Z’)
            ch=ch+32;
    else if(ch>=’a’ &&ch<=’z’)
            ch=ch-32;
    printf(“opposite is: %c”, ch);
}
	void main()
	{ char ch;
	    printf(“enter a alphabet :”);
	    ch=getchar();
	    if(ch>=65 && ch<=90)
	            ch=ch+32;
	    else if(ch>=97 &&ch<=122)
	            ch=ch-32;
	    printf(“opposite is: %c”, ch);
	}
		void main()
		{ char ch;
		    printf(“enter a alphabet :”);
		    ch=getchar();
		    if(isupper(ch))
		         ch=tolower(ch);
		    else
		         ch=toupper(ch);
		    printf(“opposite is: %c”, ch);
		}

always we can’t remember the ascii difference, this code is not recommended	Always we can’t remember asccii codes, this is not recommended	This is the best choice to understand easily.

Printing ASCII/SCAN code of a given input symbol

Note: This program works only on DOS based compilers like Turbo c/c++

The keys in the keyboard divided into two types, normal keys, and special keys.

The normal keys are alphabets, digits, operators, punctuation symbols, etc.

The special keys are F1,F2, ….F12, up-arrow, down-arrow, pgup, pgdown, home, end, control and alt keys.

The normal keys produces only ascii-codes, whereas special keys produces two values as zero + scan code.

If first input code is non-zero then user entered a normal-key.

If first input code is zero then user entered a special-key, we have to scan again for 2nd code.

```
void main()
{       char ch1,ch2;
        printf(“enter any key in the keyboard :”);
        ch1=getch();
        if( ch1 != 0)          // user entered normal-key
                prnitf(“\n ascii code is = %d”, ch1);
        else
        {       ch2=getch();   reading 2nd code (scan-code)
                printf(“\n ascii code = %d, scan code = %d”, ch1, ch2);
        }
}
```

```
  input: A↵                          input: up-arrow↵
output: ascii code = 65            output: ascii code = 0, scan code = 72
```

Printing following patterns

```
ABCDEF
ABCDE
ABCD
ABC
AB
A
```

```
void main()
{   int i , j;
    for( i=0; i<6; i++)
    {   for( j=0; j<6-i; j++)
            printf(“%c”, ‘A’+j);
        printf(“\n”);
    }
}
```

```
A
AB
ABC
ABCD
ABCDE
ABCDEF
```

```
void main()
{   int i , j;
    for( i=0; i<6; i++)
    {   for( j=0; j<=i; j++)
            printf(“%c”, ‘A’+j);
        printf(“\n”);
    }
}
```

fflush(stdin) usage

```
void main()
{   char ch1,ch2;
     printf(“enter character1:”);
    ch1=getchar();   or scanf(“%c”, &ch1);
     printf(“enter character2:”);
    ch2=getchar();   or scanf(“%c”, &ch2);
    -----
}
```

if input is:

enter character1: **A ↵** (A + enter-key), here first character ‘A’ is scanned by ch1=getchar(); and enter-key scanned by ch2=getchar(); because, the enter-key is also a character. So does not give a change to read 2nd input character from KB. If we use fflush(stdin) before reading 2nd character then it clears keyboard buffer so that it allows to read a fresh new character. Let us see following program.

fflush() clears the keyboard buffer if any previous input character left like enter-key or space or F6.

stdin → keyboard buffer , stdout → monitor buffer, stdprn → printer buffer.

```
   void main()
   {   char ch1,ch2;
        printf("enter character1:");
       ch1=getchar();   or scanf("%c", &ch1);
        printf("enter character2:");
       ffush(stdin);  // clearing previous input values if any characters left like enter-key, space, etc.
       ch2=getchar();   or scanf("%c", &ch2);
       -----
   }
```

```
enter character1: A ↵
enter character2: B ↵
```

Strings

String is a collection of characters arranged sequentially one after the other in the memory, strings also said to be array of characters, used to represent names of persons, items, countries, cities and sometimes to represent messages. Generally, any non-computable data manipulates as strings. That is, on strings data, we don’t do any arithmetical operation such as addition, subtraction, multiplication, division. To handle strings, there is no direct data type in C, instead we have to access them as normal array of characters. However, C provides some basic facilities to handle strings easily, such as representation of string constants, initialization of string, automatic address generation of string constants and also provided several standard library functions to manipulate the strings.
Example strings are “India”, “China”, “C-Family”, “123”, “A123”, etc.

The strings are stored like array of characters in the form of ascii values, and these ascii values are again in the form binary values as computer understands it, for example, the string “INDIA” in the memory as

							→						
“India” in ASCII codes	‘I’	‘N’	‘D’	‘I’	‘A’	‘\0’	→	73	78	68	73	65	0

“India” in Binary codes	01001001	01001110	0100100	01001001	0100001	0

The string “INDIA” occupies 6 bytes in the memory (5bytes for INDIA + 1byte for null-character (‘\0’))
The value ‘\0’ is taken as null-char and its ascii value is Zero, used to represent as ‘end-of-string-marker’.
The compiler automatically appends null character at the end of string constants.

What is string constant?

Like integer float char constants, we can represent string constants too. It is the set of zero or more characters enclosed within a pair of **double quotes** is known as string constant.
Example for string constants: “India”, “C-family”, “A”, “”, “9440-030405”, “A123”, “1234”, etc.

Do not confuse 1234 with “1234”, the value 1234 is said to be integer constant, whereas “1234” is said to be string constant (with quotes). The integer constant 1234 is stored in two-byte(int) of memory in its binary format as: 00000100-11010010, whereas the string constant “1234” is stored in ascii form of every digit like shown below.

‘1’	‘2’	‘3’	‘4’	‘\0’	→	49	50	51	52	0	→	110001	110010	110011	110100	0

The null character and its importance

As we know, strings length differ from one to another, so there should be an indication to identify the end of string. For this reason, the null character (‘\0’) is inserted at the end of every string. Almost all C library functions are designed by considering this null character as a string terminator. String library functions automatically adds this null-character at the end of string, for example scanf(),gets(), strcpy(), strcat(), etc. Sometimes programmers need to be inserted explicitly null-char while handling char by char in the string. Some programmers use zero in place of null character, because, the null character value is zero.(‘\0’→0)

Note: The ascii values of all chars lie in b/w 1-255, so this exceptional value 0 took as null-character-value as a string terminator. But in programs, it is better to write ‘\0’ instead of 0(zero) to represent in symbolic way.
Note: Strings are nothing but character arrays, therefore, whatever operations applicable on arrays, they can also applicable on strings. For example, printing, scanning, passing string to function, accessing string through pointer, etc.

String as variable

we know that, variable is a name of space, used to store and access a value with variable name. Similarly, to store and access characters of a string, one is required “**char []**” array.

For example: char arr[20]; // It can hold a string with the maximum length 19 chars and 1 byte for null.

Initialization of strings: like normal arrays, we can initialize character arrays with the strings in different ways. This is as given below example.

```
char  a[9]=”C-Family”;        // here null character automatically appends at end by the compiler
char  a[]=”C-Family”;         // the size of array calculates by the compiler, which is equal to length of “c-family”+1
char  a[9]={ ‘C’, ’-‘, ’F’, ’a’, ’m’, ’i’, ’l’, ’y’, ‘\0’};     // we can initialize with char by char
char a[]={ ‘C’, ’-‘, ’F’, ’a’, ’m’, ’i’, ’l’, ’y’, ‘\0’};      // size not given, automatically calculated by compiler, it is 9
char a[]={67,45,70,97,109,105,108,121,0};       // we can also initialize with ASCCII values of “c-family”
```

The last declaration with ASCII values is somewhat confusion and not suggestible, it is just given for fun. The First two declarations are more convenient & easy to understand.

```
char a[3]=”C-Family”;     // compiler raises an error, because, the array size is not enough for the string
char a[20]=”hello”;       // remaining  14 locations gets unused(empty)
```

Accessing individual characters in a string is same as accessing elements in the array. The index of the character should be specified as array subscript.

For example: char a[10]=”C-Family”;

The expression ‘a[0]’ accesses the first character ‘c’,
‘a[1]’ accesses the second character ‘-’
‘a[2]’ accesses the second character ‘F’, …
in this way, we can access any element in the array

Console i/o functions on Strings

Input functions: scanf() , gets()
Output functions: printf() , puts()

gets() and puts() were provided simple syntax and specialized for strings data. The scanf() and printf() are generic i/o functions, they support for all types such as int, float, long, string. Here format string %s is used to read/write the string data.

① scanf(“%s”, array-address) : this reads a string char by char from keyboard until a new-line or a blank-space is found, whichever comes earlier. If input is “ jack and jill↲ ”, then scanf() accepts only “jack”.

② scanf(“%[^\n]”, array-address) : this reads a string char by char until new-line is found, that is, it accepts total string including spaces. If input is “jack and jill↲ ”, this scanf() accepts total string “jack and jill”.
The gets() and this scanf(“%[^\n]”, ..) works in similar way.
These input functions automatically append null character at the end of string. For example

Scanning string with “scanf()”	**Scanning string with “gets()”**

char a[20];
printf(“enter a string :”);
scanf(“%s”, a); //scanf(“%s”, &a[0]);
printf(“output = %s”, a);
	char a[20];
	printf(“enter a string :”);
	gets(a); or scanf(“%[^\n]”, a);
	printf(“output = %s”, a);

Input: Jack and Jill↲ //this scanf() treats as 3 strings Output: Jack	Input: Jack and Jill↲ // gets(), scanf() treats as one string Output: Jack and Jill

The puts() and printf() works in similar way, both prints the string char by char until null character.

Demo program for accepting a string and printing on the screen

ip: **Srihari is my name**↵

op: the string you entered is : **Srihari is my name**

```
void main()
{         char a[100];
          printf("enter string :");
          scanf("%[^\n]", a );          //  a → &a[0]
          printf("\n the string you entered is : %s", a);
}
```

the scanf() takes array base-address as argument and fills the scanned string into this array, at the end of string, scanf() automatically inserts null-char(not by compiler). Whereas printf() displays char-by-char until null-char is found.

Accepting a string char-by-char and printing char-by-char on the screen.

Note: this is same as above program, but scanning char-by-char until '\n' is found, later printing it.

ip: Srihari is my name↵

op: Srihari is my name

```
void main()
{         char  a[100] , ch;        int i=0;
          printf("enter string :");
          while(1)
          {         scanf("%c", &ch);    or ch=getchar();   // scanning char-by-char
                    if(ch=='\n') break;                    // if enter-key found then stop scanning
                    a[ i++ ]=ch;                           // inserting scanned char into array
          }
          a[i]='\0';      // inserting  null-char at end of string. It is our responsibility
          printf("\n output string is :");
          for(i=0;  a[i]!='\0';  i++)
                    printf("%c", a[i] );    or   putchar( a[i] );
}
```

Counting number of lower & upper case alphabets in a string.

This program accepts a string and counts the number of lower & upper case alphabets, numeric digits, spaces and other characters.

```
void main()
{   char  a[100];   int i, lower, upper, digit, space, other;
    printf("\n enter a string :");  gets(a);
    lower=upper=digit=other=0;
    for(i=0; a[i]!='\0'; i++)
    {         ch=a[i];
              if( ch>='A' &&ch<='Z') upper++;              //  if( isupper(ch)) upper++;
              else    if( ch>='a' && ch<='z') lower++;     // if( islower(ch)) lower++;
              else   if( ch>='0' && ch<='9') digit++;      // if( isdigit(ch)) digit++;
              else   other++;
    }
    printf("\n count of upper=%d , lower=%d , digits=%d , other=%d", upper , lower, digit , other);
}
```

Converting upper case alphabets to lower case and vice-versa in a string

This demo program accepts a string and converts lower case alphabets to upper case and vice-versa.

Input: This Chair Has 4 Legs

Output: tHIS cHAIR hAS 4 lEGS

We know that, uppercase ASCII ranges 65 to 90 and lowercase ranges 97 to 122. The difference between these two cases is 32. Therefore, by adding or subtracting 32, we can get corresponding opposite case.

```
void main()
{       char a[100];    int i;
        printf("\n enter string :");   gets(a);
        for(i=0; a[i]!='\0'; i++)
        {       if( a[i] >= 'a'  &&  a[i] <= 'z' )         //    if( islower( a[i] )  a[i]=toupper(a[i]) ;
                        a[i]=a[i]-32;
                else if( a[i] >= 'A'  &&  a[i] <= 'Z' )    //    if( isupper( a[i] )  a[i]=tolower(a[i]) ;
                        a[i]=a[i]+32;
        }
        printf(" output string is %s:", a);
}
```

Counting vowels in a given string

```
void main()
{       char a[100], ch;   int i, count=0;
        printf("\n enter string :");  gets(a);
        for(i=0; a[i]!='\0'; i++)
        {       ch=toupper( a[i] );
                if(  ch=='A' || ch=='E' || ch=='I' || ch=='O' || ch=='U' )
                        count++;
        }
        printf( "no of vowels is %d", count);
}
```

ch=toupper(a[i]): this function converts into upper-case, so that we can compare in only one case of upper. (instead comparing like ch=='a' || ch=='A' || ch=='e' || ch=='E'||….)

Printing ASCII values of a given person name

This demo program accepts a name of a person and prints ASCII value of each character.

Input: enter name of a person: Sri Hari

Output: S=83, r=144, i=105, ' '=32, H=72, a=97, r=144, i=105

```
void main()
{       char a[100], i, n;
        printf("\n enter name of a person :");
        gets(a);                                          // gets( &a[0] );
        printf("\n the list of ascii values are ");
        for(i=0; a[i]!='\0'; i++)                         // loop to print all character's ASCII values
           printf("%c = %d , ", a[i], a[i] );
}
```

The i/o function gets() accepts "Sri Hari" char by char from keyboard and stores into given array in &a[0], &a[1], &a[2],etc. The "%c" prints the character in symbolic form, whereas %d prints its ASCII code.

Here the for-loop repeats until null character is found.

Finding Length of string

```
    input: INDIA                input: All fruits are not apples
  output: length=5.           output: length=25
     void main()
     {        char a[100], count=0;
              printf(“\n enter any string :”);
              gets(a);
              for(i=0; a[i]!=’\0’; i++)           // counting all characters until null is found
                      count++;
              printf(“\n length = %d”, count);
     }
```

Here the loop repeats until the null character is reached and same time the characters are counted using ‘**count’** variable to find length. Generally, we don’t consider null character as a part of length.

Finding sum of digits in a given string

This program accepts a string from keyboard; the string can have digits in the input, finding sum of such digits. For example, the string is “ABC567DEF9PQ12”,

output is: 5+6+7+9+1+2 →30

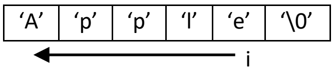
```
   void main()
   {  char a[100], i , sum=0;
      printf(“enter a string :”);
      scanf(“%s”, &a[0] );
      for( i=0; a[i]!=’\0’; i++)
      {        if( a[i]>= ’0’  && a[i] <= ‘9’ )
               {        sum=sum+(a[i]-48);       // sum=sum+(a[i]-‘0’);
               }
      }
      printf(“\n  sum = %d”, sum);
   }
```

The ASCII value of ‘0’ is 48, ‘1’ is 49, ‘2’ is 50, etc; so 48 subtracted to convert ascii value to numeric value

Printing string in reverse form

```
  ip:  Apple
 op:  elppA
```

logic: first it moves the loop variable ‘i’ to end of string, later prints char by char from last to first in a string.



```
     void main()
     {    char a[20], i;
          printf(“\n enter any string :”);
          gets(a);
          for(i=0; a[i]!=’\0’; i++)      //  loop stops when null is found.
          {                              //  this is empty loop,  the loop stops when ‘i’ reaches to null character.
                                         //  after loop, ‘i’ value attains length of string.
          }
          for( i--;  i>=0;  i-- )        // printing  string in reverse order (char by char in reverse order)
              printf(“%c”, a[i]);
      }
```

Calculating number of words in a given multiword string

This program accepts a multi-word string and finds number of words, assume there may be more than one space between two words but not with other symbols like dot, coma, etc.

input:	A	L	L		A	P	P	L	E	S					A	R	E						R	E	D	\0

```
void main()
{   char a[100];   int i, count=1;
    printf(“\n enter a multi word string :”);
    gets(a);
    for(i=0; a[i]!=’\0’; i++)
    {         if( a[i]==’ ‘ )                  // if space is found then count as one word
              {    count++;
                    while( a[i+1]==’ ‘)   // if next char is also space, then skip by incrementing loop.
                         i++;
              }
    }
    printf(“\n number of words = %d”, count);
}
```

Let us observe the input string, contained more than one space between words. The inner while loop used to skip extra spaces between words. When the index pointer ‘i’ moves to first space then ‘count’ increments and all succeeding spaces if any are skipped by the inner while loop.

Converting each word first letter to upper case and remaining to lower case.

ip: all apples are good, some are in white color and some are in red color.

op: All Apples Are Good, Some Are In White Color And Some Are In Red Color.

(Let us assume, the words in a string are separated by single white space)

```
void main()
{   char a[100]; int i ;
    printf(“enter string :”);
    gets( a );
    a[0]=toupper( a[0] );                  // converting first character to upper case.
    for(i=1; a[i]!=’\0’; i++)
    {         if( a[i-1]==’ ‘ )              // if previous char is space then word started
                     a[i]=toupper( a[i] );
              else  a[i]=tolower( a[i] );    // if a[i] is not first letter of word then convert to lower
    }
    printf(“output string is %s ”,  a);
}
```

Accepting a text (multi line string) and printing on the screen.

This demo program accepts a multi-line string (text) in char by char from keyboard and prints on the screen. Here char by char scanned until user pressed F6 as end of input. Because gets() or scanf() does not support to scan more than one line. Here every scanned char stored into an array and later printed on the screen.

```
void main()
{   char a[1000], ch;   int i=0;
    printf(“enter text at end of input, type  F6 key :”);        // here F6 is functional single key, not F and 6
    while(1)
    {          ch=getchar();               // getchar() also accepts new line (enter key)
               if(ch==-1) break;           // the getchar() produces -1 , if input is  F6 key.
               a[i++]=ch;                  //  storing into array
    }
    a[i]=’\0’;                             // inserting null character at end of text
    printf(“\n output = %s”, a );   // printing text
}
```

```
Suppose input:   All apples are good,
                 all good are not apples,
                 some are so sweet
                 F6
      Output:    All apples are good,
                 all good are not apples,
                 some are so sweet
```

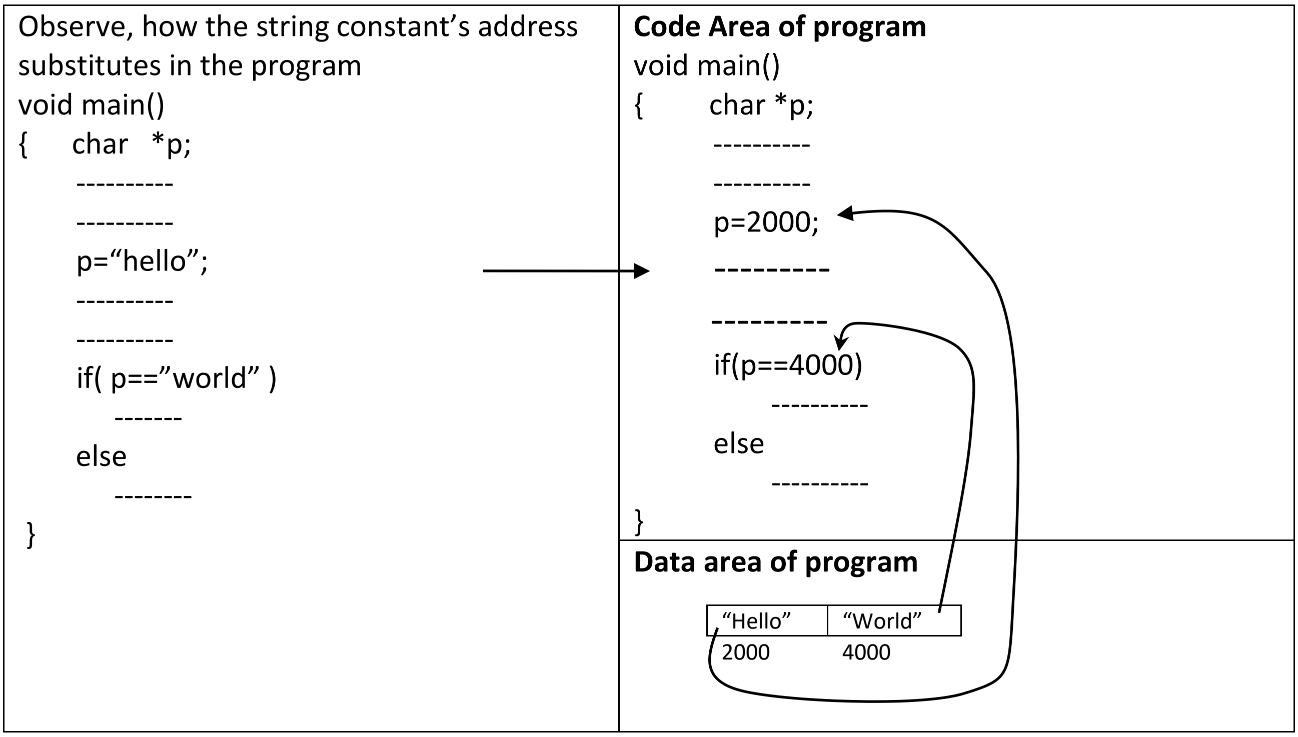
How string constants are stored in the memory

At compile time, the program’s code and data split into two separate entities and stored separately in executable file. This file loads as it is into memory before executing it. This is as

Code Area	this code area contains instructions of program	
Data Area	Heap Area	String constants
		Global variables
		Static variables
	Stack Area	Auto variables
		Function return address

The data area is again divided into two parts, **heap** area & **stack** area. In heap area, a permanent data items like string constants, global and static variables are managed. Whereas, in stack area, the temporary local variables and function return addresses are manipulated. (Refer the chapter storage-classes)

All string constants in a program are moved to data-area, whereas in the code-area their starting addresses are substituted (pasted). In this way strings are mapped with their address in the program, let us see following example



Observe, how the string constant's address substitutes in the program

```
void main()
{     char   *p;
       ----------
       ----------
       p="hello";
       ----------
       ----------
       if( p==”world” )
            -------
       else
            --------
 }
```

Code Area of program

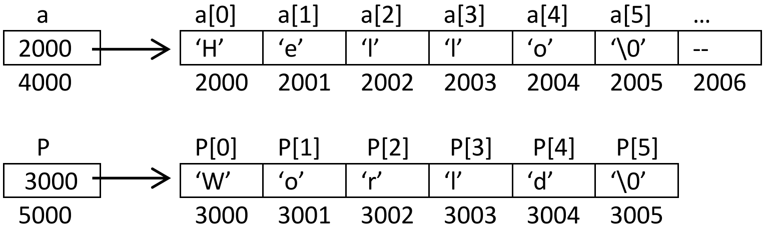
```
void main()
{       char *p;
        ----------
        ----------
        p=2000;
        ---------
        ---------
        if(p==4000)
              ----------
        else
              ----------
}
```

Data area of program

Pointer to string

As we know, string can be in the form of variable or constant. In both cases, the strings are handled in similar way. For example

```
      char  a[20]=”Hello”;
      char  *p=”World”;
```



As per first declaration, we can say that, the string is in the form of variable. In later part of the program, we can replace this string "Hello" by some other string, but new string length must be <20 chars (array-size).

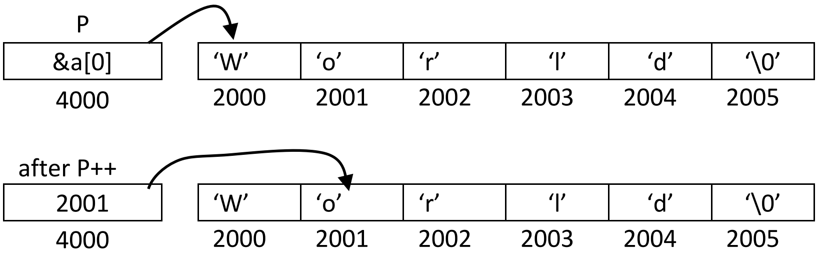
As per second declaration, we can say that, the string is in the form of constant. The space allocation for string "World" is exactly length+1(6 bytes), here first character address (&W) assigns to the pointer 'p'. Now using pointer 'p', we can read the string char by char. In later part of the program, we can replace this string "World" by some other string, but the new string length must be the same or less of this "World". If new string length is more, then it crosses memory limits and causes the program crash. This Pointer initialization widely used for string constants where no changes required in the future like company-names, country-names, person-names, codes, labels…etc.

Pointer to a string is just like a pointer to an array, either the string may be a variable or constant. In both cases, the characters are accessed in similar way. Observe the above picture. If pointer 'p' is pointing to first char in a string, the expression '*p' accesses the first char, *(p+1) accesses the second char, etc. As we know, the expressions *p, *(p+1), *(p+2)... can be written in array fashion as p[0], p[1], p[2],etc. For example

```
void main()
{          char a[ ]=”World”, *p;
           p=a;     // p=&a[0];
           for(i=0;  p[i]!=’\0’;  i++)
                     printf(“%c”, *(p+i)  or p[i]  );
```

```
            for(i=0;  *p='\0';  i++)
                    printf(“%c”, *p++ );       // this *p++ is nothing but  *p  and   p++
            }                                  //   this is equal to:  printf( “%c”, *p);  p++;
    }
```

Above two loops print same output. In first loop, the ‘p’ will not be moved/incremented, but in second loop, the pointer ‘p’ advances to next char every time. After loop, the ‘p’ moves to null char of string. This is shown below picture



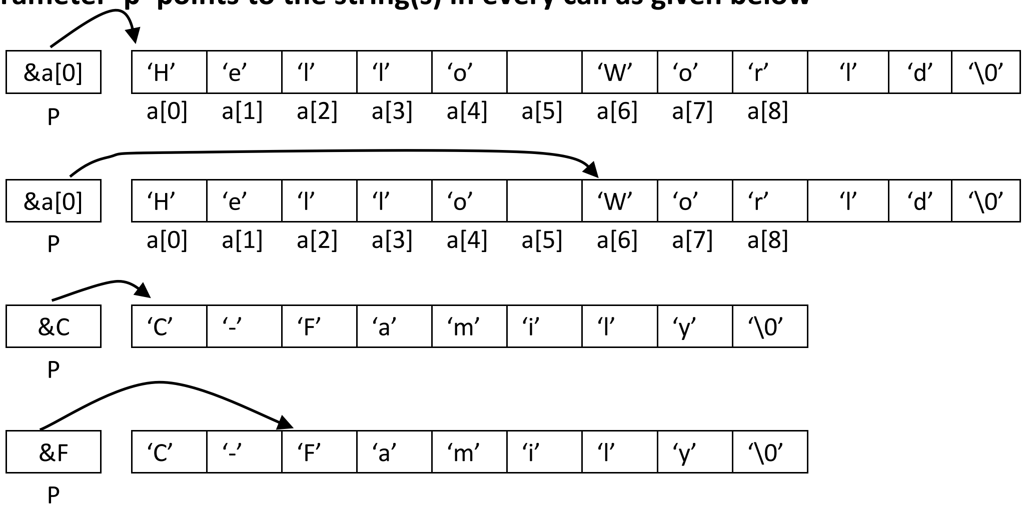
Passing string to a function

Already we know that, how to pass an array to a function. Similarly, we can pass a string as an argument to a function. Passing string to function is same as passing array to function. Here address of string is passed as an argument and the receiving parameter must be “char*” type pointer.

This program demonstrates how to pass a string to the function

```
    void main()
    {   char a[20]= “Hello World”;
        display(a);                    // display(  &a[0]  );
        display(a+6);                  // display(  &a[6]  );
        display(“C-Family");           // display(  &C  );
        display("C-Family”+2);         // display(  &F  );
    }
    void display( char *p or char p[ ] )         // both declarations are same ( use only one)
    {   int i;
        for( i=0; p[i] !='\0'; i++)
              printf ( %c", p[i] );    // printing char by char  or   use printf(“%s” , p);
    }
    output: Hello World
               World
               C-Family
               Family
```

The pointer parameter ‘p’ points to the string(s) in every call as given below



As we know, the parameter declaration “char *p” in pointer style is informal and sometimes confusion. Most of the programmers prefer array-like-pointer declaration as “char p[]”. This is formal and it tells that ‘p’ is receiving string address rather than single char address. This parameter declaration “char p[]” seems to be an empty-array, many people think like that, but implicitly it is a “char *p” pointer. This declaration provided for easy understanding and nothing more

While manipulating strings, we often need to perform some common operations like copying a string, comparing, reversing, finding sub-string, converting text format to numeric, etc. As these operations are common in programming and we prefer to abstract (hide) these tasks by writing separate functions for each operation. The string library functions named strlen(), strcpy(), strcmp(), strrev(),etc are already exist to cooperate the programmer instead of writing from low level. The following programs show, how string library functions are developed and used. Here some sample user-defined functions are implemented to know how library function works internally.

Finding length of string using function

This function takes string as an argument and returns the length of it

```
int  myStrlen( char *p )
{   int i;
    for(i=0; p[i]!=’\0’; i++)
    {                              // empty loop, indirectly calculates the length to ‘i’
    }
    return(i);                    //  after loop, the value of ‘i’ contains length, so returning ‘i’ as length
}
void main()
{   char a[100]=”hello world”;  int k;
    k=myStrlen(a);                  // myStrlen( &a[0] );
    printf(“\n length1 = %d”, k);
    k=myStrlen(“Computer”);      // myStrlen( &C );
    printf(“\n length2 = %d”, k);
}
Output: length1=11
           length2=8
```

Copying a string from one to another array

In programming, it is often need to copy a string from one to another location. For this, if one tries to copy a string using assignment operator then compiler shows an error.

```
For example:   char x[20]=”Hello World”;
               char y[20];
               y=x;              // y=&x[0];
```

He we expect that, the string “Hello World” copies from x[] to y[], but compiler gives an error, because unknowingly we are trying to assign the address of ‘&x[0]’ to ‘y’ (here ‘y’ is not a pointer, so it is an error). The solution is, copying char-by-char from source to destination array.

```
y[0]=x[0]
y[1]=x[1]
y[2]=x[2] …Using loop. Let us see, how following function copies it.
```

```
void myStrcpy( char y[] , char x[] )        // y=x;
{   int i;
    for(i=0; x[i]!=’\0’; i++)
          y[i]=x[i];
    y[i]=’\0’;                  // inserting null at end of string.
}
```

Main() fn to check the above function

```
void main()
{   char x[20]=”hello world”, y[20];
    myStrcpy ( y , x );             // y=x
    printf(“\n After copying1, the string is = %s”, y);
    myStrcpy( y, “Apple”);          // y=”Apple”
    printf(“\n After copying2, the string is = %s”, y);
}
Output:  After copying1, the string is = Hello World
         After copying2, the string is = Apple
```

// the above function can be written in pointer notation as

```
void myStrcpy( char *y ,  char *x )
{     while(*y++ = *x++);
}
```

here the value ***x** assigns to ***y**, later checks the ***y** value, if it is null then loop terminates, otherwise, y++ & x++ get incremented and loop continues till null is found.

some Valid and invalid assignments on strings

// valid assignments	// valid assignments	// invalid assignments	// valid assignments

char a[4];
a[0]=’A’;
a[1]=’B’;
a[2]=’C’;
a[3]=’\0’;
	char a[4];
	char *p=&a[0]
	p[0]=’A’;
	p[1]=’B’;
	p[2]=’C’;
	p[3]=’\0’;
	Here ‘p’ is pointing to array a[]
		char *p;
		p[0]=’A’;
		p[1]=’B’;
		p[2]=’C’;
		p[3]=’\0’;
		because, the ‘p’ is not pointing any memory
			char *p;
			p=malloc(4);
			p[0]=’A’;
			p[1]=’B’;
			p[2]=’C’;
			p[3]=’\0’;
			because, ‘p’ is pointing to dynamic memory

char a[4];
myStrcpy(a,”ABC”);
this task is just like above assignments, it is valid.
	char a[4];
	char *p=&a[0];
	myStrcpy(p,”ABC”);
	this task is just like above assignments, it is valid.
		char *p;
		myStrcpy(p , ”ABC”);
		this task is just like above assignments, invalid.
			char *p;
			p=malloc(4);
			myStrcpy(p , ”ABC”);
			this is just like above task. It is valid

Comparison of two strings

```
void main()
{   char x[20]=”Hello”, y[20]=”Hello”;
    if( x==y) printf(“equal”);              // if(  &x[0] == &y[0]  )
    else printf(“not equal”);

    if(“Hello”==“Hello”) printf(“equal”);
    else printf(“not equal”);
}
    Output: not equal
               not equal
```

Even though two array contents are same, the output will be ‘not equal’, because, unknowingly, we are trying to compare base address of two arrays, if(x==y) is nothing but if(&x[0] == &y[0]). As we know, two arrays will not be located in the same location of RAM. So always not equal.

In second if-statement also, the base address of two strings if(&H== &H) are compared and we get again “not equal”. Because two string constants will not be located in the same location.

The solution is, compare char by char using loop, which is as given below

```
void  myStrcmp( char x[], char y[] )
{   for( int i=0; x[i] != ’\0’; i++)
    {     if( x[i] != y[i] )
               break;
    }
    return x[i]-y[i];            // returning ascii difference
}
```

If both strings are equal then returns 0, otherwise returns first un-matched char’s ASCII difference of two strings. It return +ve, if string1>string2 in dictionary order(Alphabetical order) or else –ve.

For example, if two strings are:

```
    “Hello” and “Hello” then returns (0);
    “World” and “Words” then returns (‘l’-‘d’);     // ascii difference is +ve  ,   here World>Words
    “Words” and “World” then returns (‘d’-‘l’);     // ascii difference is –ve  ,   here Words<World
```

The main() function to check the above function is

```
void main()
{   char x[50], y[50];   int k;
    printf(“\n enter string 1 and string 2:”);
    gets(x);   gets(y);
    k=myStrcmp(x,y);
    if( k==0)  printf(“string1 == string2”);
    else if( k<0)  printf(“string1<string2”);
    else   printf(“string1 > string2”);
}
```

The above function myStrcmp() can be written in pointer notation as

```
int  myStrcmp(char *x , char *y)
{   while(*x && *x==*y)
    {    x++; y++;
    }
    return *x - *y;
}
```

Concatenation of two strings

Sometimes, it is required to attach one string at the end of another string and this operation is often called string concatenation.

Let x[]=”hello”, y[]=”world” → after x=x+y → the x[] would be “helloworld”

```
void myStrcat( char x[], char y[] )
{       int i , j;
        for( i=0; x[i]!=’\0’; i++)
        {      // empty loop, repeats until null character is found
        }
        for( j=0; y[j]!=’\0’; j++)          // attaching second string at end of first string
             x[ i+j ] = y[ j ];
        x[ i+j ]=’\0’;                      // inserting null character at end of first string
}
void main()
{       char a[20]=”Hello”, b[20]=”World”;
        myStrcat(a,b);
        printf(“\n after concatenation, the  string1= %s”, a);
        myStrcat( a, “ is Wonder”);
        printf(“\n after concatenation, the  string2= %s”, a);
}
```

```
output:  after concatenation, the string1= HelloWorld
         after concatenation, the string2= HelloWorld  is  Wonder
```

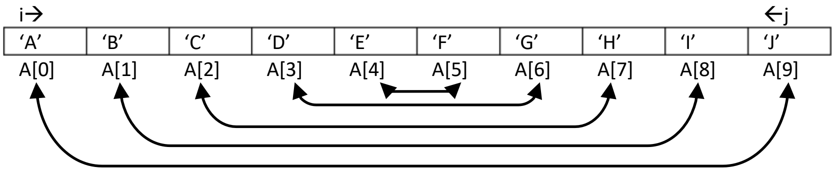
above function using pointer

```
void  myStrcat( char *x, char *y )
{       while(*x) x++;            // moving ‘x’ pointer to end of first string
        while(*x++ = *y++);       // this loop copies all characters to ‘y’ including null
}
```

Here first, the expression ‘***x’** evaluates, if it is null-char (0) then loop stops, otherwise, x++ is incremented to point to next character in the string. In this way, *x is checked and x++ incremented until null-char is found. The second loop copies y[] to x[] which is already discussed above.

Reversing a string

```
void myStrrev( char *a )
{   int  i;
    for(i=0;  a[i]!=’\0’;  i++)
    {       // empty loop to calculate length, loop ends when ‘i’ reached to null char, finally ‘i’ contains length
    }
    for( j=i-1, i=0;  i<j;  i++, j--)  //  ‘i’ points to first element, ‘j’ points to last element in the string.
    {       t=a[i];      // swapping
            a[i]=a[j];
            a[j]=t;
    }
}
```



```
void main()
{   char a[100];
    myStrrev( a );
    printf("\n string after reversing %s", a );
}
 ip: ABCDEFGHIJ
op: JIHGFEDCBA
```

String library functions

In previous examples, we have seen how to write & use string functions. Our C standard library contains a wide range of well-developed string functions, which help us in minimizing the coding time and building compact & reliable code without rewriting same code again and again. This library simplifies the logic of complex programs without feeling any difficulty while handling strings. These function's prototypes available in "string.h" header file. So we can simply include and use the functions in our programs.

strlen()

This function takes string address as an argument and returns its length. It counts & returns the number of characters in a given string without counting null character.

```
      syntax is:  intVariable=strlen( string );
proto-type is:  int strlen( char * );
       void  main()
      {   char a[30]="Hello World";
          k=strlen(a);
          printf("\n string1 length = %d", k);
          k=strlen("This is C-Family");
          printf("\n string2 length = %d", k);
      }
      Output:    string1 length=11
                string2 length=16
```

strcpy()

it copies a string from one location to another location and also appends null at end of string.

```
      void main()
      {   char a[30]="Hello World";
          char b[30];
          strcpy(b, a);     // copies char by char including null.
          printf("\n after copying, the string1 is : %s", b);
          strcpy(b, "India");
          printf("\n after copying, the string2 is : %s", b);
      }
Output:   After copying, the string1 is : Hello World
          After copying, the string2 is : India
```

strcat()

here ‘cat’ means concatenation, it attaches one string at the end of another string. That is, it concatenates two strings like addition.

```
      syntax:  strcat( string1, string2 );           // string1=string1+string2;
proto-type:  char*  strcat( char*, char* );
      void main()
      {   char a[20]=”Hello” ,  b[20]=”World”;
          strcat(a,b);
          printf(“output1 = %s”, a);
          strcat(a,”India”);
          printf(“output2 = %s”, a);
      }
Output:  output1=HelloWorld
         output2=HelloWorldIndia
```

strrev()

it reverses the given string

```
             syntax: strrev(string);
          proto type: char*  strrev(char*);
      void main()
      {   char a[20]=”Hello”
          strrev(a);
          printf(“output1 = %s”, a);
          strrev(a);
          printf(“output2 = %s”, a);
      }
output1:  olleH
output2:  Hello
```

strcmp()

compares two string whether they are equal or not? If both strings are equal then returns (0), otherwise returns “first un-matched characters ASCII difference in +ve or –ve”.

If returned +ve value, then string1>string2 in dictionary order (alphabetical order), or else string1<string2.

```
      void main()
      {   char a[30]=”Hello World”;
          char b[30]=”Hello World”;
          K=strcmp(a,b);            // copies char by char including null.
          if( k==0)  printf(“string1==string2”);
          else if(k<0)  printf(“string1 < string2”);
          else  printf(“string1 > string2”);

          k=strcmp(“xyz”, “abcdef”);
          if( k==0)  printf(“string1==string2”);
          else if(k<0)  printf(“string1 < string2”);
          else  printf(“string1 > string2”);
     }
   op: string1==string2
        string1 > string2
```

strcmpi() or stricmp()

It compares two strings, which is same as strcmp(), but it **i**gnores the upper/lower case of alphabets.

Syntax: int strcmpi(string1, string2);

```
    void main()
    {   if(strcmpi(“DELHI", "delhi")==0)
                printf("Both are equal");
        else    printf("Not equal");
    }
```

Outputs: Both are equal

strncmpi() or strnicmp()

It compares two strings up to some specified number of characters, and also ignores the case difference.

Syntax: int strncmpi(string1, string2, no-of-chars-to-compare);

```
        void main()
        {        if( strncmpi(“Garden", "gardening",3) == 0 )
                         printf("Both are equal");
                 else    printf(“Not equal");
        }
```

Output: Both are equal

strupr()

It converts all lower case alphabets into upper case in a given string.

Syntax: strupr(string);

```
        void main()
        {       char a[]="Pens and Pads";
                strupr(a);
                printf("%s", a);
        }
```

Output: PENS AND PADS

strlwr()

It converts all upper case alphabets into lower case in a given string.

Syntax: strlwr(string);

```
        void main()
        {       char a[]="Jack n JILL";
                strtwr( a);
                printf("%s", a);
        }
```

Output: jack n jill

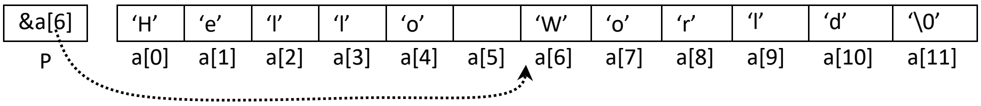
strchr()

It searches for a given character in a string, it searches one by one from first position towards end of string. It returns first occurrence of a specified character address if it is found. Otherwise it returns NULL value.

Syntax: char* strchr(string, char to search);

```
    void main()
    {   char *p;
        p=strchr("Hello world", ‘w’);            // returns “&w” in “hello world” string.
        printf("%s", p);
    }
```

This printf() will prints the message **"world" ,** because 'p' pointing rest of the string "world"

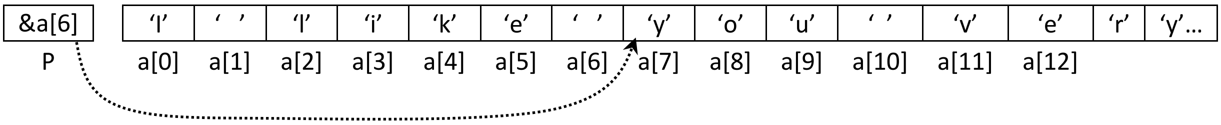


strstr()

it searches for a sub-string in a main-string. It returns the first occurrence of sub-string address if it is found. Otherwise, it returns NULL, if not found.

```
void main()
{       char *p;
        p=strchr("I like you very much", "you");
        printf("%s", p);
}
```

This printf() will print the message "you very much"



atoi()

It converts a string-value to integer-value.

Syntax: int atoi(string);

```
Example:   int k;
           k= atoi("123")+atoi("234");
           printf(" output = %d", k);
          output will be: 357
```

atol()

Converts string-value to long-int value

Syntax: long int atol(string);

```
Example:   long int k;
             k= atol("486384")-atol("112233");
           printf("output = %ld", k);
           output will be = 374151
```

atof()

Converts floating point text format value to double value.

```
Example: float f;
          f=atof("3.1412")*5*5;
         printf("%f ", f);
          output will be: 78.530000
```

itoa(),itoa(),ultoa()

These functions converts a given number (int/long int/unsigned long int)equivalent to string format based on the given numbering system radix value.

These functions take three arguments, the numeric value, target string address in which the value to be stored and radix value. Finally returns the target string address, so that function-call can be used as argument/expression.

```
syntax:    char*  itoa(int value, char *targetStringAddress,  int radix);
example:   char temp[50];             // to store output string
           itoa(45, temp, 2);         // converting 45 to binary system value
```

```
  printf("output : %s", temp);
 output : 101101
  itoa(45, temp, 8);              // converting 45 to  octal system
  printf("output : %s", temp);
  output :37
```

sprintf()

It is like printf() function but with a little difference. The printf() puts the output data on the screen, whereas sprintf() puts the output data in a given char-array.

Syntax: sprintf(array, format-string, arguments);

Example: char a[100];

```
sprintf(a, "Idno=%d , name = %s , salary = %f", 101, "Srihari", 50000.00);
```

Observe the following output, how it copies formatted-string to array a[]

```
printf("%s", a);  → Idno=101, name=Srihari, salary=500000.00
```

sscanf()

It is also like normal scanf() function. But scans formatted-string from given array instead of keyboard.

Syntax: sscanf(array, format-string, &variable1 [, &variable2,..]);

Example: char a[] = "101 Srihari 50000";

```
sscanf(a, "%d %s %f ", &idno, &name, &salary );
printf(" %d  %s  %f", idno, name, salary);  → 101  Srihari 50000
```

List of N Strings (Strings in 2D array)

(Before coming to this topic, you better to revise a topic: pointer to 2D array)

This chapter mainly concentrated on how to represent array of strings and operations such as inputting, printing, sorting, searching, and finding sub-strings, etc.

To store a group of strings, we need a data structure like 2-dimensional array with N*M size, where N is number of strings and M is maximum size of string length.

For example: char a[20][30];

in this array, we can store 20 strings each with a maximum length of 29 characters (1 for null).

Accessing of these strings is same as accessing of 2D arrays.

Initializing N strings

char a[10][20]={ "APPLE", "ORANGE", "BANANA", " GUAVA",…etc};

We can imagine, how the strings are stored in 2D array

'A'	'P'	'P'	'L'	'E'	'\0'				
'O'	'R'	'A'	'N'	'G'	'E'	'\0'			
'B'	'A'	'N'	'A'	'N'	'A'	'\0'			
'G'	'U'	'A'	'V'	'A'	'\0'				

How to accept & print a list of strings?

Scanning a group of strings is just like scanning 'N' individual strings. The way is, scanning string after string from keyboard using loop statement. If we use 2-dimensional array then we have to pass each row base address of array to the scanf() function. The following program shows how to scan and print.

```
void main()
{   char a[20][30];
    printf("Enter number of strings :");
    scanf("%d" &n);
    for(i=0; i<n; i++)
    {    printf("Enter string %d :", i+1);
         flushall();
         gets( a[i] );   // gets( &a[i][0] ); → passing each row base address  to the gets() function
    }
    for(i=0; i<n; i++)
         puts( a[i] );  // puts( &a[i][0] ); → passing each row address to puts() function
}
```

We know, the 2-dimensional array expressions a[i][j] converts into *(a+i*column-size+j);
Whereas, the expression a[i] converts into (a+i*column-size), which gives ith row address.
(refer *pointer to 2D array topic, page 199)*

This program finds the biggest string from N strings.

(Here biggest means alphabetically comes to last in dictionary order)

```
#include<stdio.h>
#include<string.h>
void main()
 {       char a[20][30],  *temp, n, i;
         printf(enter number of strings :");
         scanf("%d",&n);
         for(i=0; i<n; i++)
         {     printf("enter string %d :", i+1);
               fflush(stdin);    // clears the keyboard buffer
               gets(a[i]);
         }
        temp=a[0],              // assume the first string as the biggest
        for(i=1; i<n; i++)
          {     if( strcmp(a[i], temp) > 0 )
                     temp=a[i];
          }
          printf("Biggest string = %s", temp);
     }
```

In the above program, the first string is considered as bigger; therefore its base address is stored in 'temp' pointer. Later all strings are compared with 'temp', if any bigger string is found in the rest of array, then its address is copied to 'temp'. In this way, finally biggest string address will be stored in 'temp'.

Sorting of ‘N’ Strings

We can sort strings just like integers, but strings cannot be compared using > operator, because base address are compared instead of string’s characters. So the library function strcmp() is used to compare two strings. Following program gives demo for sorting of N strings.

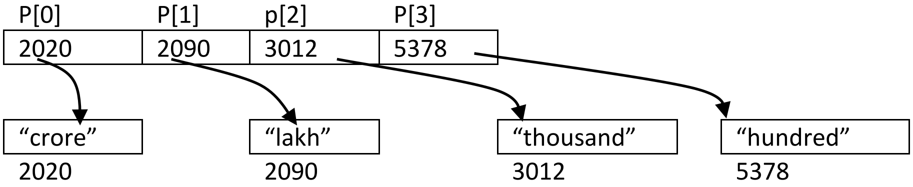
```
#include<string.h>
void main()
{     char a[20][30]; int n;
      scan(a,&n);          // scans n strings
      sort(a,n);           // using bubble sort
      print(a,n);          // printing all string
}
void scan(char a[ ][30] , int *pn)
{     int i;
      printf(“enter number of strings :");
      scanf("%d", pn);
     for(i=0; i<*pn; i++)
     {     printf(“enter string %d :",i+1);
           fflush(stdin);             // clears keyboard buffer
           gets( a[i] );              // inputting  ith string
     }
}
void sort( char a[ ][30] ,  int n )  // bubble sort
{     int i, j, k;   char t[30];
     for(i=n-1; i>0; i--)
        for(j=0; j<i; j++)
        {    if( strcmp(a[j],a[j+1] > 0 )
             {        strcpy(t,a);             // swapping the strings
                      strcpy(a,b);
                      strcpy(b,t);
             }
        }
}
void print(char a[][30],  int n)
{    int i;
     for(i=0; i<n; i++)
         puts(a[i]);
}
```

Initialization of strings to the pointer array

If string constants are fixed and do not require any modifications during program execution then it is better to initialize to the pointer array rather than normal 2D array, because, it saves a lot of memory space and also easily manageable such as swapping of strings, sorting of strings, inserting a new string b/w strings, deleting existing string, etc. Here addresses are manipulated instead of its data.

For example: char *p[] = {"crore", "lakh", "thousand", "hundred"};

Here all these strings are stored in the data area of a program, and their addresses are assigned to the pointers. Here memory occupied by each string is equivalent string length+null. This is as given below



This program accepts a number and prints in English words

Input: 2345

Output: two thousand three hundred and four five

```
#include<stdio.h>
void main()
{       char *a[] = {"", "one","two","three","four","five","six","seven","eight","nine"};
        char *b[] = {"","", "twenty", "thirty","foury","fifty","sixty", "seventy", "eighty","ninty"};
        char *c[]={"crore", "lakh", "thousand", "hundred and",""};
        long int d[]={10000000l,100000l,1000l,100,1}, n, quotient, i;
        printf("enter any number :");
        scanf("%ld",&n);
        for(i=0; n>0 ;i++)
        {       quotient=n/d[i];
                if(quotient==0) continue;
                if(quotient<20)  printf(" %s ", a[quotient] );
                else  printf(" %s %s ", b[quotient/10], a[quotient%10] );
                printf(" %s ", c[i] );
                n=n%d[i];
        }
}
```

Here the input number divides repeatedly with crore, lakh, thousand, hundred, etc

If quotient is zero then there is nothing to print on the screen so loop continues for next cycle.

Otherwise, prints equivalent words in English and takes remainder as next input for next cycle.

Recursion

Calling a function within the same function is called as recursion. It is the process of defining something in terms of itself, and sometimes also called as circular definition.

The syntax, the usage, and the execution of recursive function is just like any normal function. There is no technical difference between recursive and normal function. However, for beginners, the logic of recursion is somewhat difficult to understand and writing programs in it. The major confusion is, understanding the logic regarding self-calling, self-returning, and how the arguments & local variables are managed in recursive-calls. If you are familiar with functions, here by observing some simple examples we can follow the logic easily. Here some examples explained with pictures to understand the basic concept of recursion. Before going to examples, let us see some mathematical analysis on recurrence relations.

1) f(n)=n+n/2+n/4+n/8+….+1
the recurrence relation is:
f(n)=n+f(n/2) if n>1
f(n)=1, if n==1

2) fact(n) = n*n-1*n-2*….3*2*1
the recurrence relation is:
fact(n)=n*fact(n-1), if n>1
fact(n)=1, if n==1

3) power(x,y) = x*x*x*x….y times
the recurrence relation is:
power(x,y) = x*power(x,y-1),   if y>1
power(x,y) = 1, if y>0

==

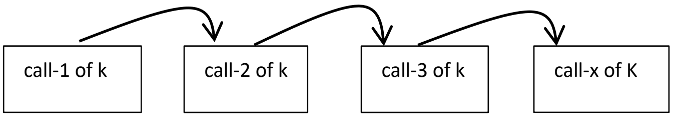
First demo program

```
void main()
{         printf("hello  ");
          main();                    // self-calling (recursive-call )
}
```

output: hello hello hello hello … until stack overflow (memory full)
Here the main() fn called recursively inside its body, therefore the control re-enter into same body again and again as it repeatedly calling again and again(), it prints "hello" many times, but in every recursive call some memory is consumed by operating system(OS) to store function-return-address, so after some calls, the memory gets full and program stopped forcefully by OS. This force stop is also called program crash. This is just a demo program and I am trying to say about recursion.

Let us see, how local variables behave inside recursive function.

```
void main()
{          int k=1;
           if( k==5)
                return;
           printf("%d  ", k );
           k++;
           main();                     // self-calling (recursive-call)
}
```



Here we expect the output 1, 2, 3, 4, but it is wrong. It shows 1, 1, 1, 1, 1, … until memory is full. For every recursive call, one new copy of 'k' is created and initialized with 1(as above pic). So output is 1, 1, 1, 1….. Here the k value never reaches to 5, because the 'k++ incremented' value at one call, will not be affected to next call of 'k', so always prints 1,1,1,1….. until memory is full.

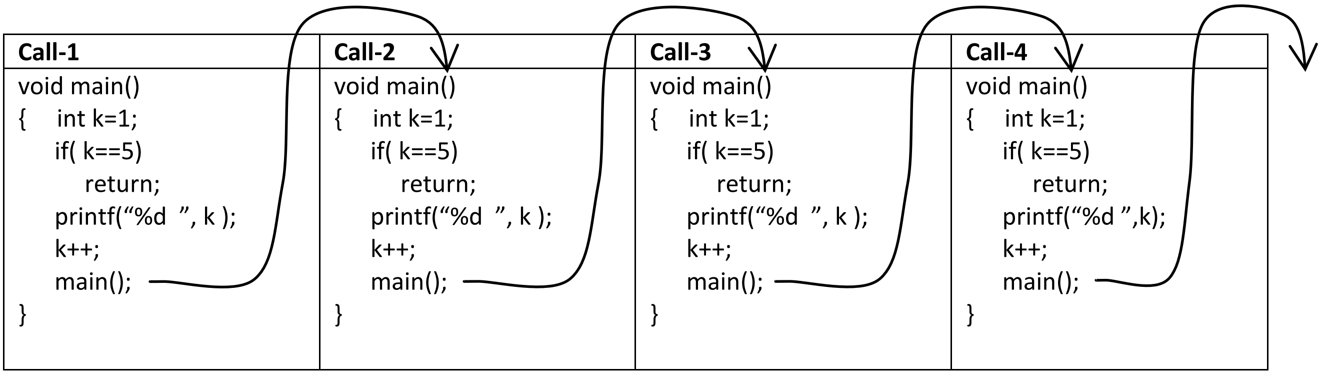
Note: as new copy of 'k' is created in every call, the memory gets full after some calls and OS stops our program.

Points to understand Recursion

- When a recursive function is called itself, the programmer should be treated as, he is calling another function(copy) of same code as shown in the below figure. Of course, in the memory, same function body will be executed like a loop.

- For every recursive call, one set of local variables (in this example 'k') are created and available up to that call is terminated. That is, when such call is terminated, all local variables are freed (deleted).

- Recursive function executes like a loop statement, as we know, every loop has a terminating condition, in the same way, to stop the recursive calls, there must be a termination condition and it is often called base condition. Generally, it appears at the beginning of function (we will see in the next examples).
  This figure explains how the recursive calls should be imagined for above example



Call-1	Call-2	Call-3	Call-4

void main()
{ int k=1;
if(k==5)
return;
printf("%d ", k);
k++;
main();
}
	void main()
	{ int k=1;
	if(k==5)
	return;
	printf("%d ", k);
	k++;
	main();
	}
		void main()
		{ int k=1;
		if(k==5)
		return;
		printf("%d ", k);
		k++;
		main();
		}
			void main()
			{ int k=1;
			if(k==5)
			return;
			printf("%d ",k);
			k++;
			main();
			}

*Here in every recursive call, one fresh copy of 'k' is created initialized with 1, the incremented k++ will not be affected to next call. So output is 1 1 1 1….

Taking 'k' as parameter instead of local variable, let us see following examples

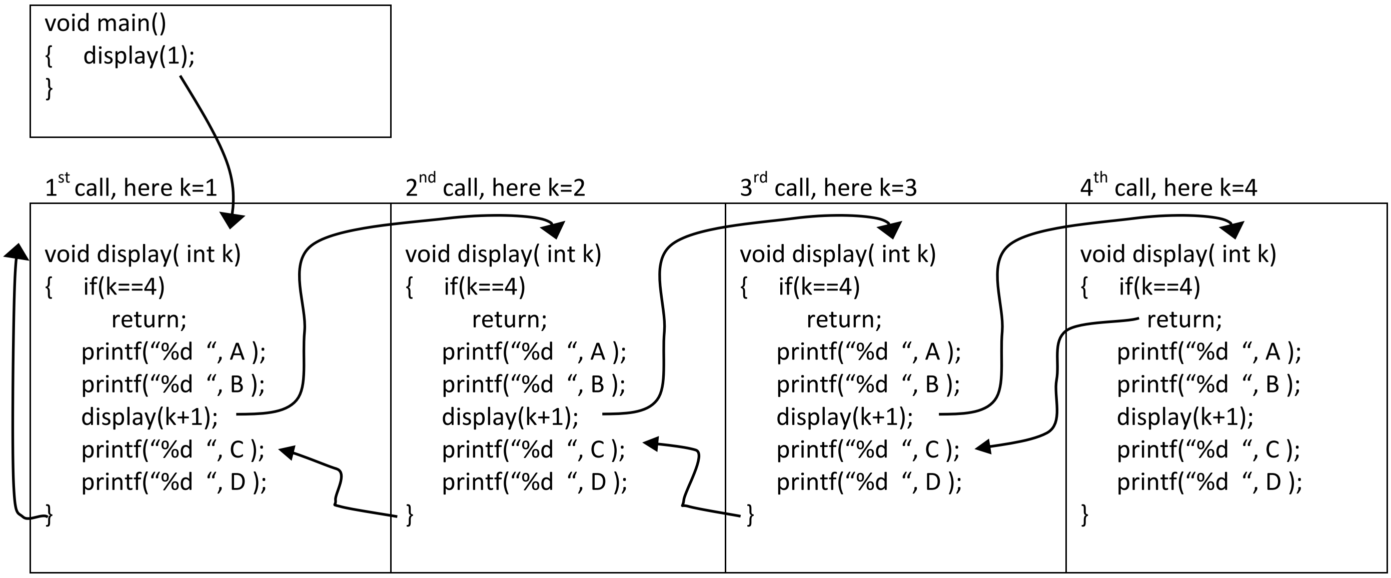
Expect the output of following program

```
void main()
{         display(1);                  // passing the argument 1
}

void display( int  k  )
{         if(k==4)
              return;                  // termination of recursion
          printf(" A " );
          printf(" B " );
          display(k+1);                // recursive call
          printf(" C " );
          printf(" D " );
}
```

Output: AB   AB   AB   CD   CD   CD

For every next call, we are passing k+1 value as argument, therefore at first-call k=1, at second-call k=2,… at fourth-call k=4, here terminates the recursion and returns the control.

When above 'return' statement gets executed in recursion, many people think that, the control immediately transfer back to main(), this is the big misunderstanding people think like that, actually here the control returns to previous-call in this series of calls. Here is the following picture explains it clearly.



```
void main()
{    display(1);
}
```

1st call, here k=1 | 2nd call, here k=2 | 3rd call, here k=3 | 4th call, here k=4

```
void display( int k)
{    if(k==4)
         return;
     printf(“%d  “, A );
     printf(“%d  “, B );
     display(k+1);
     printf(“%d  “, C );
     printf(“%d  “, D );
}
```

*** The last closing braces of function works as 'return' statement, observe above shown picture.**

*** Here 4th call returns to previous 3rd call, and 3rd call returns to previous 2nd call, in this way recursive calls are returned & terminated. The above picture is the best view how the recursive calls can be imagined.**

*** In recursive function, all instructions which are above recursive-call are executed just before going to next call, so output is AB AB AB, and all below instructions are executed just before returning to previous call, so output is CD CD CD.**

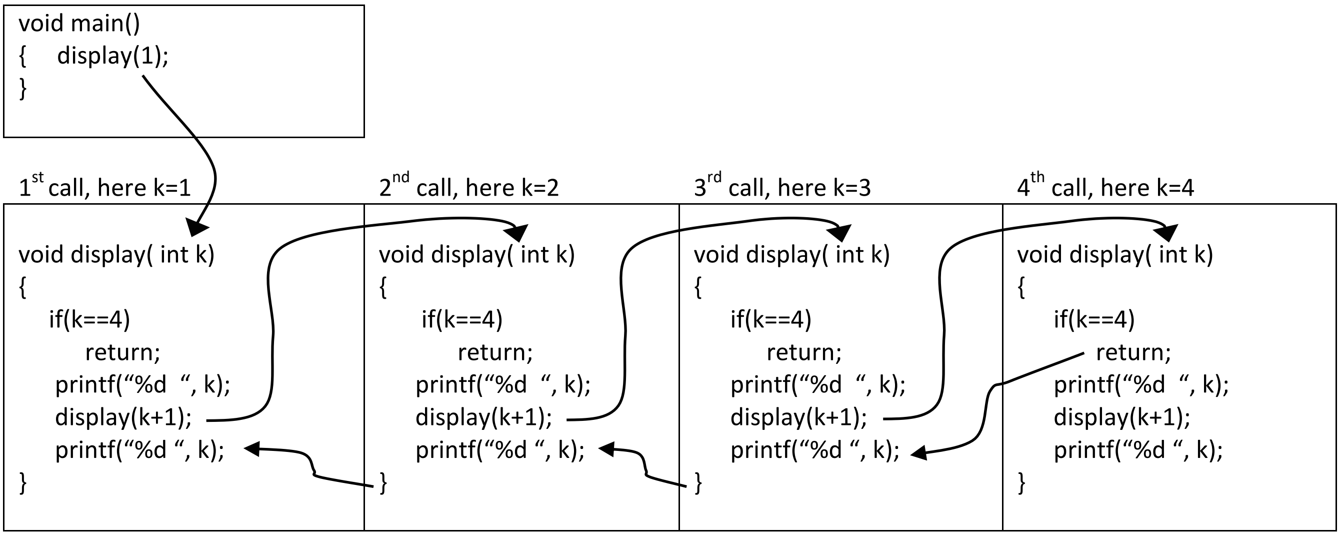
**** The main difference between loop and recursion is, loop works on single data set variables, whereas recursion works on multiple data set variables.***

*Printing 1 2 3 3 2 1 recursively

```
void main()
{         display(1);                    // passing argument 1
}
void display(int  k)
{         if(k==4)  return;
          printf("%d ", k);              // printing output before going to next-call
          display(k+1);
          printf("%d ", k);              // printing output after coming back from next-call
}
```

Here the output '1 2 3' prints by first printf(“%d “, k), whereas second output '3 2 1' prints by second printf(“%d “, k) statement. The first printf() executes just before next call is made. Whereas second printf() executes after returning from next-call.

This figure shows how the output 1 2 3 3 2 1 prints on the screen.



```
void main()
{    display(1);
}
```

1st call, here k=1

```
void display( int k)
{
    if(k==4)
        return;
    printf(“%d  “, k);
    display(k+1);
    printf(“%d “, k);
}
```

2nd call, here k=2

```
void display( int k)
{
    if(k==4)
        return;
    printf(“%d  “, k);
    display(k+1);
    printf(“%d “, k);
}
```

3rd call, here k=3

```
void display( int k)
{
    if(k==4)
        return;
    printf(“%d  “, k);
    display(k+1);
    printf(“%d “, k);
}
```

4th call, here k=4

```
void display( int k)
{
    if(k==4)
        return;
    printf(“%d  “, k);
    display(k+1);
    printf(“%d “, k);
}
```

Printing output N, N/2, N/4, N/8, N/16, ...1 using recursion

This function takes N as argument and prints the output from N to 1.

ip: N=100 op: 100, 50, 25, 12, 6, 3, 1

```
void main()
{         scanf(“%d”, &N);
          show(N);
}
void  show(int N)
{         if( N==0) return;
          printf(“%d “, N);
          show(N/2);            // sending half value to next call
}
```

Note: Most of the recursive programs covered in this chapter are simple and these can be done using any loop control statements rather than recursion. For these problems, loop is simpler than recursion; just to make easy understanding of recursion, these problems explained in recursion. But the last two problems in this chapter can’t be solved using loops.

Printing 1 to N numbers using recursive function

```
  ip: N=13
 op: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13.
void main()
{       scanf("%d", &N);
        show(N);
}
void show( int N )
{       if(N==0)  return;
        show( N-1);
        printf("%d  ", N );      // the value N is printed while returning control to previous call
}
```

when 'N' reaches to 0, the recursive calls are stopped, and prints the 'N' value from last-call to first-call while returning control to previous calls.

Guess the output

```
void main()
{       show(1);
}
void show( int  N )
{       if(N == 6)
        {   printf("  and  ");
            return;
        }
        printf("  hello  ");
        show( N + 1 );
        printf("  world  ");
}
```

Printing Multiplication table using recursion

```
Output: 8*1=8
        8*2=16
        8*3=24
        … 10 terms
void main()
{       printTable( 8 , 1 );     //  here '8' is table number , '1' is first term value.
}
void printTable( int n , int i )
{       if(i==11)  return;
        printf("\n %d*%d=%d", n, i, n*i);
        printTable(n, i+1);      //  i+1 is the next term value
}
```

Printing Multiplication table upto desirable number of times

```
void main()
{       printTable( 8 , 20 );          // here '8' is table number, 20 is the number of terms to print.
}
void printTable( int n , int i )
{       if(i==0)   return;
        printTable( n, i-1 );
        printf("\n %d*%d=%d", n, i, n*i);
}
```

Printing output 1, 2, 4, 8, 16, …. <1000 using recursion

```
void main()
{      show(1);                        //  '1' is the first value of series
}
void  show(int P)
{       if(P>1000) return;
        printf("%d  ", P );            // p = 1, 2, 4, 8, 16 …
        show( P*2 );
}
```

Printing output 1, 2, 4, 8, 16, …. 7 times using recursion

```
void main()
{       show(1, 7);                    //  '1' is the first term of series and 7 is the number of times to print
}
void  show( int P, int count )
{       if( count==0) return;
        printf("%d  ", P );            // p = 1, 2, 4, 8, 16… ,  count=7,6,5,4,3,2,1
        show( P*2 ,  count-1 );
}
```

Generate and print list of numbers from N to 1

Here N is input from keyboard and print list of numbers as long as the value of N becomes 1.

if N is even then next number of N is → N/2 (half)

if N is odd then next number of N is → 3N + 1

if input N is 13, then we have to print as: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

```
void main()
{       int N=13;
        show(N);
}
void  show(int N)
{       printf("%d  ", N);
        if(N==1)  return;
        if( N%2==0) show(N/2);
        else  show(3*N+1);
}
```

Guess the output

```
void main()
{       show( 345 );
}
void  show( int  N  )
{       if( N==0 )  return 0;
         show(N/10);
        printInEnglish( N%10 );
}
void printInEnglish( int digit )
{       if(digit==0) printf("zero");
        else if(digit==1) printf("one");
        else if(digit==2) printf("two");
        else if(digit==3) printf("three");
        -----
}
```

Guess the output

```
void main()
{       int N=8;
        show( 0,  1,  N );
}
void show( int x, int y, int N )
{       if(N==0) return;
        print("%d ", x );
        show( y,  x+y,  N-1 );
}
```

Complete the following code to print output as shown below(do not use loops)

```
123456789
 123456789
 ------
 5 rows
void main()
{       int rows=5;             // no.of rows to print
        show(rows);
}
void show( int row)
{       ------
        printRow( 9 );          //  this function prints 1 to 9 values of a row
        ------                  //  call show() recursively 5 times, for 5 rows
}
void printRow( int i )
{       if(i==0)  return;
        printRow( i-1 );
        printf("%d ", i );      // prints each row here
}
```

Guess the output (Decimal to Binary)

Process: continuously divide N with 2 and collect the remainders, the remainders collection forms a binary

```
        void main()
        {          show( 13 );
        }
        void  show( int  N  )
        {          if( N==0 )
                       return;
                   show( N/2 );       // N=N/2→ 13, 6, 3, 1    output → 1%2, 3%2, 6%2, 13%2
                   printf("%d", N%2 );
        }
```

Printing all factors of 15

```
        ip: 15   ,   op: 1, 3, 5, 15
        void main()
        {          show( 15, 1) ;
        }
        void show( int n , int i)
        {          if( i>n) return;
                   if( n%i==0 )
                        printf("%d  ", i );
                   show( n , i+1);
        }
```

Program to add 7+7+7 … 4 times (it doesn't work, check out)

```
        void main()
        {      k=find( 4 );
               printf(" %d  " ,  k ); ?
        }
        int  find ( int  count )
        {          int sum=0;
                    if( count==0 )
                        return  sum;
                    sum=sum+7;
                    return  find( count-1 );
        }
```

At 1st call	At 2nd call	At 3rd call	At 4th call	At 5th call
int sum=0 sum=sum+7 = 0+7	int sum=0 sum=0+7 =0+7	int sum=0 sum=sum+7 =0+7	int sum=0 sum=sum+7 =0+7	int sum=0 return sum // returns '0'

output: 0 (Wrong output)

The above function produces wrong output. Because, for every recursive call, one fresh copy of local variable 'sum' is created and initialized with 0. The added value 7 will not be affected to next call.

To solve this problem take 'sum' as parameter and pass 'sum' value to next-call. See next problem

Above program with 'sum' as parameter (works fine)

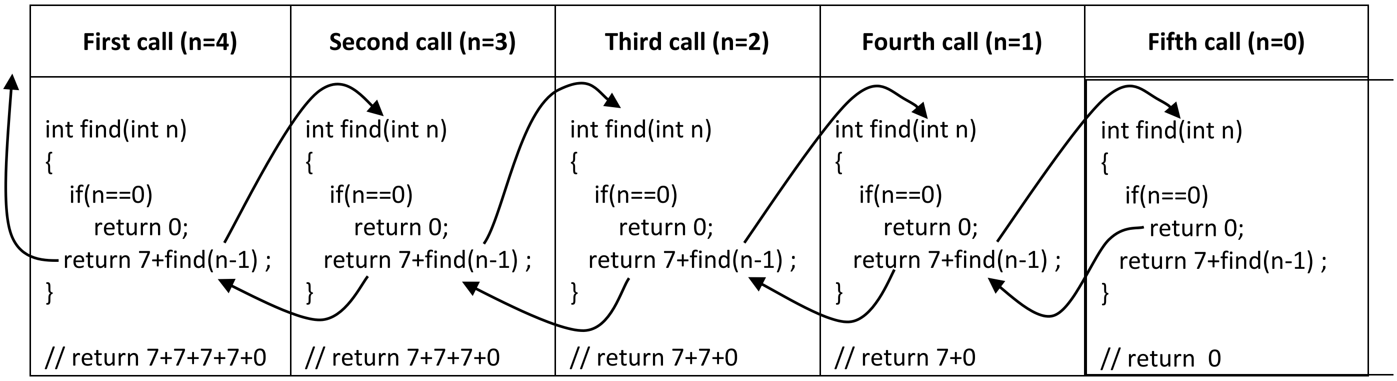
```
void main()
{       k = find( 4 , 0 );
        printf( " %d  ", k );
}
int find ( int  count , int  sum )
{          if( count==0 )
                return sum;
           sum=sum+7;
           return  find( count-1 ,  sum );
}
```

At 1st call	At 2nd call	At 3rd call	At 4th call	At 5th call
int sum=0	int sum=7	int sum=14	int sum=21	int sum=28
sum=sum+7	sum=sum+7	sum=sum+7	sum=sum+7	return sum;
= 0+7	= 7+7	=14+7	=21+7	// returns 28

here 'sum' is a parameter, the added 7 value is passed to next calls, at 1st call the sum=0, at 2nd call the sum=7, at 3rd call the sum=7+7, ….

Above program without parameter (simplified)

```
void main()
{          k=find( 4 );
           printf( " %d  ", k );    ?
}
int  find ( int  n  )
{          if( N==0 )  return 0;
           return  7 + find( n-1);
}
```



First call (n=4)	Second call (n=3)	Third call (n=2)	Fourth call (n=1)	Fifth call (n=0)

int find(int n)
{
if(n==0)
return 0;
return 7+find(n-1) ;
}
	int find(int n)
	{
	if(n==0)
	return 0;
	return 7+find(n-1) ;
	}
		int find(int n)
		{
		if(n==0)
		return 0;
		return 7+find(n-1) ;
		}
			int find(int n)
			{
			if(n==0)
			return 0;
			return 7+find(n-1) ;
			}
				int find(int n)
				{
				if(n==0)
				return 0;
				return 7+find(n-1) ;
				}

// return 7+7+7+7+0	// return 7+7+7+0	// return 7+7+0	// return 7+0	// return 0

Note: Myself, I solved many problems in recursion, but most of result calculating functions comes under above two logics only. this logic is better than above program logic.

Guess the output

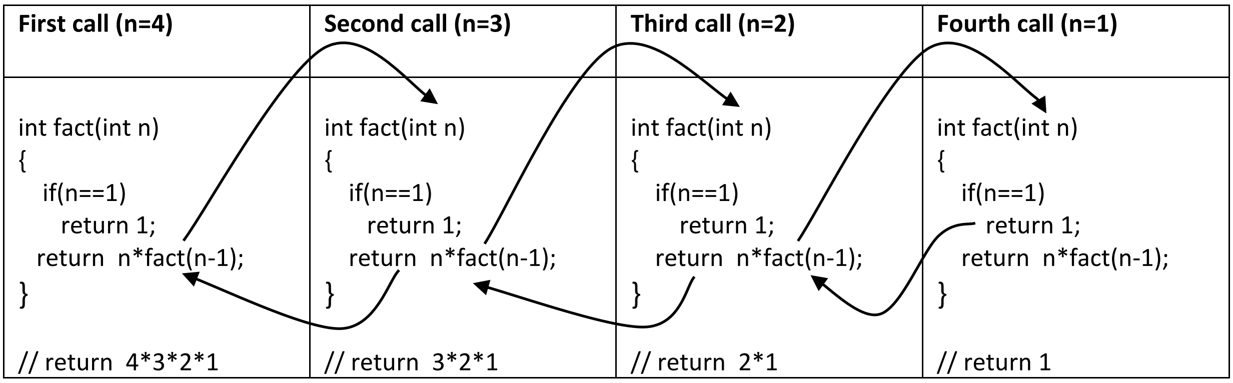
```
void main()
{          k=find( 345 );
           printf( " %d  ", k );     ?
}
int  find ( int  N  )
{          if( N==0 )  return 0;
           return  N%10 + find( N/10 );
}
```

Finding factorial of a given number using recursive function

The recursive definition of a factorial can be written in mathematical form as

```
f(n)=n*f(n-1)      if  n>1
f(n)=1             if  n=0 or 1
f(5)=4*f(3)
     4*3*f(2)
     4*3*2*f(1)
     4*3*2*1  =  24
```

```
int  fact(int n)
{      if(n==1)
           return 1;             // base condition
      return  n*fact(n-1);       // fact(n-1) is a recursive call
}
void  main()
{       int k;
       k=fact(4);
       printf("%d ",k);
}
```



First call (n=4)	Second call (n=3)	Third call (n=2)	Fourth call (n=1)

int fact(int n)
{
if(n==1)
return 1;
return n*fact(n-1);
}
// return 4*3*2*1
	int fact(int n)
	{
	if(n==1)
	return 1;
	return n*fact(n-1);
	}
	// return 3*2*1
		int fact(int n)
		{
		if(n==1)
		return 1;
		return n*fact(n-1);
		}
		// return 2*1
			int fact(int n)
			{
			if(n==1)
			return 1;
			return n*fact(n-1);
			}
			// return 1

- Finding factorial in iterative process (using loop) is simple & easy compared to recursion. This is just a demo program to explain the logic of how the recursion executes.

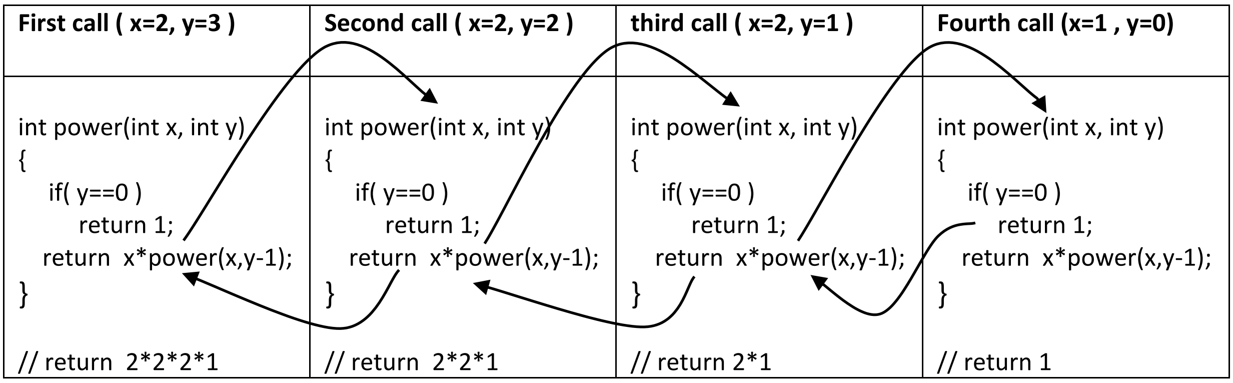
- The last call returns ‘1’ and it is substituted in its previous-call, the previous-call again returns ‘2*1’ and is substituted in its previous call. In this way, the final result 24 is returned to main() fn.

Finding x^y, where ‘x’ is base and ‘y’ is exponent

The recurrence relation is

```
fun(x,y)→  x*fun(x , y-1)      if y>1
fun(x,y) → 1                   if y==0
```

```
int power(int x, int y)
{         if(y==0)
               return 1;
          return x*power(x, y-1);
}
void main()
{         int a=2,b=3,c;
          c=power(a,b);
          printf("\n %d^%d=%d",a,b,c);
}
```



First call (x=2, y=3)	**Second call (x=2, y=2)**	**third call (x=2, y=1)**	**Fourth call (x=1 , y=0)**

int power(int x, int y)
{
if(y==0)
return 1;
return x*power(x,y-1);
}
 	
// return 2*2*2*1
	int power(int x, int y)
	{
	if(y==0)
	return 1;
	return x*power(x,y-1);
	}
	
	// return 2*2*1
		int power(int x, int y)
		{
		if(y==0)
		return 1;
		return x*power(x,y-1);
		}
		
		// return 2*1
			int power(int x, int y)
			{
			if(y==0)
			return 1;
			return x*power(x,y-1);
			}
			
			// return 1

Recursive function to find GCD. (Greatest Common Divisor)

This program finds the greatest common divisor of two numbers

Let the input values are x & y, where x<y. (The given below logic also works for x>y)

The recursive definition can be written in mathematical form as

f(x,y)=f(y%x , x) if y%x != 0

f(x,y)=x. if y%x == 0

The logic is, continuously divide the **y** with **x** until the remainder is zero.

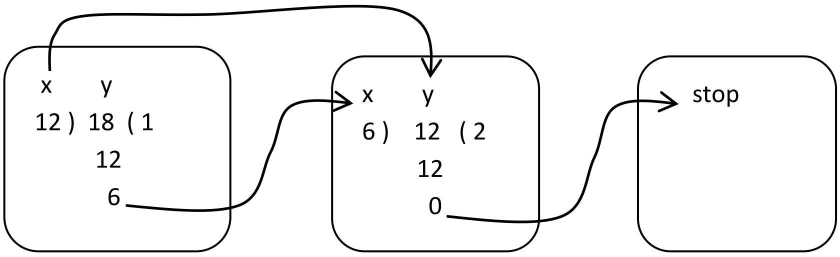
Here take **y** as **dividend** and **x** as **devisor.**

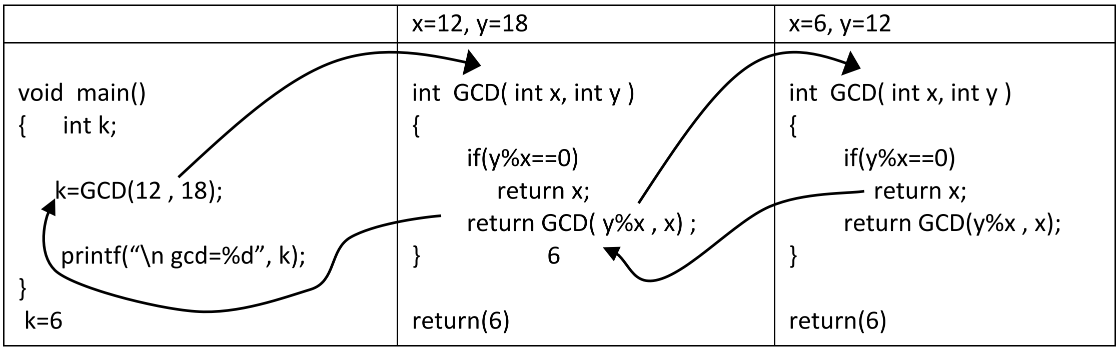
If **y%x** is zero then **x** will be the **GCD** and stop the process.

If **y%x** is not zero then take **x** value to **y** and **y%x** value to **x** for next cycle.

Do this process until **y%x==0**.

For example, the input values are 12 and 18 then the function and their calls are as





	x=12, y=18	x=6, y=12

void main()
{ int k;
 	
k=GCD(12 , 18);
 	
printf("\n gcd=%d", k);
}
k=6
	int GCD(int x, int y)
	{
	if(y%x==0)
	return x;
	return GCD(y%x , x) ;
	} 6
	
	return(6)
		int GCD(int x, int y)
		{
		if(y%x==0)
		return x;
		return GCD(y%x , x);
		}
		
		return(6)

Printing Fibonacci numbers up to n terms

Recursive function to print Fibonacci series up to user desirable limits.

```
void fibo( int count )
{        int x=0, y=1, new;          // here x, y, new are local variables to the function.
         if(count==0)  return;       // stop condition
          printf("%d  ", x);
          new=x+y;                   // generating next new term
          x=y;  y=new;               // moving x,y to next terms
          fibo(count-1);             // repeat until 'count' down to 0.
}
```

```
void main()
{         printf("enter how many no of terms to print:");
          scanf("%d", &n);
          fibo(n);
}
```

output: 0 0 0 0 0 0 0 0 0 0 0 0 0 0…… (Wrong output)

The above function produces wrong output. Because, for every recursive call, one fresh copy of local variables **x &y** are created and initialized with 0 and 1. (We already discussed many times)

To solve this problem take x,y as parameters and pass one-call updated values to next-call.

This is shown below

```
void  fibo(int x, int y, int count)
{         if(count==0) return;              // repeat until 'count' down to 0.
          printf("%d ", x );
          fibo( y, x+y, count-1);
}
void main()
{         int count;
          printf("enter how many no of terms to print:");
          scanf( "%d", &count );
          fibo( 0, 1, count );
 }
```

In this program, one may think that, only one copy of x & y is enough for all calls rather than creating a fresh copy in every call. So to avoid more copies, we can declare x, y as static/global. But this creates another problem! Once the static/global variable is created, it will not be killed even after function call ends; moreover they cannot be reinitialized for upcoming calls. Let us have one more example with global variable

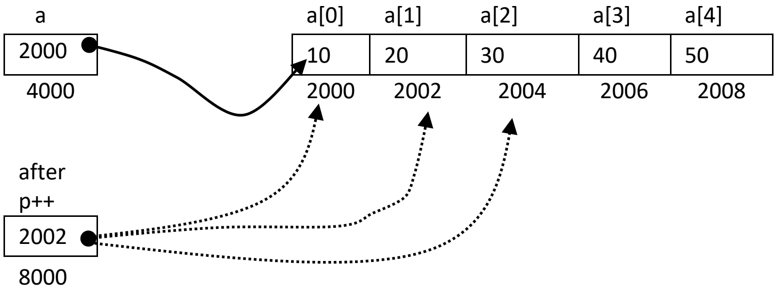
```
int k;
void main()
{         show();          //  at this call, output  is  1 2 3 4
          show();          //  at this call, no output
}
void show()
{         if(k==5)  return;
          printf("%d   ", k);
          k++;
          show();
}
```

op: 1 2 3 4

we expect output as 1 2 3 4   1 2 3 4 [as two times show() fn called from main() fn], as 'k' is global variable, only one copy is created and shared to all calls. After printing 1 2 3 4 at first call, the k attains 5 and will not be re-initialized to 1 for next calls. So no output is displayed for next calls. As per this experience, we don't recommend static and global variables in the recursion.

Guess the output of following program

```
void main()
{       int a[5]={ 10 , 20 , 30 , 40 , 50 } ;
        show( a , 5 );
}
void show( int *p ,  int  n )
{       if( n==0)  return;
        printf("%d  ",  *p );
        show( p+1 , n-1 );
}
```



Guess the output of following program

```
void main()
{       show( "Hello" );
}
void  show ( char *p  )
{       if( *p=='\0' )
            return;
        show( p+1 );
        printf("%c",  *p );
}
```

Guess the output of following program

```
void main()
{       show( "APPLE" );
}
void show( char *p  )
{       if( *p=='\0' )
            return;
        printf("%s\n",  p );
        show( p+1 );
}
```

Finding Nth Fibonacci number using recursion (double recursive calls)

Consider the first two terms are 0 & 1 and remaining terms generated by adding previous two values.

fibo(n) = fibo(n-1)+fibo(n-2). if n>2

fibo(n) = 0.   If n==1

fibo(n) =1.   If n==2

0	1	1	2	3	5	8	13	21	34	55	89	…
1	2	3	4	5	6	7	8	9	10	11	12	..

Fibo(5) → fibo(4)+fibo(3)

→ fibo(4-1)+fibo(4-2)+fibo(3-1)+fibo(3-2)

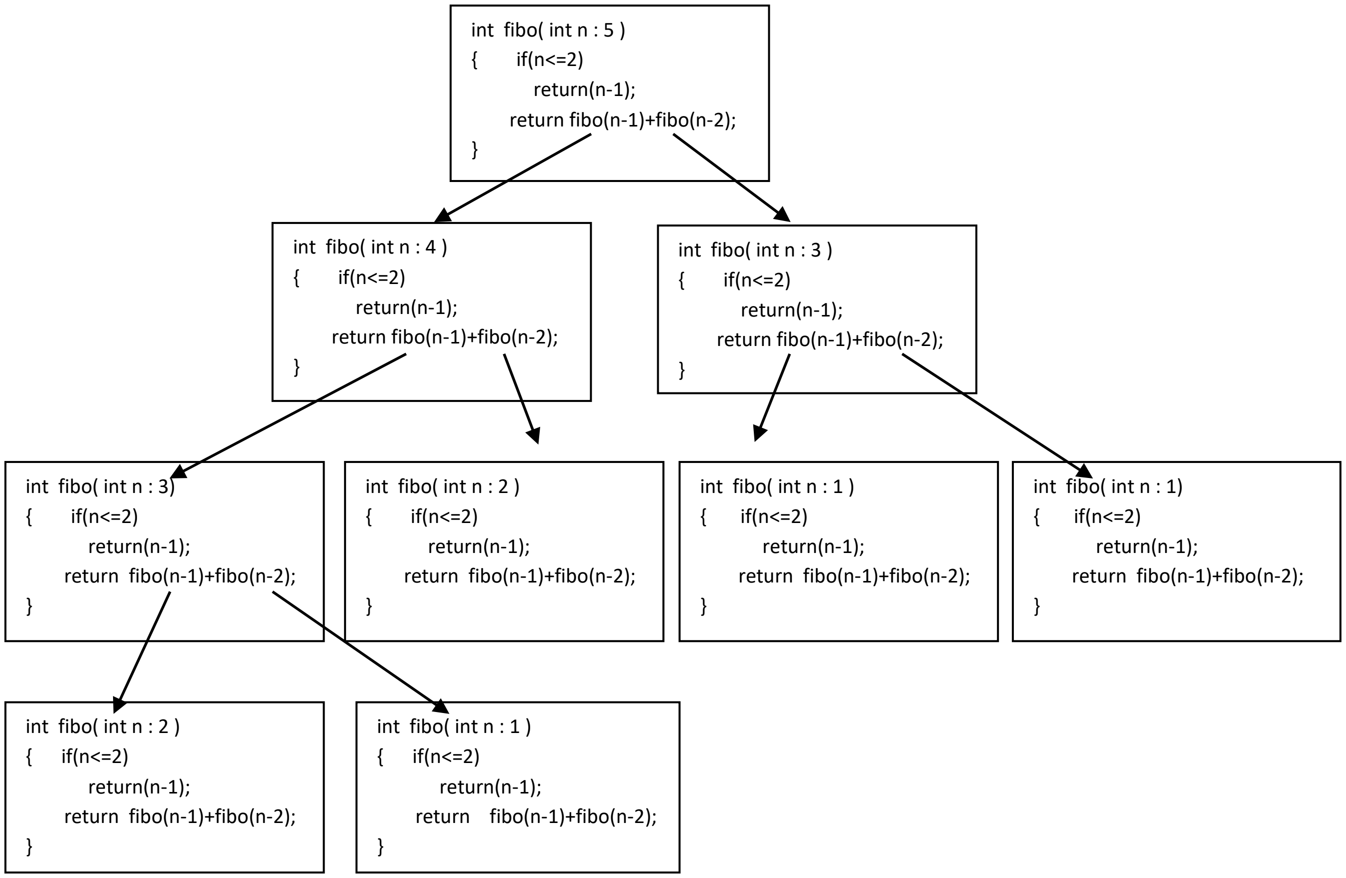
→ fibo(3-1)+fibo(3-2) + fibo(2)+fibo(2)+fibo(1)

→ fibo(2)+fibo(1)+fibo(2)+fibo(2)+fibo(1)

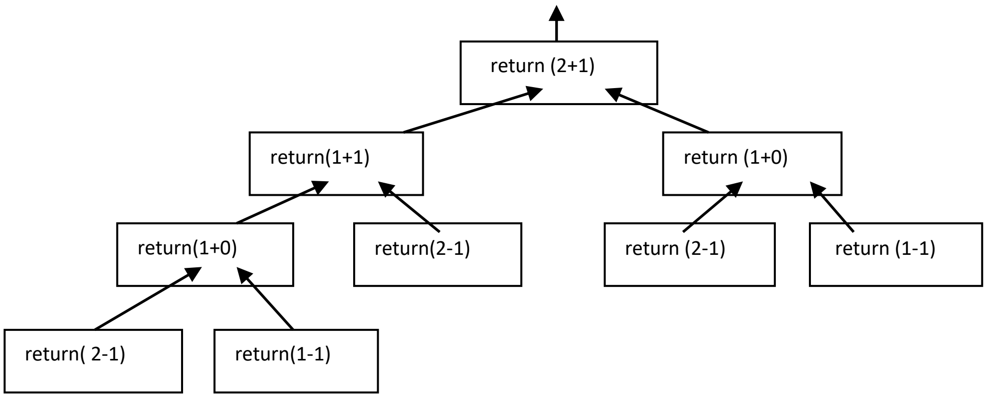
→1+0+1+1+0 → 3

```
void main()
{         int n=5;
          x=find(n);
          printf(“ the Nth term = %d”, x);
}
int fibo( int n)
{         if(n<=2)  return(n-1);
          return fib(n-1)+fibo(n-2);
}
```

Observe the following figure and try to understand how recursive calls are made.



While terminating function calls, the following statements are gets executed



Program to print all permutations of a given string

input: ABC output: ABC , ACB , BAC , BCA , CAB , CBA

```
int main()
{         char a[]= "ABC";
          permute(a, 0, strlen(a)-1 );
}
void permute(char a[ ] , int i, int len)
{         int j;
          if(i==len)
          {    puts(a);  return;
          }
          for(j=i;  j<=len;  j++)
          {    k=a[i]; a[i]=a[j]; a[j]=k;          // swapping a[i], a[j]
               permute(a, i+1, len);
               k=a[i]; a[i]=a[j]; a[j]=k;          // swapping a[i], a[j]
          }
}
```

Program to traverse entire chessboard with Knight (horse)

The following demo program is to traverse the entire chessboard with knight. Here Night visits every cell only once and it follows the rules of its movement ie, it moves only in L shape. This function takes x, y coordinates of first step of 8x8 board and displays order of movements in terms of step-number of every cell. Following figure gives an idea

		6		
		1		
	7		5	2
	4			
		8..	3	

```
int steps[8][2]={{-2,-1}, {-2,+1}, {-1,+2},{+1,+2}, {+2,+1}, {+2,-1},{+1,-2},{-1,-2}};
// this values are to choose all 8 possible L shape steps from the current standing step
int  a[8][8];               // chess board, where step movements are recorded.
#define bSIZE 5             // currently we are taking chess board size as 5
void main()
{         int x,y;
          printf("enter first step x, y values :");
          scanf("%d%d", &x, &y);
          chess(x,y,1);    // x,y are first step-step cell values, and 1 is first-step order
}
chess(int r, int c, int stepNo)
{         int i;
          if(r<0 || r>bSIZE-1 || c<0 || c>bSIZE-1 || a[r][c]!=0)   // if stepped out of chess board area
                    return;                                       or already such cell is entered then go back
          a[r][c]=stepNo;                      // making a cell as visited
          gotoxy(10+c*4, 10+r*2);              // printing in a particular position
          printf("%4d", stepNo);               // printing step number on the screen
          delay(1000);                         // delaying program to watch steps carefully
          if( kbhit()) exit(0);                //  if any key pressed program will be stopped
          if( stepNo==bSIZE*bSIZE) exit(0);    // if reaches all cells, then program stops
```

```
        for(i=0; i<8; i++)                      // stepping into next cell by choosing 8 possibilities
                chess( r+steps[i][0], c+steps[i][1], stepNo+1);
        a[r][c]=0;                              // if next choosing position is not good then go back
        gotoxy(10+c*4, 10+r*2);                 // removing step from screen
        printf("    ");
        delay(1000);
    }
```

Advantages and Disadvantages of recursion

Every recursive algorithm can be converted into non-recursive with the help of loop control and sometimes with stack. However, doing so, some programs may increase the complexity in code and prone to logical errors, and even it is difficult to debug the iterative code. So coding recursive nature problems, recursion is simpler than iterative process.

Recursive function slows the execution of program because of creation & destroying of local variables. Additional storage space also required to store function return-address. But writing code in recursion is easy, clear, and modify.

We know passing arguments from one to another function is common in modular programming. If any set of arguments need to be passed to a recursive function, then same set of arguments have to be passed for every call, even though they are unnecessary. In some cases, it is useful, however, in some other cases, it seems to be unnecessary, ie, one copy is enough for all calls. There is no alternative to overcome this problem. One might think that about usage of static or global variables, but once static variable creates, it will exist till the end of program. Second time if the same function is called it will not be reinitialized automatically. So using such variable is not suggestible.

Conclusion

If your recursive application needs to be used *frequently* which is considerably making unnecessary copy of local variables and if you are fit enough to write non-recursive code then it is most advisable to implement in non-recursive as it would be faster and takes lesser space.

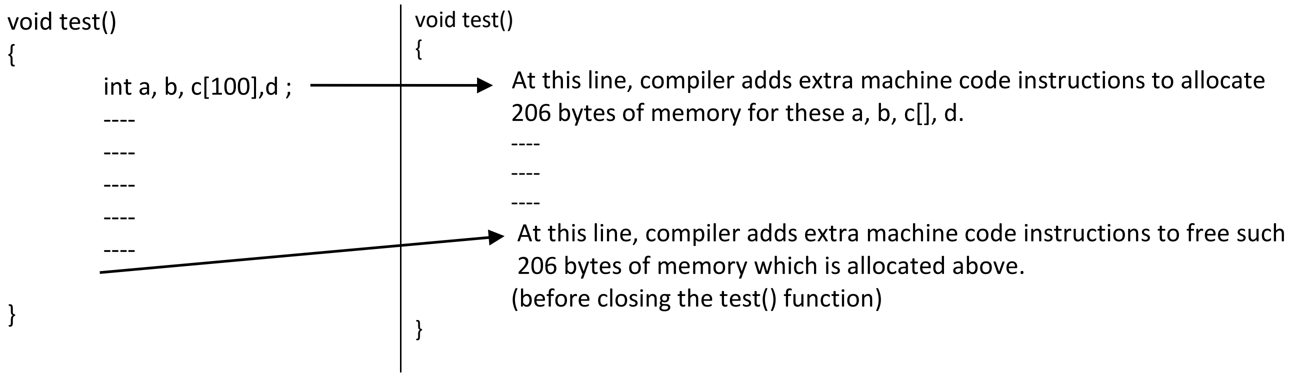
If an application is very complex, having rare usage, and taking less space for local variables, then it is better to implement recursive rather than non-recursive. Recursive is simple and easy to code
It is the programmer’s choice, which method is to be used when and where. If execution time and space are not constraints, then it is better to implement recursive application.
Implementation of recursion nature problem in non-recursive using loop and stack is an advanced concept. That topic is beyond the scope of this book.

Memory Allocation Systems

We have two memory allocation systems ① static allocation ② dynamic allocation
Allocating memory at compile time is said to be static allocation, this is created and managed by the compiler. Allocating memory at runtime based on input data is said to be dynamic allocation.

Static allocation system

if size of data is known at coding time then static allocation is the good choice. The static allocation system is managed by compiler by adding necessary machine code instructions in the program. The compiler adds extra instructions in our program for allocation/de-allocation of memory for our variables. So programmer need not to bother about how to create and destroy variable's memory in the program.
Let us see following example



```
void test()
{
        int a, b, c[100],d ;
        ----
        ----
        ----
        ----
        ----

}
```

```
void test()
{
  At this line, compiler adds extra machine code instructions to allocate
  206 bytes of memory for these a, b, c[], d.
  ----
  ----
  ----
  At this line, compiler adds extra machine code instructions to free such
  206 bytes of memory which is allocated above.
  (before closing the test() function)
}
```

Here compiler creates 206 bytes for **int a, b, c[100], d**. It adds instructions for allocation & de-allocation of memory as shown above. [It creates memory in stack as stack.push(206) and stack.pop(206)]
The static allocation system is much suitable when we know the size of data like employee-name(25 chars), address(50 chars), pin-code(long-int=4 bytes), phone-number(long-int 4 byte),…

Dynamic Memory Allocation system

in many real time systems, the size of data not known or cannot be expected until runtime. Here dynamic system is the choice. Dynamic allocation refers to allocation of memory during execution of program. Here programmer has to write special instructions to allocate & de-allocate memory with the help of pointers.

This DMA system provides us to create, expand, reduce, or even destroy the memory at runtime. This feature is very much useful for efficient memory management especially while handling huge collection of data using 'data structures' for example arrays, lists, stacks, queues, trees, graphs, etc. To work with this feature, C provides the following 4 built-in functions

1. malloc() // include alloc.h or malloc.h file to use these functions
2. calloc()
3. realloc()
4. free()

malloc()

It allocates user wanted size of memory and returns the starting byte address of allocated memory. This allocated memory is said to be **raw-memory** and we can store any type of data such as int, float, char, etc. For this reason, malloc() returns this memory address as generic type (no specific type). That is, it returns address as **"void*"** type and it is often called generic address. Unfortunately, if memory is not available then malloc() returns NULL value.

```
prototype:   void*  malloc( unsigned int );
   syntax:   pointer= malloc( no of bytes to be allocated );
  example:   p=malloc( 10 );   // creates 10 bytes of memory and returns starting address to 'p'
```

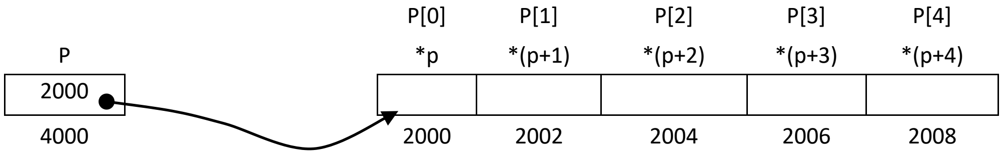
In the above example, the function ‘malloc()’ allocates 10 bytes of memory in the RAM and returns the starting byte address, which is then assigned to 'p'. Here one question arises! What type of pointer ‘p’ should be? This is fairly depends upon what type of data which we are going to keep in this memory. Suppose if you are going to store array of float values then pointer should be “**float*”** type. Remember that, malloc() returns just address which has no specific type, that is, it returns **“void*”** type address. So it should be **type-casted** as per our pointer type. (All these functions return-type is **void*** type only)

Example: Allocating memory for ‘n’ integers

```
int   n, *p;
scanf(“%d”, &n);
p= (int *)malloc( n*sizeof(int) );
```

Converting **void*** to **int***

The above allocated memory maps to the pointer ‘p’ is as follows:



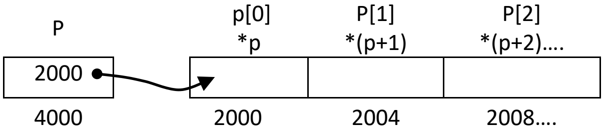
How to access dynamic memory?

Accessing this memory is same as accessing array elements through pointer.
Observe the following expressions

*(p+0) → p[0] accesses the first location (first 2bytes)
*(p+1) → p[1] accesses the second location (second 2bytes)
*(p+2) → p[2] accesses the third location
*(p+ i) → p[i] accesses the i+1th location.

The link between arrays and pointers already discussed in pointer chapter, please refer it.
If you are creating memory for floating point values, then **float*** type pointer is required.
For example creating memory for **‘N’** floats. Here ‘N’ is the input value

```
float  *p;
scanf(“%d”, &N);
p=(float *)malloc( N * sizeof(float) );
```



Demo program, accepting ‘N’ integers and printing

```
#include<alloc.h>
void main()
{         int n, i, *p;
          printf(“enter no.of input values:”);
          scanf(“%d”, &n);
          p=(int*) malloc( sizeof(int) * n );
          if(p==NULL)
          {         printf(“\n error, memory not created");
                    return;
          }
          for(i=0; i<n; i++)
          {         printf( "enter any value :");
                    scanf( “%d", &p[i] );
          }
```

```
            for(i=0; i<n; i++)
                    printf("%d  ", p[i] );
    }
```

In the above, the malloc() creates memory for n-integers, later accessing through the pointer **'p'**. Here the input size can be anything, thus it makes efficient usage of memory.

calloc()

It is also like malloc(), but after allocation of memory, it clears the garbage values by filling with zeros in all bytes of memory.

pointer = calloc (no.of items, size of item);

It takes two arguments, first one is number of items to be allocated and second one is size of item. For example, allocating memory for 5 floats

```
float *p;
p = (float *) calloc(5, sizeof(float));
```

realloc()

It is used to resize (reduce or expand) the memory which is already allocated using malloc() or calloc() or realloc(). It takes previously allocated memory address and adjusts to new size.

syntax: **pointer = realloc(oldAddress, newSize);**

For example

```
int *p,
p=(int*)malloc(10*sizeof(int));
----
p=(int*)realloc(p , 20*sizeof(int));     // increasing size 10 integers to 20  (10 integer's space added)
----
p=(int*)realloc(p , 5*sizeof(int));      // reducing size 20 integers to 5    (15 integer's space reduced)
----
```

Note: p=(int*)realloc(NULL, 10*sizeof(int)) // here **oldAddress** is null, so it creates fresh memory like malloc() function.

While expanding the size of old memory, if required amount of memory is not available at the adjacent locations, then it allocates in new location and copies the old memory contents to new location and then releases the old memory. If the memory is not available, then it returns null.

free()

It is used to release the memory which is no longer required by the program. This memory should not belongs to static allocated system like **"int a[10]"**. This must be a dynamic memory which is allocated by malloc() or other functions. It takes the base address of memory as an argument and frees it.

```
Prototype:  void  free(void *);
   Syntax:  free(address);
  Example:  int *p;
            p=(int *)malloc(50*sizeof(int));
             ---
             ---
            free(p);
```

Prone and cons of systems

* Program's data area divides into **stack & heap** area. In stack area, the temporary local variables and function-call return addresses are stored, whereas in heap area, permanent variables are stored.
For DMA system a special block of memory is created in the RAM and managed as heap.
* In static allocation system, at coding time (before compilation), the size of array must be given with constant value like **"int a[100]"** and it cannot be changed at run time.
* Static allocation system is faster than dynamic, because allocation of memory in the stack is faster than heap. In static system, at compile time, the relative memory address is given to all variables, so it will be faster. Whereas in DMA system, at run time, the memory is allocated by searching a suitable location and also maintains a separate table for registering address which location is allocated and which is not, this makes the program slow. If size is known then static is the good choice otherwise dynamic is the choice.

Program accepts N strings (country names) and prints them

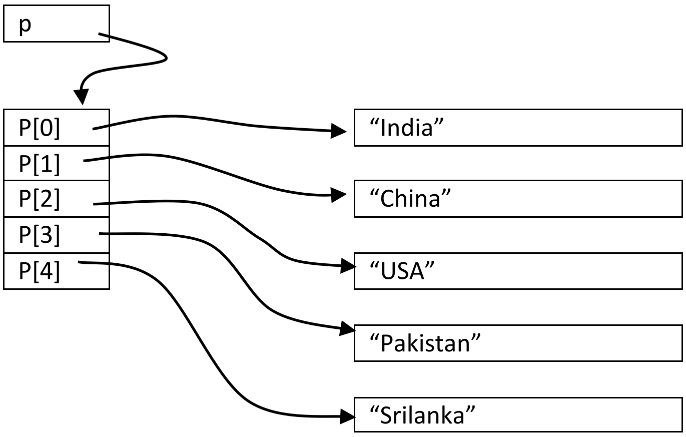
This program accepts strings one by one until user entered termination string as **"end".**
Later displays all such inputted strings on the screen.

```
#include<stdio.h>
#include<string.h>
#include<alloc.h>
void main()
{       char  **p=NULL;
        int count=0,i;
        char a[50];
        while(1)
        {       printf("enter country name :");
                fflush(stdin);
                gets(a);
                if( strcmpi( a , "end" ) ==0 ) break;
                p=(char**) realloc(p,sizeof(char*) * (count+1) );
                p[count] = (char*)malloc(sizeof(char) * (strlen(a)+1));
                strcpy( p[count] , a );
                count++;
        }
        printf(" U r entered \n");
        for(i=0; i<count; i++)
                puts( p[i] );
}
```

If input strings are

India
China
USA…
end

For the above program, the memory creates for the pointer as →



Command Line Arguments

Just like other functions, main() also receive the arguments and returns a value. Here few questions may rise! From where does the main() function is called and passed the arguments? The main() function is called from Operating System's command line by just typing the program name as shown below.

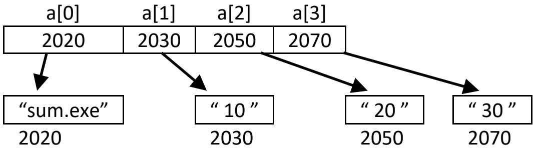
C:\tc\bin\>sum.exe // here ".exe" is optional
(Suppose if our C program's file name is "sum.c" then its compiled version will be "sum.exe")

When a program is run, the operating system loads **.exe** file into memory and invokes the startup function main(). Here, we can assume the OS is the caller of main() function. Therefore, the arguments can be passed to the main() by specifying a list of values at the command line as shown below.

C:\tc\bin\>sum 10 20 30
Unlike other functions, the arguments to main() passes in a different manner. They pass as array of string constants. That means, even if you give an integer values, they pass as strings. In the above case also, the arguments will be passed as "sum", "10", "20", "30". (Passes as array of addresses as shown in below picture).

But main() receives them as two arguments, the first argument is integer, which specifies the count of arguments which we passed including file-name. The second argument is, an array of addresses containing strings. Thus main() function should be defined as

```
void main(int count , char *a [])
{       ---
        ---
}
```



Here, 'count' holds the count of arguments whereas 'a[]' holds address of all string constants that are passed to the program. The size of array is, number of values we passed to it. (This as above figure)

Example1 : this program adds all such integers which we passed through command line as above shown

Input: c:\tc\bin\> sum 10 20 30 40

Output: sum of numbers = 100

```
void main( int count, char *a[ ] )
{       int sum=0,i=0;
        for(i=1; i<count; i++)
                sum = sum+atoi(a[i]);        //atoi() function converts numeric string to integer value
        printf("\n sum of numbers = %d", sum);
}
```

Example2: The following program takes string as input from command line and displays length.
It also shows an error, if user enters less number of strings.

Input: C:\tc\bin\>sample C-Family

Output: length of C-Family = 8

```
void main( int count, char *a[] )
{       if( count==1)
        {       printf("insufficient arguments");
                return;
        }
        prinf("the length of %s = %d", a[1], strlen(a[1]) );
}
```

User Defined Data Types

The data types can be classified into two types ①**predefined types** ②**user defined types**.
Predefined types are also called built-in or primitive types, whereas user types are custom types.

Primitive Data types: these are also called built-in or basic types, for example int, char, float, int*, etc; these primitive types and their functionalities were already designed and developed in compiler level during development of C software. Therefore, we can directly use them in programs.

User defined Data types: 'C' gives a freedom to the coder to define his own data types according to requirement in the program. Therefore, these are called user-defined types.
We have 3 types, ①**structure** ②**union** ③**enumerator.**

Generally, it is difficult to handle large collection of data using basic types. Using arrays, we can handle only collection of homogeneous items of known size. However, to handle collection of heterogeneous items then user-types are required. For example, consider an employee data containing id, name, address, salary, etc. Handling such heterogeneous items individually is difficult and leads to complex task. If we group all these items into single (as one record) then they can be handled easily. For this, fortunately, C is providing a structures concept.

1) Structures

struct structure-name variable1, variable2,…,

Structure is a collection-type user-defin as single item. So we often state that, array is collection of homogenous items, whereas structure is a collection of heterogeneous data items. Sometimes, the items in a structure can be same type.
The items in a structure are called **data-members** and we can have as many members as we want.

Syntax to declare a structure

```
struct  structure-name
{       data_memberl;
        data_member2;
        …
        data_memberN;
};
```

For example:

```
struct student
{        int idno;
         char name[30];
         int marks1, marks2;
};
```

This structure declaration defines or creates a new data-type called **"struct student"**, using this, we can create variables and its data can be managed.

This structure declaration works as a blue-print or template of our type and it does not occupy any space in executable file. Just it gives layout idea to the compiler i.e, about structure-name, data type of each member and space required for each member. Now using this structure template, we can create variables and its data can be managed. Here structure-name is the name given to the structure and it follow rules of variable naming. The structure can be declared in local or global scope, depends upon requirement of program.

Syntax to declare structure variables:

For example: __**struct student**__ __**K , S ;**__
Data type Variables

The memory allocation for the 'K' and 'S' will be as

	K				**S**			
Member's name →	idno	Name	marks1	marks2	idno	name	marks1	marks2
Bytes occupied →	2	30	2	2	2	30	2	2

Many people says, the declaration of structure works as logical idea like building plan on paper whereas the variables of structure works as physical space like constructed building. (logical vs physical or plan vs implementation)

Accessing members in a structure: we have two operators

1. Selection operator, dot (.) // accessing through structure variable
2. Indirect selection operator, arrow (→) // accessing through structure pointer

a) Using selection operator

syntax: struct_variable **.** member_name

for eg: the expression **K.idno** accesses the 'idno' in **K**

the expression **K.name** accesses the 'name' in **K**

the expression **K.marks1** accesses the 'marks1' in **K**

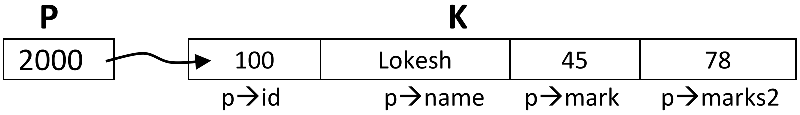
printf(" %d %s %d %d ", K.idno, K.name, K.marks1, K.marks2);

K			
100	"Lokesh"	45	78
k.idno	k.name	k.mark1	k.marks2

b) Using indirect selection operator

if structure type is '**struct student'**, then its pointer type is ' **struct student* '**

```
for example: struct student  *p;         // 'p' is a pointer to a structure
                struct student  K;
                 p=&K;                   // making pointer 'p' to 'K'
```



now the expression '***p'** accesses the total memory of 'K' [so *p == K]

the expression (*p).idno accesses the 'idno' in 'K'

(*p).idno is equal to K.idno

(*p).idno is equal to p->idno // this 'p->idno' is a short form, this is called: p arrow idno)

(*p).name or p->name is equal to 'K**.name'**

The arrow operator (->) introduced in second version of C language
this is a short-cut operator, used to access the members in a structure through pointer.

> Remember: **(*p).name** is equal to **p->name ,** so the (***p)** can be taken as **p->**

Note: the expression **(*p).name** <u>cannot be taken</u> as ***p.name**, because, the dot operator(.) has high priority than pointer operator(*) , here first the compiler tries to evaluate **p.name** thereby shows an error. Because 'p' does not have members of a structure, actually it contained address. The expression (*p).name is valid expression because it is 'K.name'

Initialization VS assignment of structure variables

A) Initialization of structure is same as initialization of array, here all values must be put in pair of braces{}

```
          syntax:   struct variable = {list of structure values};
        example:  struct student  K={100,"Lokesh",33,44},   S={101, ”Srihari”,55, 66};
```

K

100	Lokesh	33	44

S

101	Srihari	55	66

B) Assignment of structure members is same as assigning values to basic data- type variables, for example

```
                K.idno=100;                         // filling idno number with 100
                K~~.name=”Srihari”~~ ;                  // we can’t assign like this, because ‘name’ is arrays’s pointer
                strcpy(K.name, “Srihari”);          // filling name with “Srihari”
                K.m1=66;
                K.m2=77;                            // filling marks with 66,77
```

Demo program filling and displaying structure member values

Let student contained values like idno, name, and two subject marks as said in above example and printing all details. (Here each member assigned with a value instead of initializing the whole structure)

```
        struct student                              // this is global declaration of a structure
        {       int idno;
                char name[30];
                int  m1, m2;
        };
        void main()
        {       struct student    s;
                s.idno=100;                         // filling idno number with 100
                strcpy(s.name, “Srihari”);          // filling name with “Srihari”
                s.m1=66;   s.m2=77;                 // filling marks with 66,77
                printf("the filled details are:");
                printf( "%d  %s  %d  %d “ , s1.id, s1.name, s1.m1, s1.m2 );
        }
```

The following program accepts student details from keyboard and printing on the screen.

```
        void main()
        {       struct student    k;                //declaration of student variable
                printf("\nEnter student details:");  // scanning details
                printff \n enter idno :");
                scanf("%d", &k.idno);
                printf("enter name :");
                fflush(stdin);
                gets(k.name);
                printf(“enter marks1  &  mark2:");
                scanf("%d%d",&k.m1, &k.m2);
                printf("\n ---------------the scanned details are:--------------------------\n");
                printf("%d  %s  %d  %d  %d", k.idno,  k.name,  k.m1,  k.m2);
        }
```

like normal variables, the structure members are scanned individually as built-in type.

The total structure cannot be read as a single value unless a special function is written for it.

Let us see advantages of structures

taking two student’s data variables with and without structures

```
void main()
{       // declaration of two student’s variables without structures
        int  idno1 ,  idno2;
        char  name1[30], name2[30];
        int   m11, m12, m21, m22;
        ----
}
```

This is declaration of 2 student’s variables this seems to be difficult to understand

If we use structures then the code will be

```
void main()
{       // declaration of two student’s variables with structures.
        struct student  s1,s2;
        ----
}
```

This seems to be so easier than the above. Thus structures provide a great convenience.

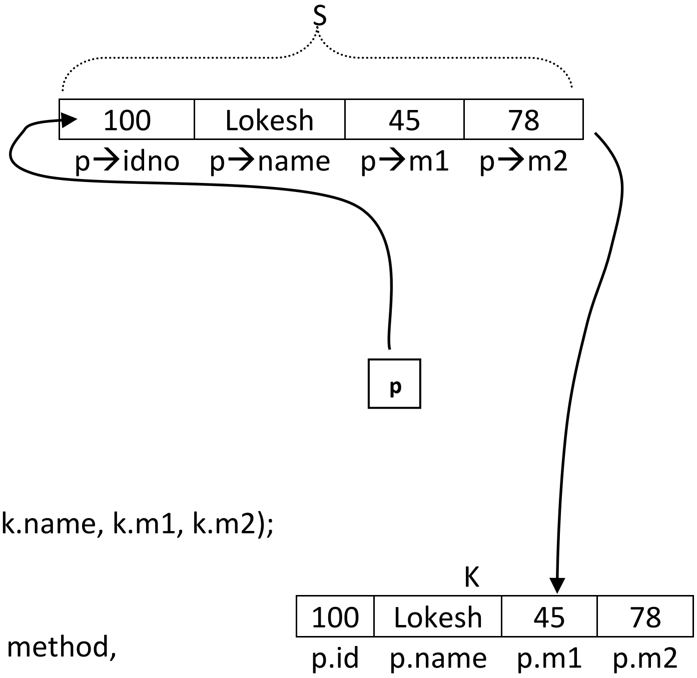
Filling and displaying student details through functions

Following two functions fill() & show() manipulates the student data.

The fill() function, fills the student details with the data: **“100, Lokesh, 45, 78”** and show() fn displays it.

The fill() function followed **call-by-reference**, whereas, show() function followed **call-by-value.**

```
struct student
{       int idno;
        char name[30];
        int m1, m2;
};
void main()
{       struct student  s ;
        fill(&s);
        show(s);
}
void  fill(struct student *p)
{       p->idno=100;
        strcpy( p->name, ”Lokesh“);
        p->m1=45;
        p->m2=78;
}
void display( struct student  k)
{
        printf(“ %d  %s  %d  %d ”, k.idno, k.name, k.m1, k.m2);
}
```



The function fill() has followed call-by-reference method, here the pointer ‘p’ is pointing to structure ‘s’ as shown in figure. The expressions p->id, p->name, p->m1 accesses the memory of structure ‘s’.

That is, the fill() function indirectly inserts the values into members of ‘s’;

Whereas, the function show() has followed call-by-value method. When it calls, all the members data of ‘s’ are copied into ‘k’, later displayed one by one using ‘k’.

Scanning two dates from KB and finding whether they are equal or not?

* trying with & without using functions

```
 ip:  12  9  2010                    ip:  12  9  2010
      12  9  2010                         13  9  2011
op:  equal                           op: not equal

struct Date
{        int d, m, y;
};
void main()
{        struct Date  a , b;
         int k;
         read( &a );                 // scan date1 (day, month, year) using read() function.
         read( &b );                 // scan date2 (day, month, year)
         k=compare( a , b );         //  compares two dates and returns 1/0
         if( k==1)  printf("equal");
         else  printf("not equal");
}
void read( struct Date *p )
{        printf("enter day  month  year:");
         scanf("%d%d%d", &p->d, &p->m, &p->y);
}
int compare( struct Date p ,  struct Date q )
{        if( p.d==q.d  &&  p.m==q.m  && p.y==q.y )
                  return 1;
         else
                 return 0;
}
```

the read() fn followed call-by-reference, because, the scanned values at read() fn need to be returned to main() fn, therefore we have to store indirectly through pointers, whereas compare() fn follows call-by-value, it is just comparing given two dates.

Scanning two times from keyboard and printing addition of them

Let the two times are employee working time in two shifts, later add & print total time worked in two shifts.

```
ip:  12   50   58            (shift-1 worked duration  :  Hours=12, Min=50, Seconds=58)
      2   53   55            (shift-2 worked duration  :  H=02,  M=53, S=55)
op: 15hr  44min   53sec      (total duration worked in two shifts)

struct Time
{        int h, m, s ;
};
void main()
{        struct Time a, b, c;
         a=read();             // this read() fn returns the scanned time ( it is opposite to above read() fn)
         b=read()
         c=add(a,b);           //  add two times like  c=a+b
         write(c);             // print output time on the screen
}
```

```
struct Time read()
{         struct Time k;
          printf("enter time :");
          scanf("%d%d%d", &k.h, &k.m, &k.s);
          return k;   // returning scanned time(three values)
 }
struct Time add(struct Time  a, struct Time b)
 {        long int  k;
          struct Time  t;
          k=(a.h+b.h)*3600 + (a.m+b.m)*60 + (a.s+b.s);
          t.h=k/3600;
          t.m=(k%3600)/60;
          t.s=(k%3600)%60;
          return t;
}
void write( struct Time k)
{         printf("output time is : %d - %d - %d", k.h, k.m, k.y);
}
```

Above program followed completely call-by-value method, but we can write same program using call-by-ref, the following example explains it

```
void main()
{         struct Time a, b, c;
          read(&a);
          read(&b);
          add(&c, &a, &b);
          write(&c);
}
void read( struct Time *p)
{         printf("enter time :");
          scanf("%d%d%d", &p->h, &p->m, &p->s);
}
void  add(struct Time  *t,  struct Time *p,  struct Time *q)
{         long int k;
          k=(p->h+q->h)*3600 + (p->m+q->m)*60 + (p->s+q->s);
          t->h=k/3600;
          t->m=(k%3600)/60;
          t->s=(k%3600)%60;
}
void write( struct Time *p)
{         printf("output time is : %d - %d - %d", p->h, p->m, p->s);
}
```

Conclusion: here read() fn should be written in call-by-ref, but add() and write() can be either of two.
if structure size < 20 bytes, then call-by-value is the choice where the members can be easily accessed.
if structure size > 20 bytes, then call-by-ref is the choice, because it is unnecessary creating 2nd copy at called function for a big size data. Here in this case, for the two functions add() & write(), the call-by-value is the best, because the size of structure is less than 20 bytes..

Scanning two matrices data from keyboard(KB) and printing addition of them

```
struct  matrix
{         int a[10][10];                      // array to hold matrix elements
          int  r,c;                           // r, c  are  order  of  matrix
};
 void main()
{         struct matrix  x, y, z;
          int  k;
          accept(&x);               // call-by-reference
          accept(&y);
          k=add(&z, x, y);          // partly call-by-value and call-by-reference
          if(k==0) printf("\n Matrix cannot be added ");
          else  display(z);         // call by value
}
void accept( struct matrix  *p )
{         int i,j;
          printf("Enter no.of rows and columns :");
          scanf("%d%d", &p->r, &p->c );
          for(i=0;  i<p->r;  i++)
          {         for(j=0;  j<p->c;  j++)
                    {         printf("enter value of [%d][%d]:", i, j);
                              scanf("%d", &p->a[i][j]);
                    }
          }
}
void display( struct matrix z)
 {        int i,j;
          for(i=0; i<z.r;  i++)
          {         for(j=0;  j<z.c; j++)
                              printf("%d  ", z.a[i][j]);
                    printf("\n");
          }
}
int add(struct matrix *p, struct matrix x, struct matrix y)
{         int i,j;
          if(x.r != y.r | | x.c != y.c)
                    return 0;                 // zero indicates addition failed
          for(i=0; i<x.r;  i++)
                    for( j=0; j<x.c; j++)
                              p-> a[i][j] = x.a[i][j] + y.a[i][j];
          p->r=x.r;
          p->c=x.c;
          return 1;                           // return ‘1’ indicates addition succeeded
}
```

Array of structures

Array of structures is useful when several instances of details are to be handled in the program.
We can create array of structures just like any normal data types.
Syntax: struct structure-name array-name[size];

For example:
```
struct Person
{         char  name[20];   // name of person
          int age           // age of person
};
struct Person   s[3];       // creates array of 3 structure variables.
```

The array of 3 structures is as follows

s[0]		s[1]		s[2]	
name	age	name	age	name	age

Like normal arrays, the structure arrays are accessed, here each element s[0], s[1], s[2] is a structure item.
Syntax to access members in the structure array is: **array-name[index].member-name**

The expression s[0].name → accesses the name in s[0]
The expression s[0].age → accesses the age in s[0]
The expression s[1].name → accesses the name in s[1]
The expression s[1].age → accesses the age in s[1]

Following program explains how to scan & print array of 3 structure values.

```
void main()
{         struct  Person  s[3];
          scan(s);                 // scan(&s[0] ) → passing  base  address of array
          display(s);              // display(&s[0]);  → passing  base  address of array
}
void scan(struct person  p[]  or struct person   *p)
{         int i;
          for( i=0; i<3; i++)
          {         printf("enter name & age of a person :");
                    gets(&p[i].name);              // gets( (p+i)->name );
                    scanf("%d", &p[i].age);        // scanf("%d",  &(p+i)->age );
          }
(or)      for( i=0; i<3; i++)
          {         printf("enter name & age of a person :");
                    gets(&p->name);
                    scanf("%d", &p->age);
                    p++;                           // increments by 22, to points to next structure in the array
          }
}
void display ( struct person  p[]  or struct person  *p )
{         int   i;
          for( i=0; i<3; i++)
          {      printf("\n %s    %d ", p[i].name, p[i].age);   //or printf("\n %s    %d ", (p+i)->name, (p+i)->age);
          }
 (or)     for( i=0; i<3; i++)
          {         printf("\n %s    %d ", p->name,  p->age);
                    p++;             // increments by 22, to points to next structure in the array
          }
}
```

above two loops do same job in scan() & display() functions, but second loop little faster than first loop.

Scanning N student details and printing with result.

```
input:   enter no.of  students: 3
         enter student idno  name    mark1    mark2
                         101   Srihari      70        80
                         102   Narahari   90        90
                         103   Murahari   50        50

output:   idno   name      mark1    mark2   total  average
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
          101    Srihari     70         80       150     75
          102   Narahari    90         90       180     90
          103   Murahari   50         50       100     50
```

```
struct student
{       int id;
        char name[30];
        int m1, m2, total, avg;
};
void main()
{       struct student  s[100];
        scan(s, &n);                    // scan( &s[0], &n );
        find(s, n);                     // find( &s[0], n );
        display(s,n);                   // display( &s[0], n );
}
void scan( struct student  s[] , int  *pn)
{       int n,i;
        printf("enter no.of students :");
        scanf("%d", &n);
        for(i=0; i<n; i++)
        {       printf("enter student idno  name  mark1   marks2:");
                scanf("%d%s%d%d%d", &s[i].idno, &s[i].name, &s[i].m1, &s[i].m2);
        }
        *pn=n;          // no.of students
}
void find( struct student  s[] , int n )
{       int i;
        for(i=0; i<n; i++)
        {       s[i].total=s[i].m1+s[i].m2;
                s[i].avg=s[i].total/2;
        }
}
void display(struct student s[] , int n)
{       int i;
        printf("output is : idno     name    marks1    marks2    total   average\n");
        printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
        for(i=0; i<n; i++)
                printf("\n %d   %s  %d   %d   %d   %d", s[i].idno,  s[i].name, s[i].m1, s[i].m2,    \
                        s[i].total, s[i].avg);
}
```

Adding two polynomials using array of structures

$5x^7+14x^5+3x^2+10x^1+18$

$15x^4+10x^3+13x^2+7$

The two expressions are stored in array of structures as given below

p[0]		p[1]		p[2]		p[3]		P[4]	
5	7	14	5	3	2	10	1	18	0
coeff	exp	coeff	exp	coeff	exp	coeff	exp	coeff	exp

p[0]		p[1]		p[2]		p[3]	
15	4	10	3	13	2	7	0
Coeff	exp	coeff	exp	coeff	exp	coeff	exp

```
#define maxSize 20                      // let maximum size of degree is 20
struct  POLY
{       int  exp;
        int  coeff;
} ;
void  readPoly( struct POLY p[] )
{       int prev=maxSize+1, i=0;
        while(1)
        {       printf("\nEnter coefficient (use 0 to exit) :");
                scanf("%d", &p[i].coeff );
                if(p[i].coeff==0) break;
                printf("\nEnter exponent: ");
                scanf("%d",&p[i].exp);
// the input exponents must be in sorted order like shown in above equation, otherwise shows input error
                if( p[i].exp > prev)
                {       printf("input error");
                        continue;               // go back and scan again
                }
                prev=p[i].exp;
                i++;
        }
}
void printPoly(struct POLY p[])
{       int i;
        for(i=0; p[i].coeff!=0; i++)
                printf("%d^%d + ",  p[i].coeff,  p[i].exp);
        printf("\n ");
}
void addPoly( struct POLY p3[], struct POLY p1[], struct POLY p2[])
{       int  i=0, j=0, k=0;
        while (p1[i].coeff!=0 && p2[j].coeff !=0 )
        {       if( p1[i].exp > p2[j].exp)
                        p3[k++] = p1[i++];
                else  if( p1[i].exp < p2[j].exp)
                        p3[k++] = p2[j++];
                else
                {       p3[k].exp = p1[i].exp;
                        p3[k++].coeff = p1[i++].coeff + p2[j++].coeff;
                }
        }
        while(p1[i].coeff!=0 )  p3[k++] = p1[i++];
        while(p2[j].coeff!=0)   p3[k++] = p2[j++];
        p3[k].coeff = 0;
}
void main()
{       struct POLY  p1[maxSize],  p2[maxSize],  p3[maxSize];
        printf("enter polynomial 1:");
        readPoly(p1);           // accepting first polynomial
        printf("enter polynomial 2:");
        readPoly(p2);           // accepting second polynomial
        addPoly(p3,p1,p2);
        printf("\n The resultant poly after addition is\n");
        printPoly(p3);
}
```

More about different declarations of structures

We can declare structure variables while defining the structure itself. However, if we define a structure in global scope then all variables also becomes as global.

```
struct Account
{       int  accNo;
        char name[30];
        float balance;
} a1, a2;           → Here a1, a2  are variable of type “struct Account”;
```

Above example with Initialization of variables

```
struct BankAccount
{       int  accNo;
        char name[30];
        float balance;
} a1={ 101, "James Bond", 2200.45}, a2={102, "Goshling”,  3400.45};
```

Structure name can be ignored while declaration of structure

```
struct              // structure  name omitted
{       int  accNo;
        char name[30];
        float balance;
} a1, a2;
```

In this method, we cannot declare variables of same structure in the rest of the program, because “structure name” is not available to refer the structure in the future. Actually, this type of declaration is used for sub categories of data. This is also known as nested structures, for example

```
struct  Account
{       int  accNo;
        char name[30];
        float balance;
        struct                          // called nested structure
        {       int  day,month,year;
        }openDate;
};
```

Nested structures

Structure in structure is said to be nested structure, we can nested as many times as we want, complexity will not be increased even though nested many times. For example,

```
struct Student
{         int idno;
          struct Date
          {         int d,m,y;
          } joinDate;
};
void main()
{         struct Student  p;
          p.idno=10;
          p.joinDate.d=10;
          p.joinDate.m=12;
          p.joinDate.y=2020;
          -----
}
```

P			
Idno	joinDate		
	d	m	y

typedef

"typdef" is a keyword,used to define new convenient names for existing data types.
The new-name works as alias-name for existing data types, but we don't lose the old- name.
Syntax: **typedef existing-name new-name;**

```
typedef  float  RS;
typedef  float  DOLLOR;
typedef  int*    INTP;
```

Let us see, how variables can be declared

```
RS   amount1;                    → equal to:   float  amount1;
DOLLOR   amount2;                → equal to:   float  amount2;
```

Here amount1 & amount2 are nothing but **float** type variables and this is equal to
float amount1, amount2;

Similarly, structure types can be declared in two ways.

```
struct  Date
{         int  d,m,y
};
typdef   struct  Date     DATE;          //now DATE is a convenient name of struct Date;
```

The above structure can also be declared in simple way as

```
typedef  struct  Date
{         int d,m,y;
} DATE;                    → this DATE not a variable, it is alias-name of struct Date
```

Advantages of a structure

1. Easy to access member in a structure:

Using structure variable, it is easy to access each member rather than when are in individual, which gives more clarity and easy to handle.

2. Assignment between structures is possible

the compiler supports for copying one structure to another structure just like any integer value.
Here all members in the structure are copied bit by bit.

```
struct  Date  x={10, 4, 2000};
struct  Date y;
y=x;   // all members of 'x' is copied to 'y'   (  thi is equal to :  y.d=x.d;  y.m=x.m;  y.y=x.y  )
```

3. Passing & returning entire structure between functions is possible

```
For example:   display(e1);              // passing all members of structure in single call
               swap (&e1, &e2);          // swapping members
```

3. Alteration of structure is simple

Adding new or deleting existing members is simple.

4. Avoiding misuse of data

Accessing members through structure-variable avoids misusing of data.

5. Reusability

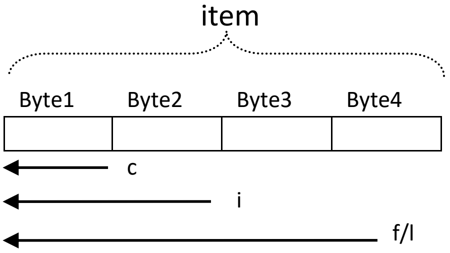
we can reuse one structure while defining another new structure.

6. We can create array of structures

Unions

Union defines the generic data-type instead of specific type, where we can store any type of value such as int/float/char/etc. The main purpose of union is to save the memory space, and also gives flexibility in coding. Union is also similar to structure but all the members in the union share the common memory area. Here space is created for only one member and remaining members share the same space, therefore we can store & access only one member at a time. The size of union is equivalent to biggest data member in it.
Note: this is used while developing customized-packages or frame-works for one language. (Not in student level programming)

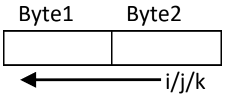
```
union myType
{          char c;
           int  i;
           long int l;
           float f;
};
union myType  item;
```



The above union variable 'item' occupies 4bytes of memory. Among 4 members, the first member 'c' uses only first byte. The second member 'i' uses first two-bytes. Similarly remaining f & l uses total 4bytes.
Here only one item can be stored & retrieved at one moment.

Demo program for how data is handled in union variables

```
union  myType
{        int i, j, k;
};
void main()
{        union myType  var;
         var.i=10;
         printf("\n    i=%d    j=%d   k=%d", var.i, var.j, var.k);
}
```



Output: i=10 j=10 k=10

In the above program, when 'var.i' is assigned a value 10, the other union members 'var.j' and 'var.k' also gain the same value as they are sharing the common memory space.

```
void main()
{        union myType  var;
         var.i=10;
         var.j=20;
         var.k=30;
         printf("\n    i=%d    j=%d   k=%d", var.i, var.j, var.k);
}
op: 30   30  30.   Here var.j=20 overwrites the var.i=10,   var.k=30 overwrites var.j=20;
```

Let us see one more example

```
union  myType
{        long l;
         float f;
};
void main()
{        union  myType  var;
         var.f=200;
         printf("\n l=%ld  f=%f\n", var.l, var.f);
         var.l=100;                                  // overwrites var.f=200 value
         printf("\n l=%ld  f=%f\n", var.l, var.f);
}
```

Output: l=garbage f=200.000000

```
  l=100      f=garbage
```

Let us see practical example, the print() function can print all types of data.

```
typedef  struct
{        union
         {        char  cc;
                  int   ii;
                  float  ff;
         }value;
         char valueType;
}MyType;

void main()
{        MyType  var;
         var.value.cc='A';
         var.valueType='c';
         print(var);
         var.value.ii=100;
         var.valueType='i';
```

```
        print(var);
        var.value.ff=300.45;
        var.valueType='f';
        print(var);
}
void print(MyType  var)
{       switch(var.valueType)
        {       case 'c':
                case 'C': printf("\n%c", var.value.cc);  break;
                case 'i':
                case 'I': printf("\n%d", var.value.ii);  break;
                case 'f':
                case 'F': printf("\n%.2f", var.value.ff); break;
        }
}
```

Enumeration

Enumerator data type is used to symbolize a list of constants with names, so that, it makes the program easy to read and modify. Enumerator maps the values to names there by we can handle the values with names.

```
enum   enumerator-name { name1=value1, name2=value2, name3=value3, ... nameN=valueN };
```

Let us see one program: this program displays no.of days in a given month.

```
enum month { jan=1, feb=2, mar=3, apr=4,  ….};
void main()
{       int  mon;
        printf("enter month number(1-12):");
        scanf(%d",&mon);
        switch(mon)
        {       case jan: printf(" January month has 31 days"); break;
                case feb: printf(" February month has 28 days"); break;
                case mar: printf(" March month has 31 days"); break;
                 ---
        }
}
```

Here all month names are replaced by its constant value at compile time, so the enumerator set symbols are evaluated of above program is

```
void main()
{       int  mon;
        printf("enter month number(1-12):");
        scanf("%d",&mon);
        switch(mon)
        {       case 0: printf(" January month has 31 days"); break;
                case 1: printf(" February month has 28 days"); break;
                case 2: printf(" March month has 31 days"); break;
                ---
        }
}
```

Enumerators with implicit default values

```
enum Colors{ red, green, blue };   // here red=0, green=1, blue=2
enum Boolean{ false, true };       // false=0, true=1
```

At compile time, all these names are automatically assigned with integer values 0, 1, 2, 3 etc.
In this way, we can specify enumerated set with valid unique names.

Enumerators with explicit user-defined values

At the time of declaration of enumeration, we can give explicit values to the names and these values can be any integers. Even two names can have same value (duplicate values).

For example

```
enum cityNames { Vijayawada=10, Guntur=20, Vizag=30 };

enum cityNames { Vizag=30, Guntur=20, Vijayawada=10 };     // may not be in order

enum cityNames { Chennai=1, Madras=1, Mumbai=2, Bombay=2 };   // duplicate values allowed

enum month {jan=1, feb, mar, apr, may, june, july, aug, sept, oct, dec };
```

Here name **"jan"** assigned with 1 by the programmer, and remaining names are automatically assigned with next succeeding numbers by the compiler. That is feb=2, mar=3, ... etc.

```
enum  Set { A=10, B, C=20,  D,  E,  F=30, G, H };        //Here B=11, D=21, E=22, G=31, H=32
```

Ambiguity error

There is one problem with enumeration, if any variable exist with same name in enumerator set in same scope then we get error at compile time.

For example,

```
enum color { red, green, blue, white, yellow};
int red=0;                    // ambiguity error with red, so either one must be removed
void main()
{        int white=100;   // preference is given to local white variable, instead of white in enumeration
         printf("%d %d %d %d", green, blue, white, yellow};
}
output:  0  1  100  4
```

Enumeration variables

Using enumerator name, we can specify variables of its type. This makes the convenient and increases the readability of the program. The enumerator variables are nothing but int-types. For example

enum enumerator-**name variable-1 ,variable-2, variable-3, ... ;**

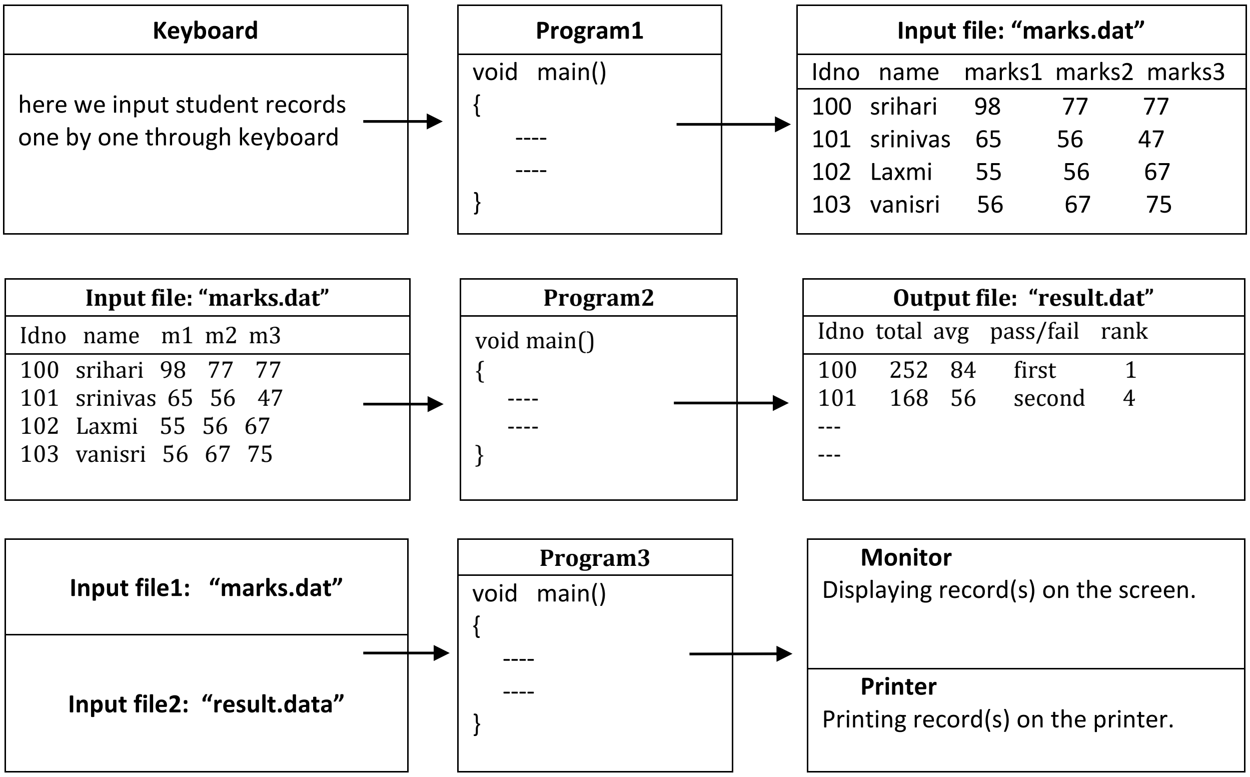
```
enum Color { Red=15, Green=26, Blue=13, White=5, Yellow=4, Grey=17 };
void main()
{        enum  Color   wallColor, floorColor;
         wallColor=White;
         floorColor=Grey;
          …
          …
}
```

File handling in C

In real life applications, some kind of data needs to be accessed more times in present and future purpose like bank transactions, business transactions, etc. Therefore such data is kept permanently in the computer, so that it can be accessed many times later.

For this purpose, secondary storage devices are used to store the data permanently in the form of files. These files often called data files. Thus, data file can be defined as collection of data, stored permanently in secondary storage device. Each file is identified by a valid name called **file-name**. The interaction with files is treated as interaction with io devices in the computers. Therefore, all secondary memory devices treated as **io** devices.

Let us consider a menu driven program to automate student marks in a college. Let the marks scanned from keyboard are stored permanently in a data file called "marks.dat", later, such marks are processed and stored the results in a separate file called "result.dat" or displayed on the screen. This is depending upon the requirement of problem. Following picture explains how our programs interact with files.


Keyboard
here we input student records one by one through keyboard
Program1

```
void   main()
{
      ----
      ----
}
```

Idno	name	marks1	marks2	marks3
100	srihari	98	77	77
101	srinivas	65	56	47
102	Laxmi	55	56	67
103	vanisri	56	67	75

Idno	name	m1	m2	m3
100	srihari	98	77	77
101	srinivas	65	56	47
102	Laxmi	55	56	67
103	vanisri	56	67	75

```
void main()
{
    ----
    ----
}
```

Output file: "result.dat"

Idno	total	avg	pass/fail	rank
100	252	84	first	1
101	168	56	second	4

Input file1:   "marks.dat"
Input file2: "result.data"
Program3

```
void   main()
{
    ----
    ----
}
```

Monitor
Displaying record(s) on the screen.
Printer
Printing record(s) on the printer.

In computer science, the files are classified into 2 types

1. Data files: for example, pdf files, jpg image files, mp3 sound files, mp4 video files, business data files, and program's source code files like c, cpp, java files, etc.

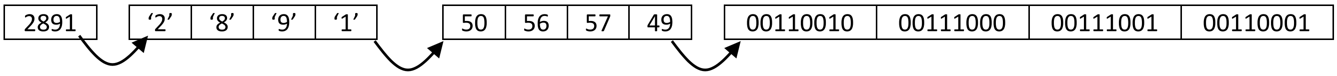
2. Program files: for example, .exe, .com, .dll, etc; Contains machine code instructions(executable code) this chapter deals with data files, the data files are stored in two formats **text & binary**.

1. Text Format Files: Here the data is stored in the form of ascii values (string format)

2. Binary Format Files: here the data is stored as it is in program variables (raw format))

Text Files

In text files, everything is stored in the form of ascii values, even **numeric data** is converted into string format with its ascii values of each digit and stored in file. For example, **'k'** contained a value 2891, this number is stored in text file as



2891 → ‘2’ ‘8’ ‘9’ ‘1’ → 50 56 57 49 → 00110010 00111000 00111001 00110001

Here 50, 56, 57,49 are ascii values of 2, 8, 9,1 respectively. Internally each digit ascii code is again stored in binary form as above shown (as computer understands only binary values), in this way, the text data is stored and retrieved from text files. These files are only suitable for public sharing related files such as document files, help files, message files, source code program files, small data files, configuration files, etc. In general, these files are handled in two ways, by writing a specific program to manage them, or by using ready-made text editors. Using text editor, we can create & modify the text data. We have several text editors like notepad, word pad, ms-word, etc.

The value 26 is inserted as end-of-file mark and also the new-line character is stored as "\r\n"(unix format). The library functions does this job while writing ‘\n’ to file, whereas while reading back, it is considered as only one character (\n). (We need to worry about this ‘\n’)
Some people raise a doubt, if file data itself contained 26 like employee age, then how it differentiates with the end-of-file 26. Our 26 is stored as ‘2’ and ‘6’ as its ascii values 50 & 54, so no conflict between these two.

Binary files

In this kind of file organization, the data is stored as it is in program variables. Here no conversion is made while storing and reading back unlike text files. Generally, binary files are extensively used for maintaining similar type of data like employee records or bank accounts, insurance accounts, etc.
In binary i/o, as it is of storing & reading back takes place between the RAM & hard disk, therefore binary files are faster than text files. At end of text files, the value 26 is inserted as end-of-file-mark, whereas in binary files no such value is inserted, where end-of-file mark is known by the file-size.

In case of binary files, we cannot manipulate the data using text editors. For example, take student structure idno|name|marks→int|char(30)|int. if this data dumped as it is into file, then it has to be read back as it is. (2byte at a time for idno, 30byte at a time for name,..) So, text editors unaware of this user-defined structure, hence we cannot handle binary files using text-editors.(text editors are designed to read/write the data only in the form of ascii values)
There should be separate software needed to handle binary files. For example ‘pdf’ file is a binary format file and cannot be opened in text editor, a special software like ***Adobe-R*eader** is used. Similarly, jpg files, image files, sound files, movie files are managed using special software.

In C, files can be handled in two ways, 1. High Level file handling, 2. Low Level file handling.

In high level file handling, there are plenty of library functions with file streaming is available, here C provides more support like reading/writing of text/binary data from file, converting numeric to text, text to numeric, interaction with printer, random file organization, checking end-of –file pointer, etc. So it is easy to handle files in high level organization, but in low level, there is no library functions support, here we need to write the code from zero level. This low level is used in hardware industries for chip level programming to develop customized softwares (packages), drivers, utilities. Etc.

File Streaming or File Buffering:

The total hard disk space is divided into several blocks, where each block size 1024/2048 bytes. This is not fixed size and it may vary. Our files content are stored in these blocks of hard disk; if file size is 3000 bytes then it takes two or more blocks. We can't do any insert/delete/modify operations on file data when it is in the hard disk, because, the hard disk supports only block by block reading or writing. So first we dump the file data from hard disk to RAM and then we do respective operations, later we store back updated data to disk. RAM supports byte-by-byte read/write operations thereby we can do any operations when data is in the RAM.

When we open an existing file, the ***total file*** contents would not be dumped into RAM, only the first block dumped into RAM and then file pointer set to it, the remaining blocks are loaded one by one when file pointer moves advance. File pointer is a pointer used to read/write file contents. When a file pointer moves end of first block, the second block loads into same memory (1st block replaces by 2nd block), in this way files data loads into RAM and operated, this process is called file-streaming and managed with help of operating system. For random file organizations, the blocks are loaded & replaced randomly as per movement of file pointer. OS provides this file streaming facility to move the file-data from disk-to-RAM and RAM-to-disk.
Stream = array of bytes memory + read() fn + write() fn + open() fn + close() fn + other functions.
This array of bytes memory is also called buffer or stream-memory. The C library functions interact with this OS stream functions, we don't need to bother about how to manage file-streaming.

High-level file IO

Operations on files are mainly three

1. Opening a file
2. Applying read/write/modify operations on file data
3. Closing a file

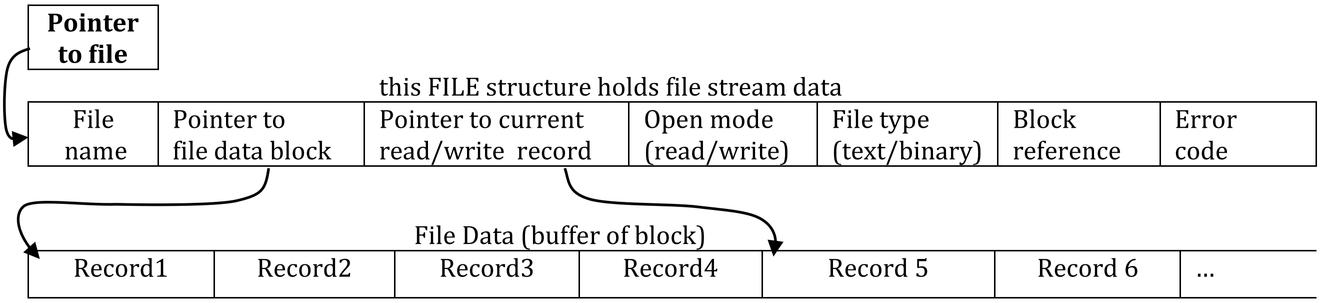
Opening a file: The function fopen() is used to open a file, when we open a file, the file contents are dumped from hard-disk to RAM and then file pointer set to it. The function proto-type is

FILE* fopen(char *fileName, char *openMode);

The first argument is the name of file which we want to open, and second argument is opening mode.

```
For example:   FILE *fp;                        // pointer to FILE structure
               fp=fopen("marks.dat" , "rt");   // r: read-mode,  t: text-file
```

when a file is opened in the RAM then its memory picture as given below structure



Here FILE is a predefined structure, defined in "stdio.h" file. This structure holds extra information about file-name, open-mode, file-type(binary/text), buffer-size(block-size), pointer to file data, pointer to current read/write byte, etc. This **FILE** structure is used for file streaming as said above.
The fopen() function first creates a memory for FILE structure and file-data-buffer (follows dynamic memory allocation system), later loads file contents from disk to this memory. Finally returns this FILE structure address (as shown above). If unable to open a file then it returns NULL.

File open modes:

r → read mode: Opens an existing file for reading. (We cannot write/modify existing data).
If file not found while opening, then fopen() returns NULL.

w → write mode: Creates a new file for writing, if file already exists with same name, its content will be over written. (old data erased)

a → append mode: Opens an existing file for adding new data at the end. If file does not exist with that name, then it will create a new empty file and adds the data to it.

r+ → allows all operations: Opens an existing file for updating, it allows all operations like reading , writing, modifying. If file not found then it returns NULL.

w+ → write + modify: Creates a new file for writing. If file already exist with same name then its contents are erased. if once new data is written to file then "w" mode does not allows to go back and modify it, but "w+" allows it.

a+ → append + modify: Appends new data and also allows to modify new data (not old data)
we can append new contents at the end of file, as well as we can read-back/modify the just newly written data. If file not exist then opens new file for adding.

b/t → (b for Binary file/ t for Text file): To specify that a given file be a text or binary file.

Closing a file: The function 'fclose()' closes an already opened file. After completion of our task on file, it must be closed. This operation involves in updating file contents on disk (saving back to disk) and also releases buffer & FILE structure's space in the RAM. The syntax is

int fclose(FILE *filePointer); // It returns zero, if successfully closed a file, otherwise returns -1.

The I/O functions on files

fprintf(): it works same as printf(), instead of writing data one the screen, it writes to file
fscanf(): it works same as scanf(), instead of reading data from keyboard, it reads from file.
These two functions are specialized to handle text files.
Using these functions, we can also perform formatted input/output on text files.
Formatted means, writing data using io flags such as %d %5d %-5d %.2f %s %30s etc;

fread(), fwrite(): These are specialized for binary files reading/writing data. Of course, we can also use for text files, but formatted input/output does not support.
EOF: it is a macro, represents end of file mark for the text files. (EOF value is equal to -1)

Scanning integers from keyboard and writing to a file

Scans input values one by one from keyboard until last input is zero, writes each scanned value into disk.

Input from keyboard
20 30 40 45 56 23 34 0 ↲

output file →"sample.txt"
20 30 40 45 56 23 34

```
#include<stdio.h>
void main()
{       FILE *fp;                            // pointer to file
        int n;
        fp=fopen("sample.txt", "wt");        // w=write mode, t=text file.
        if( fp==NULL )
        {    printf("file not opened");  return;
        }
        while(1)
        {    printf(" enter a value :");
             scanf("%d" , &n);
             if( n==0) break;
             fprintf( fp , "%d\n",  n );    //  writes 'n' to file
        }
        fclose( fp );     // saving file contents on disk
}
```

Reading integers from a file

This is opposite to above program, reading integer from file and showing on the screen.

```
#include<stdio.h>
void main()
{       FILE *fp;   int n , k;
        fp=fopen("sample.txt", "rt");
        if(fp==NULL)
        {    printf("file not opened");  return;
        }
        while(1)
        {    k=fscanf(fp, "%d", &n);        // reading an integer from file, fscanf() returns -1  if  EOF reached
             if(k==-1 or k==EOF) break;    // stopped when end of file is reached, EOF value is -1
             printf("%d ", n);             // displaying on the screen
        }
        fclose(fp);   // will not be saved on disk, because opened in read-only mode(no updations),
}                                                                        so only clears from RAM.
```

Above program in binary i/o format

```
#include<stdio.h>
void main()
{         writeToFile();
          readBackFromFileAndPrint();
}
void writeToFile()
{         FILE *fp;
          int n;
          fp=fopen("sample.dat", "wb");   // w=write-mode ,  b=binary-file
          if(fp==NULL)
          {         printf("file not opened"); return;
          }
          while(1)
          {      printf(" enter a value :");
                scanf("%d" , &n);
                if( n==0) break;
                fwrite( &n, sizeof(int), 1, fp );      // for binary files, use only fwrite(), fread()
          }
          fclose(fp);        // saves file contents on disk
}
void readBackFromFileAndPrint()
{         int k , n;   FILE *fp;
          fp=fopen("sample.dat", "rb");
          if(fp==NULL)
          {         printf("file not opened"); return;
          }
          while(1)
          {         k=fread(&n, sizeof(n), 1, fp);
                    if(k<=0) break;           //  fread() returns 0 or -1  when end of file is reached
                    printf("%d  ", n );        // printing integer on screen which has read from file
          }
          fclose(fp);        // will not be saved on disk, because opened in read-only mode, only clears from RAM.
}
```

fwrite() syntax: fwrite(&varaible, sizeof(variable), no.of variable items, file-pointer)

&variable → this gives address of variable where the data is exist

sizeof(variable) → this gives size of data to be written to file from given variable address .

no.of variable items → in case array of items exist.

file-pointer → refers pointer to file.

fread() syntax: fread(&varaible, sizeof(variable), no.of variable items, file-pointer)

&variable → this gives address of variable where the data to be inserted

sizeof(variable) → this gives size of data to be read from a given file.

no.of variable items → in case array of items exist.

file-pointer → refers pointer to file.

Copying odd numbers from one file to another file

Let our text file **“input.txt”** contained some even & odd numbers, now read numbers one by one from file and copy only **odd** numbers into another file called **“output.txt”**.

file name: “input.txt”		File name:”output.txt”
12 17 20 13 29 30	→	17 13 29 35 43 27
32 35 40 43 27 20		21 29 31 39 11 3
21 29 31 39 11 12		
10 8 2 3		

```
void main()
{       FILE *fs, *ft;   int n, k;
        char sName[30], dName[30];
        printf(“enter source & destination file names :”);
        gets(sName);   gets(dName);
        fs=fopen(sName,”rt”);
        ft=fopen(dName,”wt”);
        if( fs==NULL || ft==NULL)
        {       printf(“file(s) not opened”); return;
        }
        while(1)
        {       k=fscanf(fp,”%d”, &n);
                if(k<=0) break;
                if(n%2==1)   // if odd number then write to file
                    fprintf(fp, “%d space“, n);
        }
        fclose(fs);  fclose(ft);
}
```

Copying one file content to another file (file can be binary or text)

Note: the binary file organization works for both text/binary format files

this program reads byte by byte from source file and writes to target file, to store byte values, the suitable data type is char.

```
void main()
{       FILE *fs, *ft;  char ch;
        char sName[30], dName[30];
        printf(“enter source & destination file names :”);
        gets(sName);  gets(dName);
        fs=fopen( sName , ”rb”);
        ft=fopen( dName , ”wb”);
        if( fs==NULL || ft==NULL) { printf(“files not opened”); return; }
        while(1)
        {       k=fread( &ch, sizeof(ch), 1, fs );
                if(k<=0) break;
                fwrite( &ch, sizeof(ch), 1, ft );
        }
        fclose(fs);  fclose(ft);
}
```

More file IO functions

fputc(): It is used to write single character to a file. It takes two arguments, a character which is to be written, and a pointer of FILE structure. **Syntax: int fputc(character, FILE pointer);**

fgetc(): It is used to read single character from a file and returns ASCII value of it. **ch=fgetc(file pointer);**

fputs(): Writes a string to file

fgets(): Reads a string from file

ftell(): returns current read/write position of the file pointer.

fseek(): moves the file pointer, for example

fseek(fp, 10, SEEK_ CUR): moves file pointer 10bytes forward from current position.

fseek(fp, -10, SEEK_ CUR): moves file pointer 10bytes backward from current position.

fseek(filePointer, 0, SEEK_END): moves the file pointer to end of file.

fseek(filePointer, -100, SEEK_END): moves the file pointer 100 bytes back from end position.

fseek(fp, 0, SEEK_SET): moves file pointer to beginning of file. SEEK_SET defines beginning of file

fseek(fp, 100, SEEK_SET): moves file pointer to 100bytes forward from beginning of file.

SEEK_END is a macro defines the end-of-file. Value is equal to 2.

SEEK_SET is a macro defines the begging-of-file. Value is equal to 0.

SEEK_CUR is a macro defines the current-position of file. Value is equal to 1.

EOF: it is a macro, represents end of file mark for the text files. (EOF value is equal to 26)

feof(): It is used to check whether a given file has reached the End Of File mark or not.

rename():changes the name of an existing file. syntax: rename(“oldName”, “newName”);

remove():deletes a file from disk. syntax: remove(“fileName”);

filelength(): it gives length of file (the number of bytes occupied by the file)

Writing scanned text to a file

This program accepts a text (char by char) from keyboard and writes into a file called "sample.txt". The written contents of file can be seen using any text editor like notepad.

```
#include<stdio.h>  // to include FILE structure and its IO functions.
void main()
{    char ch;  FILE *fp;
     fp=fopen("sample.txt","wt");           // opening text file in write mode
     if(fp==NULL)
     {        printf(“unable to open the file ");  return;
     }
     printf("\n enter text (at end of input type F6) :”);
     while(1)
     {        ch=getchar();  or scanf(“%c”, &ch)      //accepting char by char (text) from Keyboard
              if(ch==EOF) break;                     // EOF is end of file mark, when F6 pressed
              fputc(ch,fp);  or fprintf(fp, “%c”, ch);  // writing single char to file
     }
     fclose(fp);        // storing (saving) file contents into disk
}
```

```
Input:
        hello ↲
        how are you ↲
        how do you do ↲
        fine ↲
        thanks ↲
    ^Z   (press f6) (^Z is equal to EOF
```

```
Output:
File name: "sample.txt"
hello
how are you
how do you do
fine
thanks
```

Counting upper, lower, digits and others in a given file

This program counts number of upper case alphabets, lower case alphabets, digits, words and lines in a given text file. It reads char by char from a given file and checks the each character and counts.

```
#include <ctype.h>
#include<stdio.h>
void main()
{     FILE *fp;
      int upperCount,lowerCount, digitCount, lineCount,wordCount;
      char fileName[30], ch;
      puts("Enter the file name:");
      gets(fileName);
      fp=fopen(fileName,"rt");
      if(fp== NULL)
      {         printf("file not found");
                exit(0);
      }
      upperCount=lowerCount=digitCount=lineCount=wordCount=0;
      while(1)
      {         ch=fgetc(fp);              // reading single char from file
                if(ch==EOF) break;      //  EOF → end of file indicator
                if( isupper(ch) ) upperCount++;
                else if( islower(ch) ) lowerCount++;
                else if( isdigit(ch) ) digitCount++;
                else if(ch==' ') wordCount++;
                else if(ch=='\n')
                {         wordCount++;
                          lineCount++;
                }
      }
      fclose(fp);
      printf("lower count = %d", lowerCount);
      printf("upper count = %d", upperCount);
      printf("digits count = %d", digitCount);
      printf("words count = %d", wordCount);
      printf(" lines count = %d",  lineCount);
}
```

Handling student details (binary files)

This program accepts student marks from keyboard and inserts each record into a file called “marks.dat”, later read back and process the result and shows on the screen.

Note: this is demo program, for binary file IO system; here fread() & fwrite() functions are used

input:

```
enter idno (0 to stop): 101
                    enter name:  Srihari
          enter marks1 , marks2:  66 77

           enter idno (0 to stop): 102
                    enter name:  Narahari
          enter marks1 , marks2:  88 99
```

ouput:

```
        idno      name        mark1     mark2     result
     -------------------------------------------------------------------
        100       Srihari        70         80        pass
        101       Narahari       80         20        fail
```

```
#include<stdio.h>
struct Student
{      int  idno,m1,m2;
       char name[30];
};
void main()
{      acceptMarks();
       processAndShow();
}
void acceptMarks()
 {     FILE *fp;
       struct Student st;
       fp=fopen("marks.dat","wb");
       while(1)
       {        printf("\n enter idno (0 to stop):");
                scanf("%d",&st.idno);
                if( st.idno==0)  break;  // 0 is end of input
                printf("enter name:");
                fflush(stdin);
                gets(st.name);
                printf("enter marks1 , marks2:");
                scanf("%d%d",&st.m1, &st.m2);
                fwrite( &st, sizeof(st), 1, fp);    // for total, average garbage will be stored
       }
       fclose(fp);
 }
void processAndShow()
{         FILE *fp;
          struct Student st;
          int k, total, avg;
          char *result;
          fp=fopen("marks.dat","rb");
          printf("\n%6s   %-20s %5s %5s   %-20s", "idno", "name", "mark1", "mark2", "result");
          printf("\n-------------------------------------------------------------------------");
          while(1)
          {         k=fread( &st, sizeof(st), 1,fp);
                    if(k<=0) break;                    // end of file reached then stop it.
```

```
            total=(st.m1+st.m2);
            avg=total/40;
            if( st.m1<35 || st.m2<35 )
                    result="Failed";
            else if(avg>=60)
                    result="First class";
            else if(avg>=50)
                    result="Second class";
            else  result="Third class";
            printf("\n  %-6d %-20s %5d %5d   %-20s", st.idno, st.name, st.m1, st.m2, result);
    }
    fclose(fp);
}
```

A menu driven program to handle employee records (mini project work)

A menu driven program to handle employee records

This program manages employee information: it supports adding newly joined employee details, deleting relieving employee record, modifying address or any other information of employee, printing particulars. Employee details are stored in a separate file called "emp.dat".
in this menu run program, the user can select his choice of operation.

```
              Menu Run
        -------------------------------
        1. Adding new employee
        2. Deleting employee record
        3. Modifying existing record
        4. Printing employee details
        0. Exit
                   Enter choice [1,2,3,4,0]:
        #include<stdio.h>
        typedef  struct                    // structure of an employee
         {        int  empNo;
                  char name[30];
                  float salary;
        }EMP;
        void append(); void modify(); void delete(); void print();      // fn proto-types
        void main()
        {         int choice;
                  while(1)          // loop to display menu continuously
                  {   printf("\n\n==========================================================");
                      printf("\n 1.append \n 2.delete \n 3.modify\n 4 print \n 0.exit");
                      printf("\n   Enter choice [1,2,3,4,0]:");
                          scanf("%d", &choice);
                          switch(choice)
                          {         case 1:  append(); break;
                                    case 2:  delete(); break;
                                    case 3:  modify(); break;
                                    case 4:  print(); break;
                                    case 0:  exit(0);
                          }
                  }
        }
// -----------------------------------------------------------------------------------------------------------------
//  this append() function appends a record at end of file
    void append()
    {       FILE *fp;   EMP e;
            fp=fopen("emp.dat", "ab");    // 'a' for append, 'b' for binary-file
            if(fp==NULL)
            {       printf("\nUnable to open emp.dat file");
                    return;
            }
            printf("\n enter employee no:");
            scanf("%d",&e.empNo);
            printf("Enter employee name:");
            fflush(stdin);
            gets(e.name);
            printf("enter salary :");
            scanf("%f",&e.salary);
            fwrite(&e,sizeof(e),1,fp);
            fclose(fp);
            printf("\n successfully record added");
    }
//  ------------------------------------------------------------------------------------------------------------------------
//  delete() function.  Direct deletion of record is not possible from a file, alternative is, copy all records
// into another temp file except deleting record; later temp file will be renamed with the original file name.
    void delete()
    {       FILE *fp,*ft;   EMP  e;
            int eno, found=0, k;
            fp=fopen("emp.dat", "rb");
            ft=fopen("temp.dat", "wb");
            if(fp==NULL || ft==NULL )
            {       printf("\nunable to open file");
                    fclose(fp);  fclose(ft);  return;
            }
            printf("\nenter employee number to delete record :");
            scanf("%d",&eno);
            while(1)
            {       k=fread(&e,sizeof(e),1,fp);
                    if(k==0)break;
                    if(eno==e.empNo)
                            found=1;        // record is found
                    else
                            fwrite(&e,sizeof(e), 1 ,ft);
            }
            if(found==1) printf("\nRecord deleted success fully");
            else  printf("\nRecord Not found");
            fclose(fp);     fclose(ft);
            remove("emp.dat");                  // deletes old-file from disk
            rename("temp.dat","emp.dat");
    }
//  -------------------------------------------------------------------------------------------------------------------------------------
//  Modifying data in a record. First, it searches for modifying record in a file, if record is not found then
// displays error message & returns. If found, then old-salary overwrites by new-salary of a record
void modify()
{       EMP e;
        int found=0 , eno, k;
        long int pos;
        FILE *fp;
        fp=fopen("emp.dat","rb+");
        if(fp==NULL)
        {    printf("\nfile not found ");
             exit(0);
        }
        printf("\n enter employee number:");
        scanf("%d",&eno);
        while(1)
        {       k=fread(&e,sizeof(e),1,fp);
                if(k==0) break;
                if(eno==e.empNo)
                {       found=1;        // record is found
                        break;
                }
        }
        if(found==0) {  printf("\n Record not found"); return; }
        pos=ftell(fp);                  // this function returns the current position of read/write pointer
        pos=pos-sizeof(e);
        fseek(fp, pos, SEEK_SET);       // move the file pointer one position back
        //  after reading our search record , the file pointer moves to next-record, so to replace our record data, we have to move
        //  the file pointer back. The above code moves like that.
        printf("old salary : %.2f", e.salary);
        printf("enter new salary:");
        scanf("%f", &e.salary);
        fwrite(&e,sizeof(e), 1 ,fp);            // overwriting old record
        fclose(fp);
        printf("\n address suceessfully modified");
  }
//  --------------------------------------------------------------------------------------------------------------------------------
// print function, prints all records
        void print()
        {       EMP e;  int k, count=0;
                FILE *fp;
                fp=fopen("emp.dat", "rb");
                if(fp==NULL)
                {       printf("\nfile not found ");
                        return;
                }
                while(1)
                {       k=fread(&e,sizeof(e), 1,fp);
                        if(k==0) break;
                       printf("\n employee number: %d",e.empNo);
                       printf("\n employee name : %s",e.name);
                       printf("\n Salary: %.2f", e.salary);
                       count++;
                       printf("\n-------------------------------");
                }
                printf("\n(%d) records found", count);
                fclose(fp);
        }
```

This book softcopy is available in pdfhost.io website, to download the file, search through keyword ‘cfamily’. This pdf file(s) designed for 2-sided print-copy (Xerox hard copy).