



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.**

A soft copy of this is available on the C-Family website.

Search in Google with the text: cfamily computers vijayawada.

Chapter Index	Page No
Number Systems & memory measurements	5
Number System conversion	5
Memory size and min-max value capacity	8
Introduction to C	10
Evolution of languages	11
Features of C language	12
Operating systems	13
Data types	15
Operators	16
garbage values, comment lines	22
Introduction to Programming	23
library-functions, user-functions, proto-type of function	24
about main()	25
first C program	28
developing , compiling and running a program	33
escape sequence characters	36
type casting	37
representation of constants in C	39
flow chart, algorithm, pseudo code	41
Decision Control Structures	45
if, if-else	45
nested-if	55
If-else-if ladder	65
If-else linking, about Boolean value, null instruction	70,
switch statement, conditional operator, goto statement	75
Looping Control Structures	83
while loop	83
for loop	111
do-while loop	110
break, continue	111
nested loop	119
One Dimensional Arrays	129
accessing & initialization of arrays	130
about array size , array boundaries	132

Functions	143
types of functions, library functions, user functions	145
function body & its call syntax	150
more about functions	152
Pointers	165
operators in pointers	167
arrays & pointers, passing array to function	159,160
call-by-value vs call-by-reference	172
Sorting & Searching	189
Storage Classes	197
Preprocessor Directives	205
2D Arrays	211
Characters & Strings	221
initialization of strings	227
pointer to string, passing array to function	234
array of strings	240
Recursion	249
Dynamic Memory Allocation	267
Command Line Arguments	272
User Defined Data types	273
structures, array of structures	274
different declaration structures, typedef, advantages	277
union, enumeration	290
File Handling	293
text files, binary files	295
IO operations on files	300

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 vs Binary System

The word "decimal" means 10; the base of the decimal system is 10. Here, we have symbols 0, 1, 2, ...9.

The word "binary" means 2; the base of the binary system is 2. Here, we have symbols 0, 1.

We know computers follow the binary number system, where they manage data and instructions in the form of binary numbers (0/1). These binary numbers are managed as two voltages, for example, 5 volts for 0 and 12 volts for 1. So, computers use the binary system for storing and processing data.

We know humans follow the 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 the decimal number 13 into its binary form.

Continuously divide 13 by 2 and collect the quotients and remainders. The collection of remainders forms a binary number, as given below.

$$\begin{array}{r}
 13 & \text{Remainders} \\
 \downarrow & \downarrow \\
 \hline
 13 \% 2 \rightarrow 1 \\
 6 \% 2 \rightarrow 0 \\
 3 \% 2 \rightarrow 1 \\
 1 \% 2 \rightarrow 1 \\
 0
 \end{array}$$

In repeated division, take the quotient as the integer part, for example $13/2 \rightarrow 6$ (not 6.5)

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

Binary to Decimal Conversion

Multiply all digits(bits) of N with $2^0, 2^1, 2^2, 2^3, 2^4 \dots$ 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.

$$\boxed{\begin{array}{r} 1 & 1 & 0 & 1 \\ 2^3 & 2^2 & 2^1 & 2^0 \end{array}} \rightarrow \boxed{1*2^3 + 1*2^2 + 0*2^1 + 1*2^0} \rightarrow 13$$

Example2: converting binary number 11001 to its decimal 25.

$$\boxed{\begin{array}{r} 1 & 1 & 0 & 0 & 1 \\ 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \end{array}} \rightarrow \boxed{1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0} \rightarrow 25$$

Octal number system

The base of this system is 8, and we have 8 digits: 0, 1, 2, 3, 4, 5, 6, 7.

This system is old and was later replaced by the hexadecimal system.

It is used in programming to represent binary numbers in octal, since taking a binary number as it is in programming is unreadable and confusing because everything is just 1s and 0s. Therefore, we use octal or hexadecimal representations for easiness. Here every **3 binary bits forms = 1 octal digit**.

For example: Binary 101110 → 101-110 → 56 in octal → here 101 is 5, 110 is 6.

Let us take binary: 1010111011 (observe this binary value, reading and understanding is difficult)
 1-010-111-011 (dividing into 3bits from right to left)
 1-2-7-3 → 1273 (this is equal to octal number)

In this way, binary numbers are represented in octal for easier understanding. Therefore, in programming, we have two choices to write the above value: either in binary or in octal. If we want to write it in binary, it should be written as 0B1010111011; or else, in octal as 0C1273. Here, 0B represents a binary value, and 0C represents an octal value.

Decimal to octal conversion

To get octal number from a decimal value, continuously divide N by 8 and collect the quotients and remainders. The sequence of remainders forms an octal number, as shown below.

98		Remainders	
↓		↓	
98	%	8	→ 2
12	%	8	→ 4
1	%	8	→ 1
0	stop		

$98_{(10)} \rightarrow 142_{(8)}$

Octal to decimal conversion

Multiply all digits of N with $8^0, 8^1, 8^2, 8^3, 8^4 \dots$ from right-to-left. The sum of all such products forms a decimal number. Following example explains how to convert binary number 142 to its decimal 98.

$$\begin{array}{r} 1 \quad 4 \quad 2 \\ 8^2 \quad 8^1 \quad 8^0 \end{array} \rightarrow \boxed{1*8^2 + 4*8^1 + 2*8^0} \rightarrow 98$$

Hexadecimal 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 hexadecimal is updated system from the octal system. As we know, the binary system is very difficult to follow because everything is in 1 or 0. So, we represent binary numbers in the hexadecimal system for simplicity. Every 4 binary bits forms = 1 hexadecimal digit. Here, we divide a binary number into 4-bit groups, where each group contains exactly 4 bits and is represented by a hexadecimal value. Let us take a sample binary number: 1101100111000011.

This binary value is divided into 4-bit groups as 1101-1001-1100-0011 and is represented in the hexadecimal system as D9C3. In programming, hexadecimal is written as 0XD9C3. Here, the prefix 0X indicates that it is a hexadecimal value.

The binary value 1101 1001 1100 0011 represented as

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

Conversion of decimal to hexadecimal

To get hexadecimal number from a decimal value, continuously divide N by 16 and collect the quotients and remainders. The sequence of remainders forms a hexadecimal number, as shown below.

		Quotient → Reminders
16		↓ 370359 ↓
16	23147	→ 7
16	1446	→ 11 (B)
16	90	→ 6
16	5	→ 10(A)
		→ 5

370359₍₁₀₎ → 5A6B7₍₁₆₎

Hexadecimal to decimal Conversion

Multiply all digits of N with 16^0 , 16^1 , 16^2 , 16^3 , 16^4 ... from right-to-left. The sum of all such products forms a decimal number. Following example explains how to convert hexadecimal 5A6B7 to its decimal 370359.

5 A 6 B 7	→	$5*16^4 + 10*16^3 + 6*16^2 + 11*16^1 + 7*16^0$
16^4 16^3 16^2 16^1 16^0		→ 370359

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 or 0

8 bits = 1 byte → this is lowest addressable item in memory measurements.

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
8 th bit	7 th bit	6 th bit	5 th bit	4 th bit	3 rd bit	2 nd bit	1 st 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
8 th bit	7 th bit	6 th bit	5 th bit	4 th bit	3 rd bit	2 nd bit	1 st 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
4 th bit	3 rd bit	2 nd bit	1 st 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
4 th bit	3 rd bit	2 nd bit	1 st 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 $2^4 - 1$.

$$\begin{array}{cccc} 1 & 1 & 1 & 1 \\ \hline 2^3 & 2^2 & 2^1 & 2^0 \end{array} \rightarrow \begin{array}{cccc} 1*2^3 & + & 1*2^2 & + & 1*2^1 & + & 1*2^0 \\ 8 & + & 4 & + & 1 & + & 1 \end{array}$$

$$1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 \rightarrow 2^3 + 2^2 + 2^1 + 2^0 \rightarrow 2^4 - 1$$

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

$$\text{Let } X = 2^0 + 2^1 + 2^2 + 2^3$$

$$X = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 - 2^4 \quad // \text{after adding and subtracting } 2^4 \text{ to this equation (no change in value)}$$

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

$$X = 2^0 + 2(2^0 + 2^1 + 2^2 + 2^3) - 2^4 \quad // \text{taking 2 as common}$$

$$X = 2^0 + 2(X) - 2^4 \quad // \text{since } X = 2^0 + 2^1 + 2^2 + 2^3$$

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

So in 8 bits (1 byte) memory, we can store values in range 0 to $2^8 - 1$ (0 to 255),

in 16 bits (2 byte) memory, we can store values in range 0 to $2^{16} - 1$ (0 to 65,535).

Representing -ve values

For signed types, the leftmost bit is used to represent the sign (+ve/-ve). This bit is called the sign bit.

If this bit contains 0, then it is said to be a +ve sign value; otherwise, it is a -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	
8 th bit	7 th bit	6 th bit	5 th bit	4 th bit	3 rd bit	2 nd bit	1 st bit	

The value -13 is stored as

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

Note: This type of value representation is called “**signed magnitude representation.**”

So, in 8 bits (1 sign bit+7 data bits) of memory, we can store values in the range of -2^7 to 2^7-1 (**-128 to 127**).

Actually, for negative values, the **two's complement** method is used. In this method, an extra **+1** is added to the given number during conversion, so we get **-128** instead of **-127** on the negative side.

(In the next topic, we will see what two's complement is.)

Binary addition

Addition of binary values

$X \rightarrow$	0	0	1	1
$Y \rightarrow$	0	1	0	1
$X+Y \rightarrow$	0	1	1	10

When we add $1+1 \rightarrow$ 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.

$X \rightarrow$	1	Carry	1	1	1	1	carry	1	1
$Y \rightarrow$	1	$X \rightarrow$	1	1	0	1	$X \rightarrow$	1	1
$Z \rightarrow$	1	$Y \rightarrow$	1	1	1	1	$Y \rightarrow$	1	1
$X+Y+Z \rightarrow$	11	$X+Y \rightarrow$	1	1	1	1	$X+Y \rightarrow$	1	1

Binary subtraction

In a computer, subtraction is done like addition using the 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 and borrow from the next digits. For example, consider the subtraction of 3456 - 0999. Here, first, we try to subtract the last digits, 6 - 9, but since $6 < 9$, we need to borrow from the next digits. In this way, we have to look up and borrow from the next position digits. This process takes much time and is complex.

The best solution is the 2's complement method.

2's complement is \rightarrow 1's complement + 1

1's complement is \rightarrow reverse of a given number (replace 1 by 0, and 0 by 1 in a given number)

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 (quite opposite value)

If $X=0001$, then 1's complement of X is 1110

2's complement is \rightarrow 1's complement + 1

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

Actually, the 2's complement value is simply the –ve of a given number. So adding this value, we get subtracted value automatically. Let us follow the process below.

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 $0110+1 \rightarrow 0111$
3. Addition of $1101 + 0111 \rightarrow 10100$ (this is addition of X + 2's Complement of Y)
4. Remove left most bit from 10100, then it is $\rightarrow 0100$ (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.

Brief History of C

In the 1960s, there was a need for a general-purpose programming language, and the available ones were specific-purpose only. For example, COBOL was meant for business applications, and FORTRAN was extremely small and suitable only for scientific applications. Instead of being small, simple, and specific, CPL was intended to support a 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 BCPL (Basic CPL). Still, it retained some complex and difficult features of CPL, and these difficulties were simplified by Ken Thompson, who named it 'B.'

Later, while developing the UNIX operating system in 1969, Ken Thompson 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 C. Of course, the development of UNIX using the C language made it uniquely portable and improvable. Finally, C was released in 1972. Dennis Ritchie was a research scientist at Bell Telephone Laboratories in the U.S.A. Brian Kernighan also participated in making the definitive description of the language. Therefore, C is also referred to as 'K&R C.'

What are instruction, statement and program?

An instruction is a command given to the computer processor to perform a certain elementary task.

For example, consider a simple addition instruction: $10 + 20$. When this instruction is executed, we get 30.

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

For example, consider calculating the area of a circle: `area=22/7*radius*radius`.

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

Software can be defined as a collection of one or more programs. The following is an example of a program that 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, a programming language is communication between the user and the computer. The user communicates with the computer by giving instructions, which are written based on the language's syntax structure provided by the language manufacturers.

We have several languages, such as Fortran, Pascal, COBOL, C, etc. Out of all, C is a rich, simple, elegant, powerful, and structured language. Nowadays, C has become the primary basis for advanced languages like C++, Java, C#, VC, and 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 a first-generation language, used during the development of computers in the 19th century. In this language, the data and instructions of programs were written in terms of binary symbols (1s and 0s); 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 represent the code for '+' and the remaining 8 bits are 9 and 13. This language is also called binary or low-level language. Actually, this language is difficult to understand and remember, and writing programs in binary codes leads to a lot of typing mistakes because everything is in 1s and 0s. This problem led to the invention of assembly language.

B) Assembly Language

This is somewhat better than binary language. Here, symbolic names are provided for every binary code of machine language. The names, such as add, sub, mul, div, cmp, etc., allow programmers to easily understand and write instructions using these names instead of binary 1s and 0s. For example, adding 4 and 6 in 8088 assembly language as

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

However, the main drawback of assembly language is that a large set of instructions needs to be written for simple tasks, like the addition of two numbers as said above. It supports only small programs if the code is less than 1000 lines; otherwise, it becomes difficult to manage the code. Moreover, assembly differs from one machine to another. Some people consider assembly language also a machine language since the assembly code resembles the machine code.

C) High Level Language

This is an English and mathematics oriented language. Here, data and instructions are composed in terms of English words and mathematical expressions. For example, $c = a + b$, $c = a - b$, $\text{if } (a < b)$, etc. So, one can easily learn and apply high-level language code. One can write programs without the knowledge of computer hardware and operating systems, i.e., it hides (abstracts) the underlying details about the instructions and how they are being processed in the machine. Thus, programmers can easily code without worrying about hardware and the OS. The 8085 and 8086 are known as assembly languages, whereas C, C++, and 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 that translates assembly language program codes into equivalent machine understandable codes (processor understandable code). Similarly, a compiler translates high-level language code into equivalent machine understandable codes. We have several operating systems and several C compilers. Almost all compilers follow the same rules, except for a few modifications.

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 Giuseppe 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.

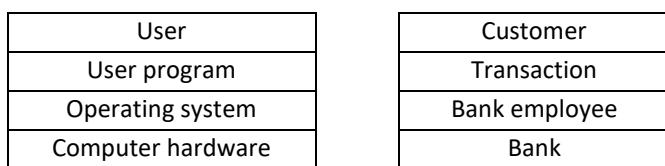
Operating System

The term O.S. refers to a set of programs that manage the resources of a computer. The resources of a computer include the processor, main memory, disks, and other devices such as the keyboard, monitor, and 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. Thus, the interface hides the underlying workings of a computer. The main jobs of the O.S. are memory management, process management, disk management, I/O management, security, and providing an 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.



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.

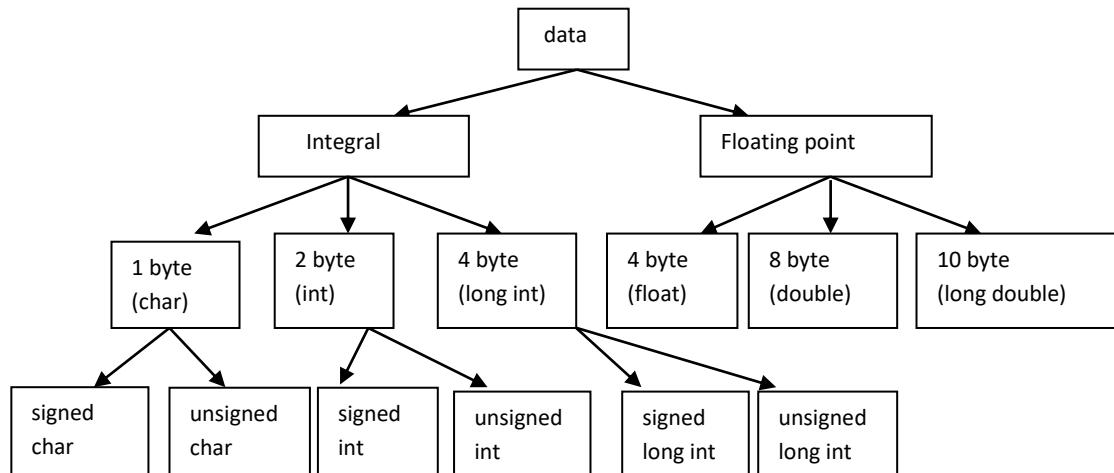
The computer hardware (processor) designed to process only two types of data

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

Integral data consists of rounded values such as 10, 20, -343, etc., whereas floating-point values include numbers like 15.44, -45.56, etc. In the real world, most data falls under these two types. For example, an employee number or age would typically be integral types, while basic salary and net salary are often stored in floating-point format.

Employee names and addresses, on the other hand, are collections of alphabets and other symbols. Interestingly, these are also treated as integer-type values because, in computers, alphabets are stored as their ASCII codes—for example, 'A' as 65, 'B' as 66, and 'a' as 97.

In C, these data types are classified into nine categories based on the application's requirements and hardware support. These categories are known as built-in, 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

$$a + b, \quad a - b, \quad a * b, \quad a / b, \quad a \% b$$

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 ($15.0/2 \rightarrow 7.5$)

The modulus-division gives the remainder of a division, for example $17\%3 \rightarrow 2$. 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 conditions 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 condition 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 OR(||) 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 programming, you might mistakenly think of it as the “equal” comparison operator used in mathematics. However, in C, it is used differently—to copy the value from one location to another.

It is used to assign (copy) the value of the right-hand-side (RHS) expression to the left-hand-side (LHS) variable; i.e., it copies the RHS value into the LHS variable.

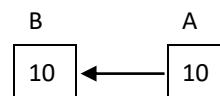
Syntax 1: **variable=expression;**

Syntax 2: **variable1 = variable2 = variable3 =... variableN = expression;**

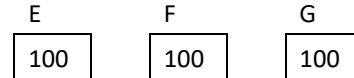
A = 10; //assigns 10 to ‘A’ (puts 10 in A’s memory)



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 to 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 follows.

`++` 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 means: increment & substitute its value**
- **post-increment means: substitute & increment its value**

Note1: the parenthesis () has no effect on these operators. For example: `2*(A++)` is equal to `2*A++`.

Note2: these operators cannot be applied to expressions, for example `(2*A)++` gives a syntax error.

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*2; // this is as given below`
`++A; // here A becomes 6`
`B=A*2; // here B becomes 12`
- 5) `B = 2*A++; // this is as given below`
`B=A*2; // here B becomes 10`
`A++; // here A becomes 6`
- 6) Let A=1; observe following substitutions.

$$\begin{aligned} B &= ++A*2 + ++A*3 + A++*4 + A++*5 + ++A; \\ &2*2 + 3*3 + 3*4 + 4*5 + 6 \end{aligned}$$

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 by combining mathematical or logical operators. While evaluating such expressions, sub-expressions are initially evaluated according to their precedence.

For example, the expression $5 + 2 * 4$ is evaluated as $5 + 8$, resulting in 13.

Observe the following table showing the relative precedence of operators in C. Take a moment to review this precedence, as we will explore it in detail in the following 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, which should be handled carefully; otherwise, they may lead to logical errors. Let us see how an expression should be written with the correct syntax.

For example:

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

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

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

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

Variable and its Naming rules

A program is a collection of data and instructions. Data is represented in terms of names such as *age*, *experience*, *salary*, etc. These names are called variables, which refer to or hold values in the program.

Thus, a variable is a name given to a memory location in the computer's main memory, where certain values are stored and retrieved. These values may change during the execution of the program.

Rules for Naming Variables: While naming a variable, we should follow the rules 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

A declaration specifies the name, size, type, and other characteristics of a variable. Before using any variable, it must be introduced to the compiler by specifying the name, size, and type of data it will hold. This introduction is called variable declaration. It involves creating memory in RAM and assigning (binding) a logical address that will be used wherever the variable appears 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 Garbage value 2 bytes	k2 10 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 a variable at the time of its declaration, i.e., at the moment the memory space is created.

This is called *initialization* of the variable. Assigning a value after the declaration is referred to as *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 a variable is not initialized with any value, it holds an unknown value by default, called a garbage value. This value can be positive, negative, or sometimes zero.

int X=100, Y;	X 100	Y Garbage
---------------	-----------------	---------------------

Here X contains 100, but Y contains garbage value.

After a program finishes execution, its binary code or data is not immediately erased/cleaned from memory by the operating system. It remains in memory until another program is loaded into the same space.

When a new program loads into this memory, the old program's code and data are simply overwritten.

However, when memory is allocated for variables in the new program, the locations may still contain leftover binary values from the previous program. These leftover values are considered **garbage** by the new program.

Therefore in the above X,Y; the variable 'Y' contains a garbage value.

Sometimes, this garbage may even originate from the same program. For example, when a function terminates, its local variables' memory space is freed and can later be reallocated for other variables in another function.

'C' comment line

Comment lines are meaningful passages of text that provide information about the author of the program, the purpose of the program, and other relevant details. Adding comments is considered good programming practice, as it enhances the readability of the code. Programmers often include comments to explain complex instructions, making the program's logic easier to understand.

Comments can be written anywhere in the program by enclosing the text within a pair of /* ... */.

syntax:

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

example:

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

These comment lines are ignored (skipped) while compiling a program and, therefore, are not executed. Comments are not part of the program; they are meant for documentation purposes. This is a C-style comment, and it can be used as either a single-line or multi-line comment in the program.

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

```
// .....comment line1.....
// .....comment line2.....
// .....comment line3.....
```

So, in C++, every comment line starts with the symbol //.

Since C++ is a superset of C, we can compile C programs using a C++ compiler. As a result, C++-style comment lines (//) can also be used in C programs. (I have personally used these comments in my explanations.)

Identifier

Whatever name is specified by the programmer is called an identifier. An identifier is a name given by the programmer to elements such as variables, functions, macros, structures, etc.

The rules for specifying an identifier are the same as the rules for naming a variable (as mentioned above).

- ✓ 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.

Functions

In general, in our daily lives, several people are often required to accomplish a single task, with each person focusing on their specialized sub-task. Similarly, a single programmer usually does not develop the entire code for an application alone; instead, multiple programmers contribute by writing sub-programs called **functions**. A **function** is a sub-program designed to perform a specific task in a program. Each function is written for a particular purpose, such as calculating powers, square roots, or logarithms. A collection of such functions forms a complete C program. For example, `sqrt()` is a function that calculates the square root of a number. For instance, `sqrt(16)` returns 4.

Functions in C are classified into: 1. **Library functions** 2. **User-defined functions**

Library functions

These are ready-made functions, designed and written by the C manufacturers, to provide solutions for basic and routine programming tasks. For example, mathematical functions like `pow()` and `sqrt()`, and I/O functions like `scanf()` and `printf()`, are such functions.

The vendors of C supply these predefined functions in compiled form along with the C software. Since there is a vast collection of such compiled functions available in C, this collection is referred to as the **functions library**.

The bodies of these library functions are stored in separate files, typically with names like `xxxx.lib`, in compiled format. These functions are automatically linked to our program during compilation and remain hidden from the programmer. (We will discuss user-defined functions in the chapter on functions.)

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>`

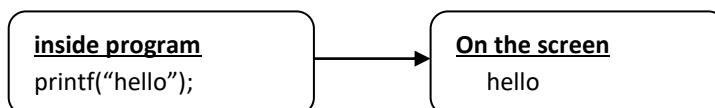
2) Terminal I/O library functions

Generally, all programs need to interact with keyboard and monitor devices to accept and display values. Input is taken from the keyboard, and the output is displayed on the screen.

The `printf()` and `scanf()` functions are well-known I/O functions defined in the `stdio.h` header file.

printf(): Displays messages or output values on the screen. Programs are stored permanently in secondary storage devices like a hard disk, and when they are executed, they are loaded from the hard disk into RAM. The processor then executes them instruction by instruction.

By default, nothing is displayed on the screen while a program is running in memory. If we want something to be shown on the screen, we must add an instruction such as `printf()`, which displays data or messages on the screen. For example:



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

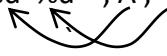
the format strings are: `%d` , `%f` , `%ld` , `%u` , `%c` , `%s` , etc.

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 a width to the format strings, for example: `%5d`, `%7d`, `%05d`, `%f`, `%.2f`, `%05.2f`, etc.

`%5d` → Here, 5 specifies the maximum width of the output value to be displayed on the screen.

If the width is greater than the size of the output value, spaces are added by the `printf()` statement.

If the width is less than the number of digits in the value, no spaces are added; the value is printed normally (width is ignored in this case). **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);` → BBBB823.400

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)

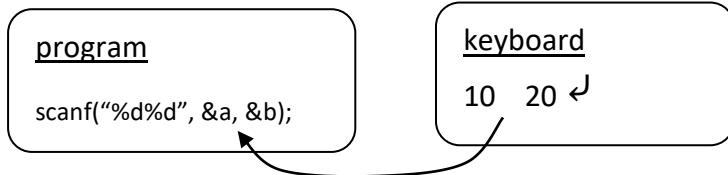
`%ld` → signed long decimal integer

`%lu` → unsigned long decimal integer

`%lf` → double precision floating point number (double)

scanf(): used to read input values from the keyboard. While running a program, the user must enter values through the keyboard. Based on these input values, the output is displayed. The `scanf()` function waits for the user's response until they enter some values from the keyboard—that is, it pauses program execution until the user provides input and presses the **Enter** key. When the **Enter** key is pressed, the program resumes execution.

For example:



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()` function serves as both the starting and ending point of a program. Execution begins at `main()` and ends at its closing brace. In fact, `main()` is also a function; among all functions, it is special, important, and reserved, as it indicates to the compiler where the program starts and ends.

The `main()` function can be written in any of the following ways, all of which have the same meaning and work in a similar way (although different C vendors have suggested different styles for this).

<code>void main () { ----- ----- ----- }</code>	<code>main() { ---- ---- ---- }</code>	<code>int main () { ----- ----- return(0); }</code>
---	--	---

Therefore, in a C program there must be at least one function, namely `main()`; its execution started by the operating system when the user runs the program. In other words, the `main` function is called by the operating system, indirectly on behalf of the user—for example, when you press **F9** or **F10** in an IDE.

General Structure of C program

The C program looks like following way

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

line1: In this line, the comments are added about program, for example, the purpose of program, input/output, who wrote it, and when it was written. Writing comments is a good programming practice. However, this is optional. The compiler ignores comment lines.

Line 2: This line includes header files for library functions and other files. For example: #include<math.h> for math functions, and #include<stdio.h> for input/output functions, etc.

Note: These files contain the declarations (syntax) of library functions, which allow the compiler to check for errors in function call statements.

line3: int main() represents the starting point of the program. It indicates that the main process begins here.

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

line5: Input statements are used to accept values from the keyboard or other input devices.

line6: Here, process statements specify how the input data is processed.

line7: Here, output statements are used to display results on the screen or other output devices.

Note: The opening and closing braces {} define a block. As per C syntax rules, instructions must be written inside a block, and each instruction must be terminated with a semicolon (;).

Coding style of C program

Unlike some other programming languages (cobol, fortran, etc.), C gives programmers the freedom to follow their own coding style. However, it is important to ensure that the program remains readable and easy for others to debug. For example, consider 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 improve readability, we can add extra spaces between expressions, that is, we can insert spaces before and after every token, such as punctuation symbols, operators, variables, and other elements.

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
int 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>
int main()
{
    float A,B,C;
    A=10.5;
    B=20.3;
    C=A+B;
    printf(" output is %f ", C); // for float-type values use %
}
```

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

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

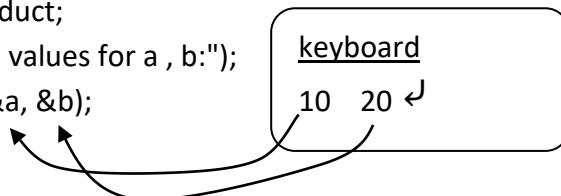
4) Finding sum and product of given two input values

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

ip: 10 20

op: sum is 30, product is 200

```
#include<stdio.h>
int 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 printf() prompts the user to enter two values for a and b.

scanf("%d%d", &a, &b); → This scanf() reads two values from the keyboard and stores in 'a' and 'b'.

For two values, we use two %d format specifiers. We will see later why %d is used.

printf("sum is %d, product is %d", sum , product); → this display output 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 (\sqrt{x})

```
#include<math.h> // when we use pow() or sqrt() function, we have to include this "math.h" file
#include<stdio.h> // we use printf() or scanf(), we have to include this "stdio.h" file
int main()
{
    int A, B;
    A=pow(2,10); // A=210
    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: $5*x*x*x + 2*x+10 \rightarrow 5*3*3*3+2*3+10 \rightarrow 135 + 6 + 10 \rightarrow 151$

```
#include<stdio.h>
int main()
{
    int x, result;
    printf("enter x value :");
    scanf("%d", &x); // we can't write X3 in the program, we need to write as x*x*x
    result=5*x*x*x+2*x+10;
    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>
int main()
{
    float C, F;
    printf("Enter Celsius:");
    scanf("%f", &C); // we can't write C in the formula, we need to write as %f
    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>
int main()
{
    float radius, area, circum;
    printf("Enter radius of circle :");
    scanf("%f", &radius); ←----- keyboard -----  
Enter Radius : 5 ↵
    area=3.14 * radius * radius;
    circum=2 * 3.14 * radius;
    printf(" circle area is %f , circumference is %f", area , circum );
}
```

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

2345/1000 → 2

(234/10)%10 → (23)%10 → 3

2345%100 → 45

2345%1000 → 456

(234%100)/10 → (34)/10 → 3

2345/10 → 234

234/1000 → 0

23%10 → 3

2345%10 → 5

27/10 → 2

7%10 → 7

```
int 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)

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

op: 9650 seconds

op: 170 seconds

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

```
int 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 by 3600 and take the quotient as hours (since 1 hour = 3600 seconds).

Now divide N by 3600 and take the remainder after extracting the hours: R = N % 3600

This R represents the leftover seconds after taking hours from N.

Next, divide R by 60 and obtain both the quotient and remainder.

The quotient will be the minutes, and the remainder will be the seconds.

```
int 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;

```
int 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 the values of two variables can be exchanged with each other.

This is a common task in programming and is also known as swapping of data.

Logic: Let the input values be 23 and 45, assigned to variables X and Y respectively. Consider what happens if the code is written as:

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

When $X = Y$ is executed, the value of X is replaced by that of Y, so the original value of X (23) is lost. Now both X and Y contain the same value, 45. When the second instruction $Y = X$ executes, the same value, 45, is copied back to Y again. As a result, both variables hold the same value, and the original purpose of swapping is not achieved.

```
int 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 $\rightarrow 7*100 + 3*10 + 5*1$.

Similarly, the reverse of it is $\rightarrow 5*100 + 3*10 + 7*1$

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

```
int 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);  
}
```

$$\begin{array}{r} 10) 47 (4 \rightarrow n/10 \\ \quad\quad\quad 40 \\ \hline \quad\quad\quad 7 \rightarrow n \% 10 \end{array}$$

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

$$\begin{aligned} \text{reverse} &= n \% 10 * 10 + n / 10; \\ &= 47 \% 10 * 10 + 47 / 10 \\ &= 7 * 10 + 4; \\ &= 74 \end{aligned}$$

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

```
int 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. eg: sizeof(int) → 2

```
int 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 into the system before it can be executed. To feed the program into the computer, a software tool is needed that allows typing the instructions, as well as modifying, deleting, copying, and other editing functions. This is provided by a special software called an **Editor**. There are several editors available for entering (typing) the program into the computer, such as Notepad, WordPad, MS Word, etc.

Notepad, for example, allows only typing and saving the code but does not provide functionality to compile or run the programs. Fortunately, today, most programming languages come with a special editor that includes features for compiling, debugging, and running programs. These features are integrated into a single tool, which is known as an **Integrated Development Environment (IDE)**

The screenshot shows the Turbo C++ IDE interface. The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. The title bar displays 'NONAME00.C'. The code editor window contains the following C code:

```
#include<stdio.h>
void main()
{
    printf("welcome to C-Family computers");
    printf("Have a nice day");
}
```

The status bar at the bottom shows keyboard shortcuts: F1 Help, F2 Save, F3 Open, Alt-F9 Compile, F9 Make, F10 Menu.

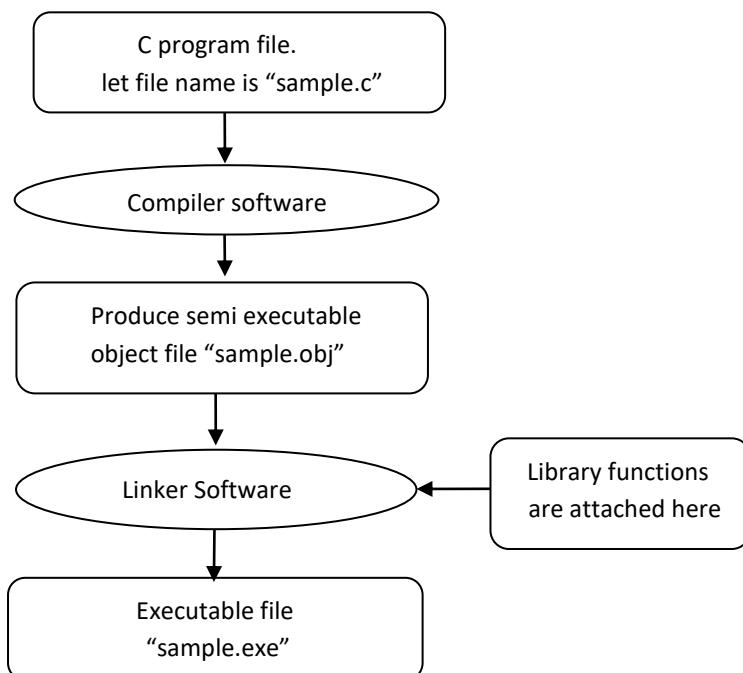
File menu: this menu-driven facility used to handle file operations such as opening a new file, or opening an existing file, saving a file, etc.

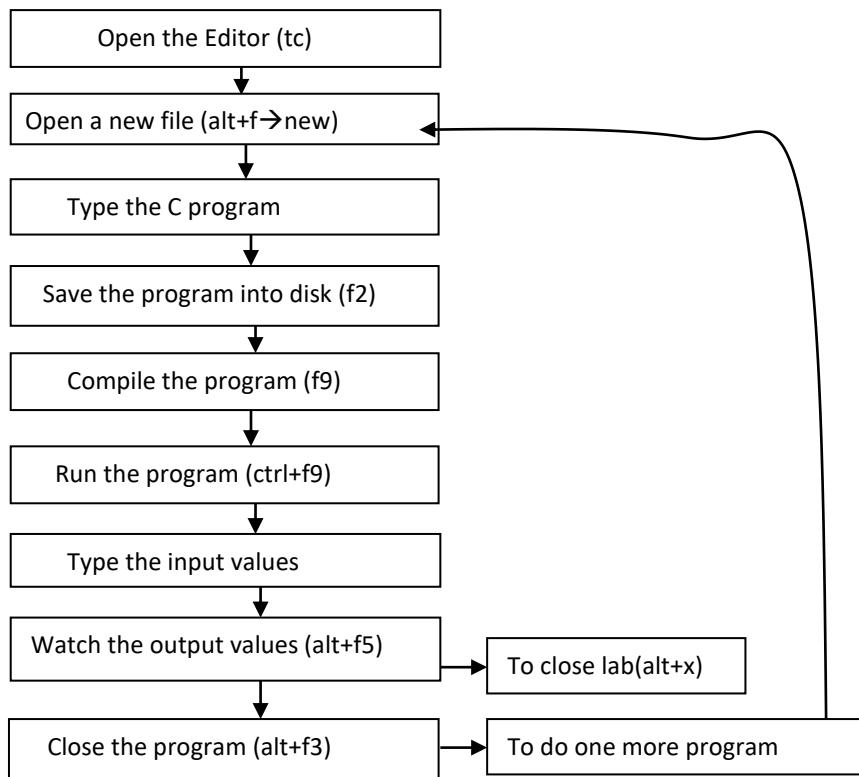
Edit menu: this is used to copy, cut, and paste a group of selected lines.

Compile menu: to create object files, making executable files, linking a program....

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

During the compilation stage, the compiler checks for typing and syntax errors. After correcting them, it creates an object file, which is a semi-executable file. In this stage, each function is compiled independently, and the code is generated separately as it appears in the source file. Later, the **linker** links all library and user-defined functions together to form a single executable file. For example, the main() function and printf() function, as shown in the previous examples. The following figure illustrates how these functions are compiled and linked into the program.



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

C) Compiler vs Interpreter

Interpreter is a translator that translates and executes a 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. Therefore, it is also called an instant compiler. Whenever we run the program, the interpreter translates and executes it. If we run the program more times, it will translate the same code each time. This seems unnecessary translating same program again & again in every use. This redundant translation leads to wastage of CPU time and other resources.

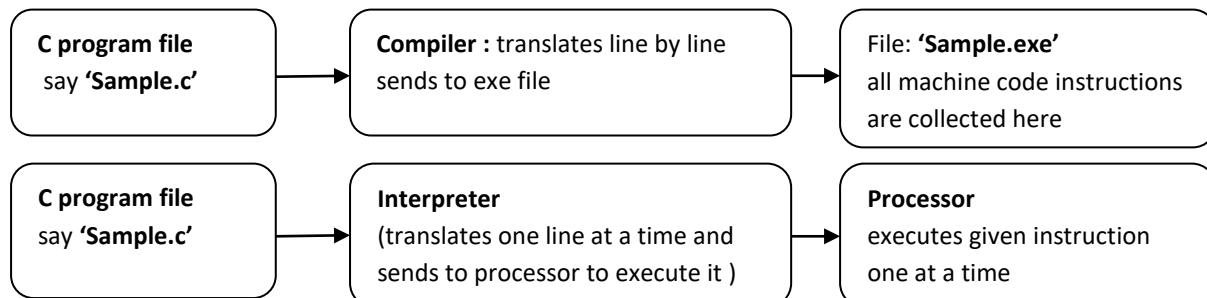
Actually, an interpreter is used for single-use applications like queries, commands, spellings in word processing, language translators, internet browsing, etc. The web-sites are coded using languages like html, css, etc and these files are downloaded and interpreted by browsers. Web browsers are nothing but interpreters. Therefore, for single use applications, an interpreter is the preferred choice.

Compiler translates the entire program at once and stores it in an ".exe" file for later execution.

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 needed, without requiring the compiler again. As a result, no translation takes place each time the program is run. In contrast, an interpreter does not create an ".exe" file, so the program must be translated each time it is executed.

Some applications, such as calculators, business accounting software, and games, need to be executed regularly. In these cases, the **compiler** is used to generate a machine code file so that it can be executed multiple times without repeated translation.

Conclusion: A compiler is the choice for programs that are used regularly, while an interpreter is ideal for single-use code. For single-use applications, creating a machine code file is unnecessary.



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. The escape sequence characters are

1. \n → new line (new line)
2. \t → horizontal tab space (4 gaps)
3. \\ → single back slash (\), inserts back-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 as type casting. In C, the type casting is done in two ways. ①**Implicit type casting (automatic casting)** ②**Explicit type casting (manual casting)**

① Implicit Type Casting

For example, in the expression $10 + 20.54$ (int + float), the value 10 is automatically converted from an integer to a float (10 to 10.00). This is because we must hand over the same size and type of values to the processor before computing. The processor performs operations bit by bit on two values, so they need to be of the same size and type. The compiler automatically handles this conversion when the types in the expression do not match. In this way, the compiler promotes lower-type values to higher-type values within 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 will be converted into 20.00 by the compiler implicitly

eg2: `10 * 20.00;` // here the value 10 will be converted into 10.00 by the compiler implicitly

eg3: `int x = 10;`
`float y;`
`y = x;` // Here 'x' value 10 will be promoted to 10.00 and stored into 'y'

eg4: `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)
// some C++ compilers does not allow this direct assignments where we need written as: `x=(int)y;`

eg5: `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 value of Y
is assigned to X, so results loss of some bits. (so proper value will not be stored in X)
// some new c compilers does not allow this type of direct assignments where we need to write as `X=(int)Y;`

eg6: `int main()`
{ `float k;`
 `int x=10, y=3;`
 `k=5+x/y;`
 `printf("\n Result of K = %.2f ", k);` // Result of K=8.00
}

In this expression, x/y gives an integer result of 3 (not 3.33) because both x and y are of type int. This result is then added to 5, which is also an integer, and the final result is implicitly converted to a float before being assigned to k. Therefore, the output is: Result of K = 8.00.

To get the exact value of the division, explicit casting is the solution, as shown in the examples below.

② Explicit 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: int main()

```
{ int x=10, y=3;
  float k;
  k = 5 + x/y;
  printf("\n Before type casting K=%f ", k );
  k = 5 + (float) x/y+5; // here x is converting by us, whereas y will be converted by the compiler.
  printf("\n After type casting K= %f ", k );
}
```

Before type casting K=8.000000

After type casting K= 8.333333

eg3: int main()

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

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

After type casting K= 600010

In the expression K=10+X*Y, the product of X*Y is first calculated and stored in a temporary variable before being added to 10. This temporary variable is a nameless variable automatically created by the compiler (let's call it T).

Then the instruction K=10+X*Y is executed as: T=X*Y; K=10+T;

Here, X and Y are of type int, so T will also be of type int. However, if the result of X*Y exceeds the range of an int, it cannot be stored in T, causing a loss of some bits due to overflow.

The solution is to write the expression as K = 10 + (long int) X*Y. Here, we explicitly convert the X to long int, and the compiler automatically converts b to long int. As a result, T will be long-int, which can safely store the X*Y value.

Changing a lower type to a higher type is called *promotion (expanding)*, while the reverse is called *demotion (narrowing)*.

Representing constants in C++

In programming, data is represented in two ways: as constants and variables. Constants are represented in a special way by adding format strings or other symbols to their values. For example, 10L represents a long integer, and 10F represents a float.

The compiler automatically recognizes certain types of constants in the program. For example, the constant 10 is considered an int-type, 34.55 as a double-type, and 'A' as a char-type. These are default types automatically understood by the compiler. However, some constant types need to be specified explicitly using format strings. The following list explains the different types of constants. Avoid using symbols like commas, spaces, or quotation marks when specifying constants.

integer constant (int constant)

A collection of one or more digits with or without a sign referred to as signed int constants. This is the default type for integral values. eg. 5, 256, 9113, 6284, +25, -62, etc are valid signed integers.

Similarly, to represent unsigned int, the format string 'u' or 'U' is suffixed to the value.
eg. 67U, 43U, 4399u, 45u are valid unsigned integers.

Note that, -3428u is also a valid number. Here this -3428 is converted into equivalent unsigned integer to 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, 0xABCD, 0xfc 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 2.32×10^5 i.e. 232000.0)

12 . 102e-3f (means 12.102×10^{-3} i.e., 0.012102)

-2 . 165e6f (means -2.165×10^6 i.e., -2165000.0)

3 . 68e3f (means 3.68×10^3 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 ('I' or 'L' is option for double)

long double: for the long double, the format string 'l' or 'L' is used.

eg: 3.14L

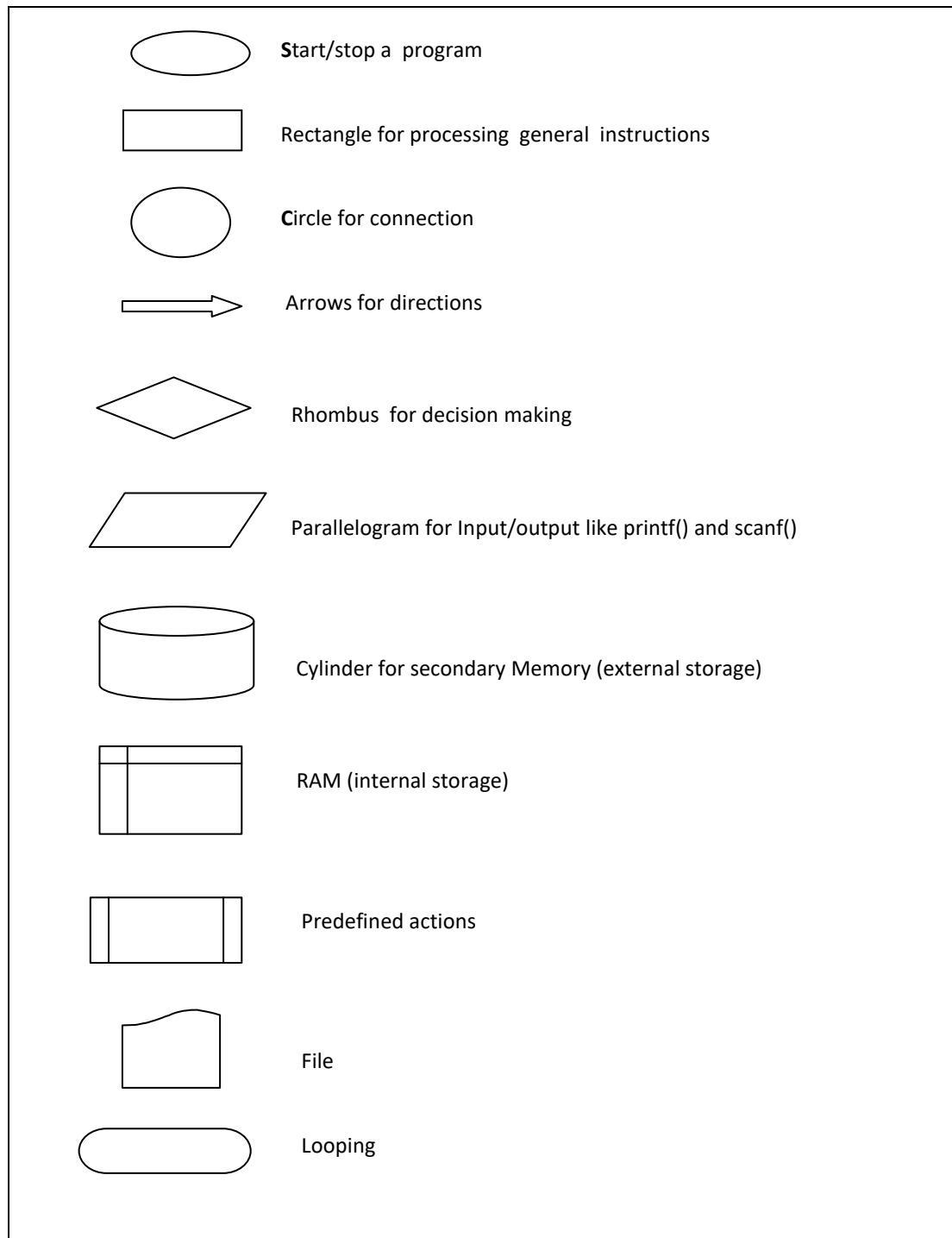
314.41 (Equivalent to 314. 0L)

3.68e3L (Means 3.68×10^3 i.e., 3680.0L)

Flow chart

It is a pictorial representation of the flow of a program, illustrating the sequence of operations to be performed to obtain the solution to a problem. It is often used as a visual planning tool to show how the control moves from one point to another to reach the solution. It is similar to drawing a building plan before constructing the building.

Flowcharts are generally drawn before constructing a program or algorithm. They facilitate communication between programmers and business people and play a vital role in understanding the logic of complicated and lengthy problems. Once a flowchart is drawn, it becomes easier to write the program in any high-level language. The following mathematical symbols are used to represent specifications, with directions indicating the flow of control.



Algorithm

In mathematics and computer science, an algorithm is a finite set of computational instructions that carry out a particular task, taking input and producing the desired output. In simple terms, it is a step-by-step explanation of the logic used to construct instructions in a program, where English words and mathematical symbols are used to express the logic.

A flowchart depicts how the control moves within a program from one point to another to reach a solution, whereas an algorithm provides a step-by-step explanation of how to construct the instructions in a program.

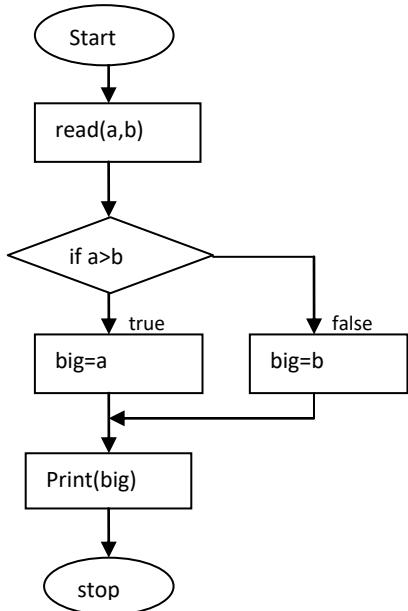
This is an independent method of explaining logic, regardless of the programming language used by the programmer. English words and mathematical symbols are used to represent the logic. Algorithms are often written for complex and complicated tasks to describe the logic in words, such as sorting algorithms, searching algorithms, and shortest-path algorithms in a network. 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

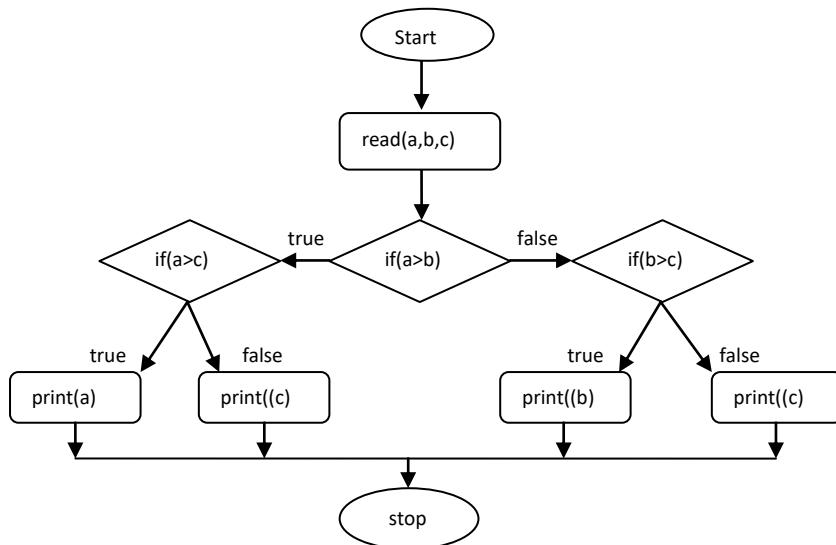


Algorithm to find big of two numbers

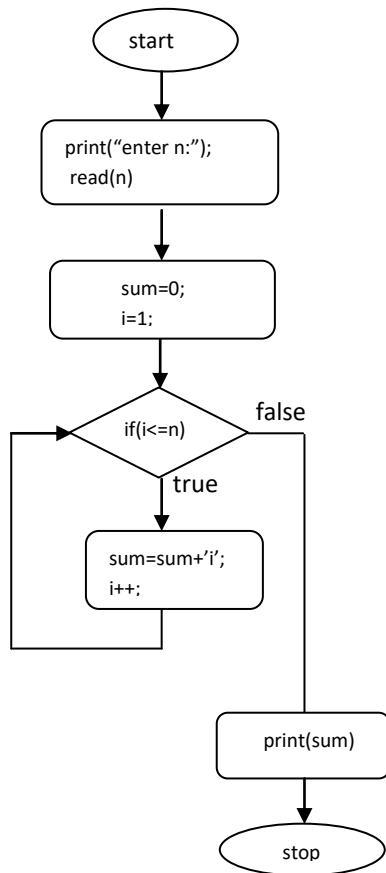
```

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($1+2+3+4\dots+n$)



Pseudo code

Pseudo code is a shorthand way of describing a program or algorithm using a general language syntax rather than a specific programming language syntax. An algorithm uses English and mathematical symbols to explain logic step-by-step, whereas pseudo code uses programming language syntax to explain logic in terms of instructions. Therefore, pseudo code resembles a program with a general language syntax. Most people adopt C or Pascal language's syntax and their control structures to describe pseudo code.

An algorithm is understandable by any non-programmer, but pseudo code is understandable by a person who knows at least one programming language. **It is a method used to explain the logic between two programmers, making it easier for them to understand the general workings of a program written by another programmer.**

A flowchart shows only the execution flow (directions). An algorithm explains what steps to take to get a solution, but it does not give a clear explanation of how to write instructions. For example, to swap two values in an algorithm, we might write as: swap(a , b). In pseudo code, we would write temp=a, a=b, b=temp. Thus, in pseudo code, every step is written in detail, which is closer to actual programming. Loosely, one could say that pseudo code is a program without accurate syntax (syntax is ignored).

[Flowchart→Algorithm→ Pseudo code→Program]

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

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 $\leq n$
 sum=sum+i;
 i++;
5. [printing output]
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($i \leq n$)
 - { sum=sum+i;
 - i=i+1;
 - }
6. print(sum);
7. stop;

Control Statements

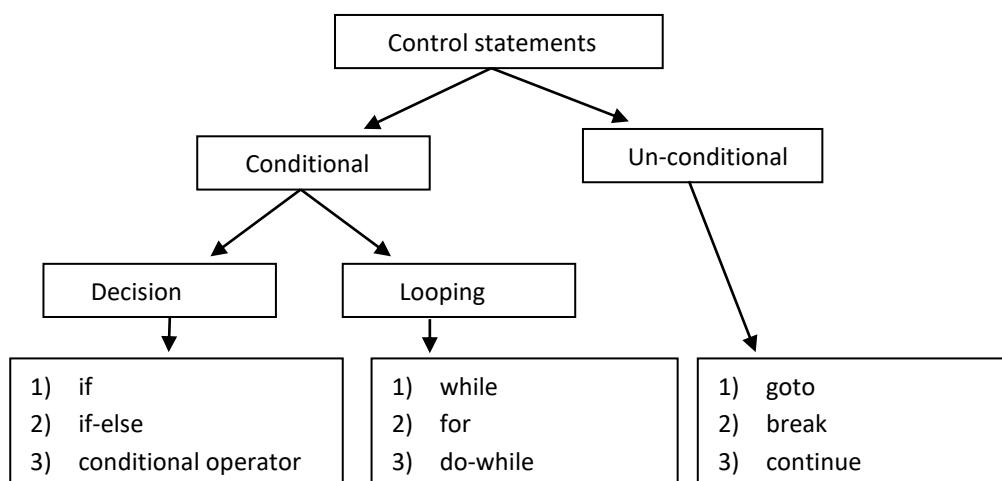
Generally, instructions in a program are executed sequentially, one after the other, in the order in which they appear in the program. However, in practice, we may need to bypass certain instructions, repeatedly execute a set of instructions, or alter the sequence of execution depending on the requirement.

For this purpose, special statements are necessary in programming to control the flow accordingly.

C provides such handy control statements as if, if-else, switch, while, etc. These statements alter the normal execution sequence of a program according to our needs. Therefore, they are called control statements.

These control-handling statements are of two types.

① Conditional control statements ② Unconditional control statements



Decision control statements: are used to execute or skip some set of instructions based on given condition.

Loop statements: are used to execute some set of instructions repeatedly until a given condition is failed.

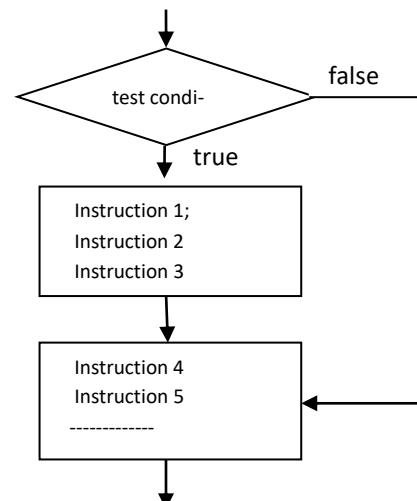
Condition-less control statements: are used to transfer the control unconditionally from one location to another location.

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 successful (true), then instruction1, instruction2, and instruction3 will be executed. Afterwards, the control comes out of the if-body and proceeds with instruction4, instruction5, and so on. This is shown in the flowchart.

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, this program demonstrates how an if-statement executes.

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

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

op: hello bye

In the above example, the first condition ($10 > 5$) is true, so `printf("hello")` will be executed.

The second condition ($10 < 5$) is false, so `printf("hai")` will not be executed.

The final `printf("bye")` executes irrespective of the if statement.

Demo2, this program demonstrates how an if-statement executes

```
int main()
{
    if( 1 < 2 )
    {
        printf(" Red1 ");
    }

    printf("Red2 ");

    if( 1 > 2 )
    {
        printf(" Blue1 ");
    }

    printf(" Blue2 ");

    if( 1 == 1 )
    {
        printf(" Green1 ");
    }

    printf(" Green2 ");

    if( 1 != 1 )
    {
        printf(" Yellow1 ");
    }

    printf(" Yellow2 ");
}
```

op: Red1 Red2 Blue2 Green1 Green2 Yellow2

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.

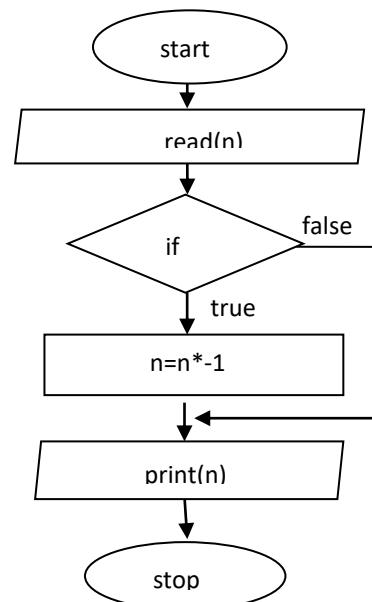
If user entered -ve value, converts it into +ve by multiplying it with -1.

ip: -14	ip: 15
op: 14	op: 15

```

int 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 the difference between two numbers, output is +ve.

ip: 12 18	ip: 18 12
op: 6	op: 6

```

int 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 arrange 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

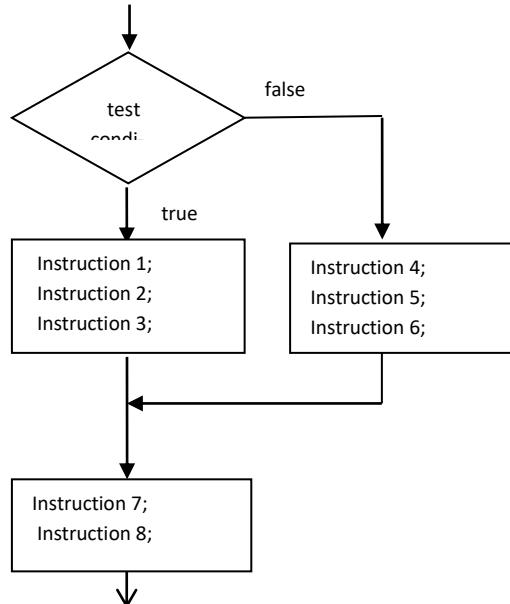
This is a two-way decision control statement that executes either the if-block or the else-block based on the given condition. If the condition is true, the if-block executes; otherwise, the else-block executes.

Let us see the syntax.

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

// if-block

// else-block



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

```
int 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

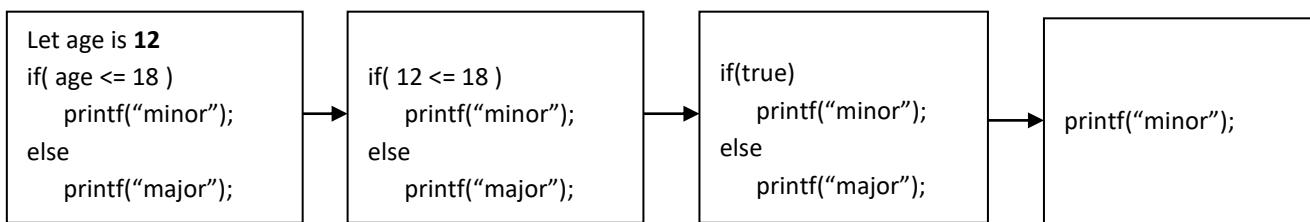
Finding a person is minor or major

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

int main()
{
    int age;
    printf("Enter the age:");
    scanf("%d", &age);
    if(age<=18)
    {
        printf("minor");
    }
    else
    {
        printf("major");
    }
}
```



Finding given number is odd/even

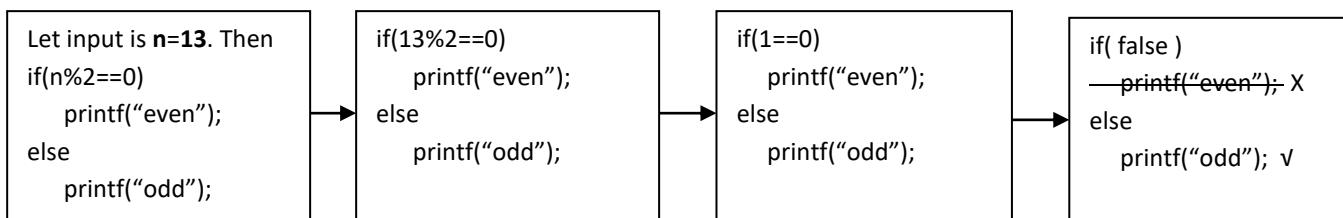
This program accepts a number from keyboard and finds whether it is odd or even?

```
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.

```
int 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



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.

<pre>if(A>B) { printf("Red"); } printf("Blue"); printf("Green"); -----</pre>	<p>// code after removing unnecessary braces</p>  <pre>if(A>B) printf("Red"); printf("Blue"); printf("Green"); -----</pre>
<pre>if(A>B) { printf("Red"); } else { printf("Blue"); } printf("Green"); -----</pre>	<p>// code after removing unnecessary braces</p>  <pre>if(A>B) printf("Red"); else printf("Blue"); printf("Green"); -----</pre>
<pre>if(A>B) { printf("Red"); } else { printf("Blue"); printf("Green"); } printf("White"); -----</pre>	<p>// code after removing unnecessary braces</p>  <pre>if(A>B) printf("Red"); else { printf("Blue"); printf("Green"); } printf("White"); -----</pre>
<pre>if(A>B) { printf("Red"); printf("Blue"); } else { printf("Green"); } printf("White"); -----</pre>	<p>// code after removing unnecessary braces</p>  <pre>if(A>B) { printf("Red"); printf("Blue"); } else printf("Green"); printf("White"); -----</pre>
<pre>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"); }</pre>	<p>// code after removing unnecessary braces</p>  <pre>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"); }</pre>

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.

<pre>if(A>B) printf("Red"); → with indentation printf("Blue"); printf("Green"); -----</pre>	<pre>if(A>B) printf("Red"); → without indentation (makes confusion) printf("Blue"); printf("Green"); -----</pre>
<pre>if(A>B) printf("Red"); → with indentation else printf("Blue"); → with indentation printf("Green"); -----</pre>	<pre>if(A>B) printf("Red"); → without indentation (makes confusion) else printf("Blue"); → without indentation (makes confusion) printf("Green"); -----</pre>
<pre>if(A>B) { printf("Red"); → with indentation printf("Blue"); } printf("Green"); -----</pre>	<pre>if(A>B) { printf("Red"); → without indentation (little confusion) printf("Blue"); } printf("Green"); -----</pre>
with indentation, no confusion <pre>if(a==10) { printf("red"); if(b==20) printf("green"); else { printf("blue"); printf("white"); } printf("black"); }</pre>	Code without indentation, here lot of confusion <pre>if(a==10) { printf("red"); if(b==20) printf("green"); else { printf("blue"); printf("white"); } printf("black"); }</pre>
something wrong in this code, fix it <pre>if(A<B) printf("one"); printf("two"); else printf("three"); printf("four"); // take into else-body printf("five"); // don't take into else-body</pre>	something wrong in this code, fix it <pre>a=4, b=4; if(a=b) // works as assignment operator, so write as if(a==b) printf("a, b are equal"); else printf("a,b are not equal");</pre>

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?

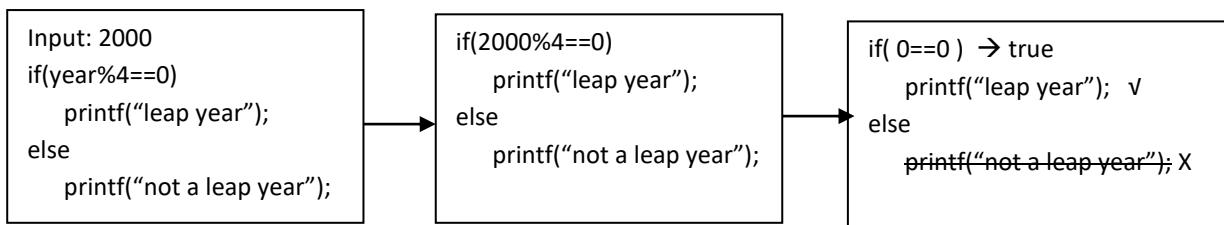
ip: 2000	ip: 2005
op: 2000 is a leap year	op: 2005 is not a leap year

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

```
int 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



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

```
int 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 it by multiplying with -1.

```
int 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 it 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 $9*10+2 \rightarrow 92$

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

```
int 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, using a count logic can simplify a program. We check if the input values are valid; if they are, we count them and then make a decision based on that count. In other cases, we might count the invalid values and make decisions based on that instead. Let's look at the 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: -ve not 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("-ve not found ");
else printf(" -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 or minutes >59 or hours > 12 then invalid-time (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 `bool=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: -ve not 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 94 30

op: valid inputs

op: invalid inputs

op: invalid inputs

Programs using Logical operators (`&&` , `||`)

Printing student is passed or failed in exam based on 2 subject marks

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 ($m1 \geq 50 \ \&\& \ m2 \geq 50$) 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 ($m1 \geq 50 \ \&\& \ m2 \geq 50 \ \&\& \ m3 \geq 50$) 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

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: “–ve 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
<pre>if(condition) { if(condition) instruction1; else instruction2; } else { if(condition) instruction3; else instruction4; }</pre>	<pre>if(condition) { if(condition) instruction1; } else { if(condition) instruction2; else instruction3; }</pre>	<pre>if(condition) { if(condition) instruction1; } else { if(condition) instruction2; }</pre>	<pre>if(condition) instruction1; else { if(condition) instruction2; else { if(condition) instruction3; else instruction4; } }</pre>

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
<pre>// simple nested-if style 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);</pre>	<pre>// nested-if with logical operators 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.</pre>

Demo finding student is passed or not? 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. But, this is not possible in all cases. The above code can be simplified using ‘&&’

```
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 is in between 10 & 20 or not?

```

int 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("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=D;
}

```

// 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;
}
}

```

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

here, if one condition is true, then rest of the conditions are bypassed(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");
```

```
int 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 $\text{root1} = (-b + \sqrt{x})/(2*a)$, $\text{root2} = (-b - \sqrt{x})/(2*a)$

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

if-block containing 2 instructions

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

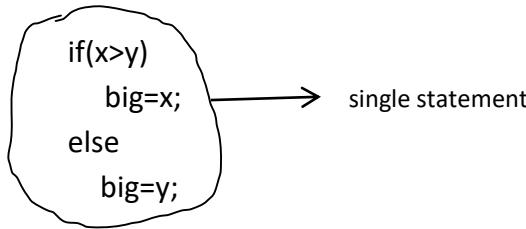
if age>19 and age<=50) then say younger and normal weight is 60

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 );
```

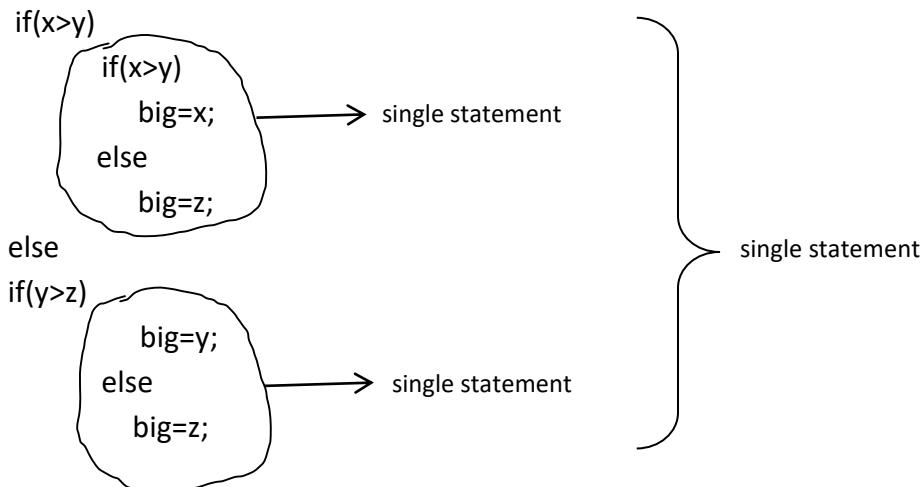
More about Pair of Braces { }

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



* As inner if-else statement is single, braces are not required to make into single for outer-if.

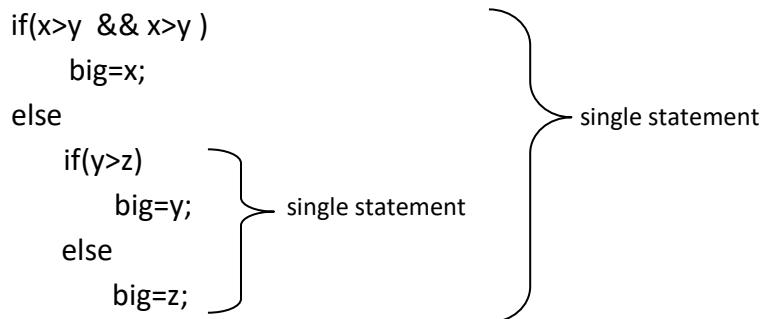
Based on this theory, we can remove braces for outer-if also. Let us see following example



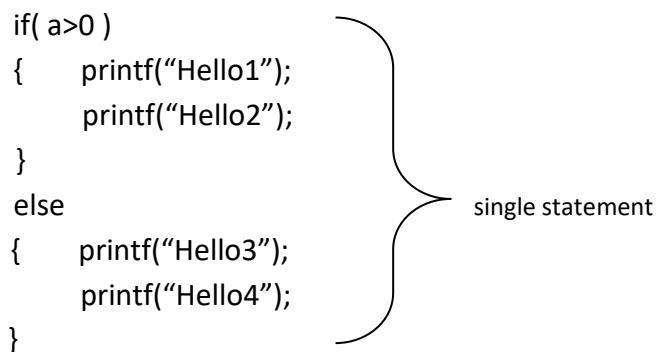
In this way, in C, any control statement can be taken as single compound statement including nested-if, thus above entire if-statement can be taken as single-compound-statement.

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

Example-3



Example-4



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);
```



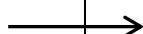
Demo program finding biggest of 4 numbers. Let A,B,C,D are 4 numbers

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-else-if ladder style
<pre>if(condition1) instruction 1; else { if(condition 2) instruction 2; else { if(condition 3) instruction 3 else { ----- } } }</pre>	<pre>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;</pre> <p>By removing un-necessary braces and arranging all nested-if statements in one column makes the if-else-if ladder style</p>

Here, the conditions are evaluated from top to bottom. 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 is executed. In other words, if all other condition tests fail, the final else instruction 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 cause some confusion for certain programmers. For this reason, the if-else-if ladder is also written as:

<pre>If(condition 1) instruction 1; else if(condition 2) instruction 2; else if(condition 3) instruction 3; ----- ----- else instruction n;</pre>
Observe here, the nested statements are started in a new line unlike same line as above said.

Accepting student marks of 3 subjects & printing result

If

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

Normal nested-if	if-else-if ladder style
<pre>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)</pre>	<pre>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);</pre>

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"

Normal nested-if style	if-else-if ladder style
<pre>int 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"); } } }</pre>	<pre>int 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"); }</pre>

The above left-side nested-if is cross-aligned; however, it is still readable because only a few decisions are involved. If the programmer does not take care to align the code properly, it becomes unreadable and difficult to debug. Hence, the if-else-if ladder style is always preferable. If more than 5 alternative decisions exist, a normal nested-if becomes awkward and confusing, so the ladder style is the better alternative in 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

```
int 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);
}
```

```
int 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 $\leq 12 \rightarrow$ Child
- age > 12 and $\leq 19 \rightarrow$ Teenager
- age > 19 and $\leq 50 \rightarrow$ Youngman
- age $> 50 \rightarrow$ old

```
int main()
{
    int age;
    printf("enter age :");
    scanf("%d", &age);
    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

int 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

```

int 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 op: 1 3 2012	ip: 31 12 2010 op: 1 1 2011	ip: 21 1 2010 op: 22 1 2010
-------------------------------	--------------------------------	--------------------------------

```

int 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 using if, if-else, nested if, or if-else-if ladder is left to the programmer. There are no specific situational rules suggesting which statement should be used when and where. However, it must be ensured that the written code is correct, clear, and easy to understand. While writing code, the programmer should remember that C always provides a well-structured alternative control statement whenever complexity increases with a single control statement.

if - else linking

eg1) the **if** and **else** statements have a **1:1 relationship**, each else is always linked to exactly one if in the program. We cannot use a single else statement for two or more if statements. The following example illustrates this rule.

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

In the above example, the **else** part is linked only to the last if-statement ($\text{marks3} > 50$). Therefore, we cannot use a single **else** part for multiple **if** statements. The solution is to write a separate **else** part for each **if-statement**, as shown below.

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

eg2) In some situations, the **else** is meant for the **outer-if** and not for the **inner-if**. In such cases, the **inner-if** should be enclosed within braces { }; otherwise, the compiler will link the outer else 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 intended the **else** part to belong to the **outer-if**, but compiler links it to the **inner- if**.

To avoid this logical error, enclose the inner-if within braces { }, as shown below.

```
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 evaluates to either 1 or 0, representing true or false respectively.

For example, if the relation $a < b$ is true, the $<$ operator returns 1; otherwise, it returns 0.

Such results are known as Boolean values in computer science.

In C, we can use any arithmetic expression in place of a test condition. In such cases, the result of the expression determines whether the condition is true or false. If the expression evaluates to any non-zero value (not necessarily 1), the condition is considered **true**; if it evaluates to 0, the condition is considered **false**. Observe the following program output

```
int 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 )
        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 multiple conditions exist, they must be combined using logical operators; otherwise, the program may produce incorrect output or a syntax error. Let us observe the output of the following program.

```
int 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($0 < -5 < 10$) // $0 < -5 \rightarrow \text{false} \rightarrow 0$
- if($0 < 10$) // $0 < 10 \rightarrow \text{true} \rightarrow 1$
- if(1)
- if(true)

Here, the test condition is true; hence, the if-block is executed. When two or more conditions exist, they must be combined using logical operators. The correct code is:

```
int 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($0 < k \&\& k < 10$) → $0 < -5 \&\& -5 < 10 \rightarrow 0 \&\& 1 \rightarrow 0$ (false)

Null instruction caused by semicolon(;)

The extra semicolon (;) in the program becomes a null statement. That is, if an extra semicolon is accidentally inserted, it is treated as a null statement by the compiler. For example:

```
a=10;
b=20;
;
c=30;
d=40;
```

Here, we have 5 instructions, not 4, because the extra semicolon is considered a null statement by the compiler. In this example, the extra semicolon does nothing.

Null instruction as if-statement's body

Observe the following example

<pre>int A=10, B=20; if(A<B) ; // null-instruction else printf(" sky is blue ");</pre> <p>Output: no output displayed</p>	<pre>int A=10, B=20; if(A>B) ; else printf(" sky is blue ");</pre> <p>output: sky is blue</p>	<pre>int A=10, B=20; if(A>B) ; else ; printf(" sky is blue ");</pre> <p>output: sky is blue</p>
---	--	--

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

<pre>A=5; if(A<0); printf("A is -VE"); prntf("\nGood Night");</pre> <p>op: A is -VE Good Night</p>	<pre>A=5; if(A<0) ; printf("A is +VE"); printf("\nGood Night");</pre>
---	--

<pre>int main() { int a=10; if(a<50) printf(" I love C "); else ; printf(" I marry C "); }</pre> <p>Output: I love C, I marry C</p>	<pre>int main() { int a=10; if(a<50) printf(" I love C "); else ; printf(" I marry C "); // this is not in else-part }</pre>
<pre>int main() { int A=10,B=20; if(A>B); printf("A>B"); ➔ compile time error else printf("A<=B"); }</pre>	<p>Output: compile time error unfortunately here if-block contained two instructions. 1. Null instruction by semicolon 2. printf("A>B");</p> <p>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)</p>

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

<pre>int main() { if(!0) printf("test passed"); else printf("test failed"); }</pre>	<pre>int main() { if(.00001) printf("test passed"); else printf("test failed"); }</pre>	<pre>int main() { int k=0; if(k) printf("hello"); else ; }</pre>
<pre>int main() { int k=20; if(k > 10) printf("I am in Right"); else ; printf("I am in Wrong"); }</pre>	<pre>int main() { int k=1000; if(10 > k < 35) printf(" I love C"); else printf(" I marry C"); }</pre>	<pre>int main() { if(-1.4) ; else printf("test failed"); }</pre>
<pre>int main() { if(-1.4) ; else printf("test failed"); }</pre>	<pre>int main() { int k=1; if(k>10) printf("k is +ve"); printf("k is > 10"); }</pre>	<pre>int main() { int k=1; if(k>10) printf(" k is +ve"); printf(" k is > 10"); else printf(" k is <=10"); }</pre>
<pre>int main() { int a=100,b=1; if(a b) printf("Good Morning"); else printf("Good Night"); }</pre>	<pre>int main() { int a=10, b=5, c, d; c=a>3; d=a>6 && b!=10; printf("c= %d , d=%d",c ,d); }</pre>	<pre>int main() { int a=30, b=40,c; c =(a!=10 && b==50); printf("b=%d, c=%d", b ,c); }</pre>
<pre>int main() { int a=20, b=230,c; c =(a==100 b>130); printf("c=%d",c); }</pre>	<pre>int main() { int k=1; if(k < 0) ; else printf("k is +ve"); }</pre>	<pre>int main() { int k=1; if(k < 0) printf("k is -ve"); else ; }</pre>
<pre>int main() { if(0) printf("A"); else printf("B"); }</pre>	<pre>int main() { if(10) printf("A"); else printf("B"); }</pre>	<pre>int main() { if(0-1) printf("A"); else printf("B"); }</pre>
<pre>Int main() { if(10); printf("A"); else printf("B"); }</pre>	<pre>int main() { if(10); printf("A"); }</pre>	

Switch Statement

Switch is another conditional control statement used to select one option from several options based on given **variable-value**. It serves as an alternative to the *if-else-if* ladder when comparisons are made against constants. The *if-else-if* ladder becomes lengthy and awkward when many comparisons or nested statements are involved, whereas the *switch* provides a cleaner and more readable structure even when multiple or nested cases are present. However, the *switch* statement has certain limitations: it compares only against with integral constants and cannot handle all situations supported by the *if-else-if* ladder.

In **switch** construct, the instructions are divided into case blocks based on condition and any number of case blocks can be defined. Observe the following example.

```
int main()
{
    int digit;
    printf("Enter any single digit (0-9) :");
    scanf("%d",&digit);
    switch( digit )
    {
        case 0: printf("zero");
        break;

        case 1: printf("one");
        break;

        case 2: printf("two");
        break;

        case 3: printf("three");
        break;
        -----
        -----
        case 9: printf("nine");
        break;

        default: printf("Invalid input");
    }
}
```

Here, the value of digit is compared with the case constants (case 0, case 1, case 2, ...) one by one. If a match is found, the corresponding printf() statement prints the English word.

For example, if the value of digit is 2, then the case2 block will be executed, and “two” will be printed as the output. After this, control comes out of the switch when the ‘break’ statement is executed. If break is not provided, control will continue into the next case (case3) as well. The break statement must be used to get out of a switch statement; otherwise, control will fall through into the next case blocks.

If no case matches the value of digit, then the default block is executed, and “Invalid input” is printed. Generally, the default block is used to handle errors such as invalid input or unexpected values.

The case-constants should be int-values, otherwise syntax error. See following code gives an error

```
float k=10.45;
switch( k )
{
    case 10.45: printf("hello"); // case constants should be integers and not 'float' constants.
    break;
    -----
    -----
}
```

- The above program can be written using if-else-if ladder form as shown below.

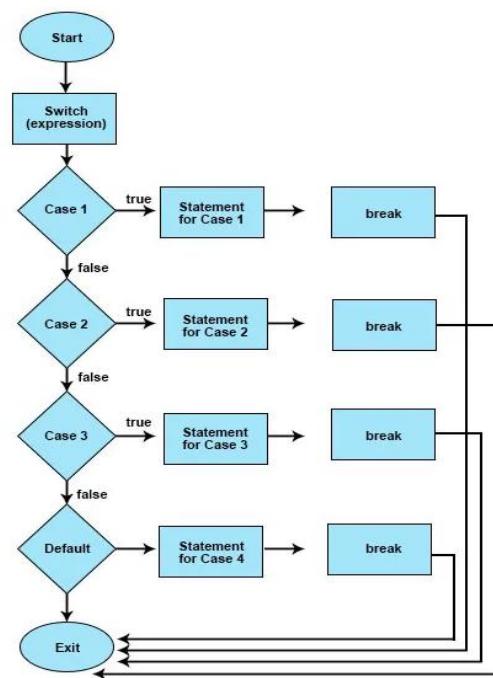
```
int main()
{
    int k;
    printf("Enter a digit(0-9):");
    scanf("%d",&k);
    if(k==0)
        printf("zero");
    else if(k==1)
        printf("one");
    .....
    else if(k==9)
        printf("nine");
    else
        printf("Invalid input"); // like default block
}
```

The above two programs generate exactly the same result. However, you might notice that the first program is easier to write, understand, and modify. The structure of the switch statement is more readable, even if it is nested two or three times. Let us see the syntax of switch statement.

```
switch(expression) // ← switch header
{
    case constant1:
        instruction1;
        instruction2;
        .....
        break;

    case constant2:
        instruction1;
        instruction2;
        .....
        break;
        .....
        .....

    default:
        instruction1;
        instruction2;
        .....
}
```



* The **expression** in the switch() header must be of an integral type, such as char, int, or long int.

* The value of the **expression** is compared with the constants (constant1, constant2, ...). If a match is found, control jumps to the associated case block and executes its instructions.

* 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 used to handle input errors.

* We can write any control statements inside case blocks, sometimes it can be nested

Demonstration of 'break' usage in switch

The 'break' statement is used to stop the execution of a switch statement, and the control comes out of the switch from that point. 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.

```
int main()
{
    int N;
    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

```
int 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;
    }
    printf("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.

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

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

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

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 often causes confusion for the programmer. Therefore, it is recommended to use it only when two-way decisions are required.

The following example shows how nesting increases complexity for the programmer. The program prints the largest of three numbers stored in the variables a, b, and c.

```
int 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 transfer the control from one location to another within a function. This unconditional control statement diverts the flow of execution to a specified location marked by a label. Using this statement, control can be moved forward, backward, or in any order within 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, modern languages provide a rich set of alternative control statements such as switch, while, for, do-while, break, continue, and return. However, in certain situations—such as breaking out of nested loops or solving recursion-like problems using loops—the available control statements may not be sufficient. In such cases, goto still has occasional uses.

syntax: **goto label;**

Here, label is a valid identifier followed by a colon. It indicates the location where the control is transferred using the goto statement. A label name must follow the same rules as variable names.

```
int 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

```
int main()
{
    printf("Red\n");
    goto one;
two: printf("Green\n"); <-- arrow from two to two
    printf("Blue\n");
    printf("Yellow\n");
    goto end;
one: printf("White\n"); <-- arrow from one to one
    printf("Cyan\n");
    goto two;
end: <-- arrow from end to 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 the machine code file. Instead, it works like a line indicator in the program. Generally, after compilation, every instruction in the program is associated with a line number in the machine code, which defines the sequence of instructions. The following program demonstrates how labels help in creating line numbers in the machine code file.

Code with labels	Code after compilation
<pre>int main() { printf("Red"); goto one; two: printf("Blue"); goto end; one: printf("Green"); goto two; end: printf("end of program"); }</pre>	 <pre>int main() { line1: printf("Red"); line2: goto line5; line3: printf("Blue"); line4: goto line7; line5: printf("Green"); line6: goto line3; line7: printf("end of program"); }</pre>

Unconditional control transfer

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

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 **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

```
int 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

```
int 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)

```
int 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 to repeat instructions more than once. This can be achieved using the goto statement; however, it is an unstructured control transfer statement that can cause severe programming complexity when used in nested form or when used excessively. To avoid such complexity, C provides structured control statements for looping, such as while, for, and do-while. These three constructs manage the repeated execution of a set of instructions until a specified condition is satisfied.

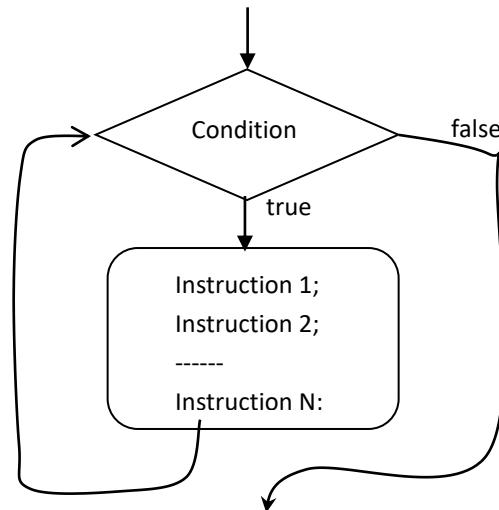
These loop control statements are used to execute a set of instructions repeatedly until a given test condition is satisfied. Each loop structure has its own style and provides convenience to the programmer. The programmer can choose any loop control statement according to the requirement and convenience, as there are no strict rules about which loop should be used in a particular situation.

All loop control statements consist of a test condition and a body. The body contains one or more instructions that are executed repeatedly until the test condition becomes false. Any type of instruction, including other control statements, can be written within this body. Let us see while-loop and its syntax.

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 a while loop, the test condition is checked first. If it evaluates to true, control enters the loop body and executes instruction-1, instruction-2, ..., instruction-N. After completing the body, control returns to the beginning of the loop for the next cycle, where the test condition is checked again. If it is still true, the loop body is executed once more. This process continues until the test condition becomes false.

If the test condition becomes false, control exits from the loop body and continues with the subsequent instructions, such as instruction n+1, instruction n+2, and so on.

If the loop body contains only one instruction, the pair of braces {} is optional, similar to the if-statement. However, if the loop body contains multiple instructions, they must be enclosed within a pair of braces {}.

The test condition must eventually become false after a certain number of iterations; otherwise, the loop will result in an infinite execution.

Do not place a semicolon after the loop header, as this will separate the header from the body."

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 $i \leq 10$
 step5. stop.

```
int 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 'i', 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

```
int 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;              // loop begins here
    while(i>=1)        // printing 'i' value on the screen
    {   printf("%d ", i); // decrementing 'i' by 1
        i--;
    }
}
```

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

Do not terminate the loop header with a semicolon; otherwise, the header and body get separated, and an empty loop is formed with a null instruction. In such cases, the semicolon itself acts as the loop body (a body with null instruction). The following example illustrates how a semicolon becomes the null instruction as the 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
<pre>i=1; while (i<=10); { printf("%d ", i); i++; }</pre>	 <pre>i=1; while(i<=10) ; { printf("%d ", i); i++; }</pre> <p>this loop goes infinite, because 'i' is not incrementing towards 10. The empty loop repeats again and again</p>

Printing numbers between two given limits

ip: 14 45

op: 14 15 16 17 1845

step1: take **lower & upper** as input variables and scan values 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.

```
int 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 'i' by 10 , to generate next output value
4. repeat step2 and step3 as far as i<= 100000
5. stop.

```
int 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 get next output value
4. repeat step2 and step3 until i>0
5. stop.

```
int 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 [$2^0, 2^1, 2^2, 2^3, 2^4, 2^5 \dots$]
3. set i=1, p=1 // starting values of i & p
4. print(p)
5. multiply p with 2, to get next output of 2^i value
6. increment loop variable 'i' by 1
7. repeat step4 to step6 until $i \leq 10$
8. stop.

```
int 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

```
int 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

```
int 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 13 111	op: 23 21 191

```

int 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.
}

```

If N is odd then printf() displays N value on the screen and then control goes to N—;

If N is even then printf() is skipped and control goes to N--;

Printing 10 numbers before and after of a given N

ip: 36

op: 26 27 28....34 35 36 37 38... 45 46

```

int 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

```
int 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 as follows

i → 1, 2, 3, 4, 5, 6, 7, ...

s → 1, -1, 1, -1, 1, -1, 1, ...

s*i → 1, -2, 3, -4, 5,

```
int 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

```
int 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($N \cdot i$) for 10 times where $i=1$ to 10 , and N is table number

```
int 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 value of N can be any number. To find its factors, divide N by all possible numbers from 1, 2, 3, ..., N . Any number that divides N exactly is a factor, and it should be printed on the screen. This is shown below.

Let N is : 15
check as : $15 \% 1 == 0 \rightarrow$ true \rightarrow print(1) as a factor
 $15 \% 2 == 0 \rightarrow$ false \rightarrow skips
 $15 \% 3 == 0 \rightarrow$ true \rightarrow print(3) as a factor
 $15 \% 4 == 0 \rightarrow$ false \rightarrow skips
 $15 \% 5 == 0 \rightarrow$ true \rightarrow print(5)

```
int 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: To find the smallest factor of N, divide N by 2, 3, 4, 5, ..., up to N. The first number that divides N exactly will be its smallest factor, and the loop can be stopped at that point.

```

int 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);
}

```

As soon as N is divisible by i for the first time, the loop stops using the break statement. After the loop terminates, the value of i is printed as the smallest factor. The arrow indicates how the control comes out of the loop. This program can be simplified as follows

```

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 lie in the range 1, 2, 3, ..., N/2 (excluding N). In other words, no factors exist beyond N/2. For example, if we take 100, the possible factors are 1, 2, 3, ..., 50.

The value 100 cannot be divided evenly by 51, 52, 53, ..., 98, 99, so it is unnecessary to check beyond N/2.

In this program, since we want the largest factor, it is efficient to check in reverse order from N/2 to 1.

```

int 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: try yourself to complete the code

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

note: try yourself to complete the code

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. (do not take N itself as factor)
logic: check for factors from 1 to N/2 and add all divisible values to ‘sum’. For example, 6 and 28 are perfect.

For example: 6 (1+2+3→6), 28(1+2+4+7+14→28)

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.

Tip: Practice by completing the code yourself.

Printing given number is prime or not?

ip: N = 17

op: yes, it is prime

ip: N = 18

op: no, it is not prime

In examinations, interviews, viva, etc., there is always one program that is frequently asked , none other than the “prime number logic”. Primes do not have factors except 1 and itself (they are not divisible by any number other than 1 and itself).

logic1: As we discussed in previous problems, for any kind of number, possible factors lie between 2 to N/2.

(not checking with 1 and N itself). So it is wise to check for prime-ness of N by dividing from 2 to N/2.

During checking process, if N is divided then stop the loop and say “not prime”. If it is not divisible at all till the end of the loop, then say “prime.” A beginner often writes the prime number logic in the wrong way as

```
for( i=2; i<=N/2; i++)
{
    if(N%i==0)
        printf(" not prime");
    else
        printf(" prime");
    i++;
}
```

Here at every division in the loop, we are printing prime/not-prime, so we get many outputs as given below

ip: N=15	(many outputs)
op: prime	// when 15%2==0 is false
not prime	// when 15%3==0 is true
prime	// when 15%4==0 is false
not prime	// when 15%5==0 is true
prime	// when 15%6==0 is false
prime	// when 15%7==0 is false

Note: to solve this problem properly, better to use boolean logic, there several other logics to find prime-ness, but this Boolean logic is standard and best, the code as given below

```
bool=1; // let us assume N is prime, so take bool as 1 (true).
for(i=2; i<=N/2; i++)
{
    if(N%i==0)
    {
        bool=0; // here N divided, therefore N is not prime, so set bool to 0 and stop the loop
        break;
    }
}
if(bool==1) printf("prime");
else printf("not prime");
```

logic2: Count all factors of N, i.e., divide N with all numbers from 1 to N and count all factors. If factors count==2 then say it is “prime” or else “not prime”. (This is simple logic but takes much time for executing)

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

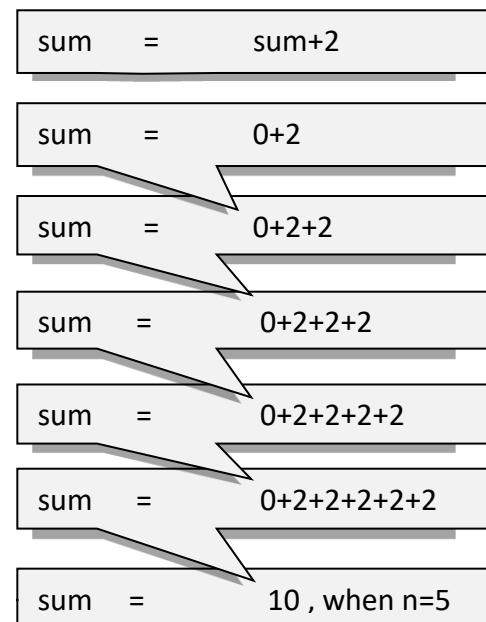
step3: add '2' to sum, this is as: sum=sum+2

step4: repeat step3 for N times

step5: print (sum)

Look at the picture, how 2 is added to 'sum' in cycles

```
int main()
{ int sum , i , N ;
  scanf("%d", &N);
  i=1, sum=0;
  while(i<=N)
  { sum=sum+2;
    i++;
  }
  printf("sum is %d", sum);
}
```



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 = garbage value+2". So, set sum=0 at beginning of loop.

Finding sum of series of $1^2+2^2+3^2+4^2+5^2 \dots +10^2$

Prove that sum of $1^2+2^2+3^2+4^2+5^2 \dots +10^2$ is equal to 385 or not?

output: yes/no

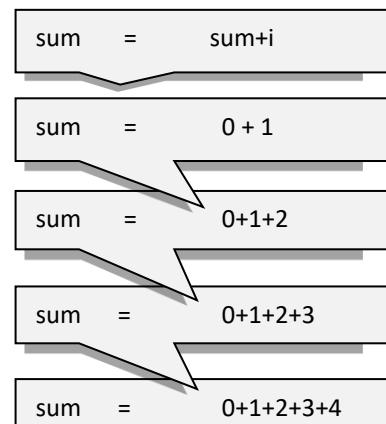
Logic: take 'sum' with zero, repeatedly add i^2 value to sum for 10 times, where i=1 to 10

after finding sum, compare with 385 for equality.

```
int 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. ($1+2+3+4\dots+n; 1*2*3*4\dots*n$)

```
int 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);
}
```



Finding sum of given N values

This program accepts numbers one by one from the keyboard until the last input value is 0. When 0 is entered, the program stops reading further input and prints the sum of all the values entered.

```
ip: 11↙ 3↙ 44↙ 30↙ 23↙ 0↙
op: sum = 111
int main()
{ int n, sum=0;
  while(1)
  {   printf("enter a value [ zero to end ] :");
      scanf("%d", &n);    // for scanning values one by one, the scanf() must be written inside the loop.
      if(n==0) break;
      sum=sum+n;          // summing value by value
  }
  printf("\n sum=%d ", sum );
}
```

Finding sum of $2^0 + 2^1 + 2^2 + \dots + 2^n$ without using pow() function

```
ip: n=5
op: sum=31
int 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);
}
```

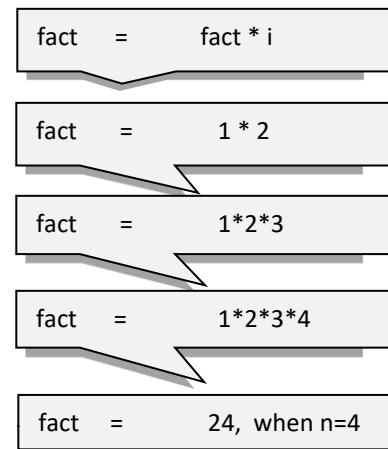
Finding sum of series of $x^1 + x^2 + x^3 + x^4 + \dots + x^n$ (without using power function)

```
ip: 3 5 (x=3, n=5)
op: 3+9+27+81+243 → 363
int 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  $x^1, x^2, x^3, \dots$  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)
int 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);
}
```

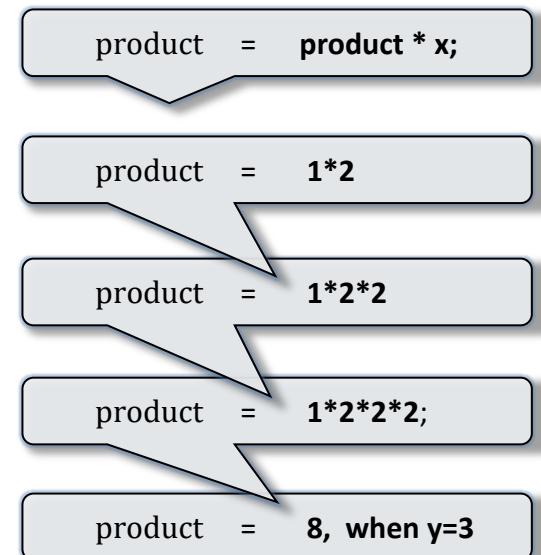


Finding x^y value based on given input values (x , y)

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

```
int 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);
}
```



$x^1/1! + x^2/2! + x^3/3!..... N \text{ times}$ [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 values $x^1, x^2, x^3, x^4, x^5, ...$
for this multiply 'x' to the 'p' to get x^i values.
3. Take a variable 'fact' and generate values 1!, 2!, 3!, 4!, ...
for this multiply 'i' to the 'fact' to get factorial values.
4. Take sum and add all these 'p/fact' values, this is like sum=sum + p/fact;

```
int main()
{   int i, fact, x, p, n;
    float sum;
    printf("enter x n value :");
    scanf("%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: $\sin(x) \rightarrow x^1/1! - x^3/3! + x^5/5! - x^7/7! \dots$ up to 10 terms

```

ip: x=3, N=5
op: sum=0.1453 [ (3.0) + (-4.5) + (2.025) + (-0.4339) + (0.05424) ]
int 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 a given N (The product of factors should be equal to N)

```

ip: N=100
op: 2 2 5 5

```

step1: Divide N by the first possible factor, 2; Since 2 is prime, if N is divisible by 2, print 2 as a prime factor and update $N = N / 2$.

step2: Repeat Step 1 as long as 2 divides N.

step3: Next, take 3 and apply the same process as in Step 1. Since 3 is also prime

step4: Now take 4. Although 4 is not prime, it will not divide N because any multiples of 2 have already been removed in step1.

step5: Repeat this process until $N > 1$.

You may wonder: why check with 4 when it is not prime? Because taking only primes is very difficult. Therefore, instead of selecting only prime numbers, we continuously check with 2, 3, 4, 5, 6,...etc.

```

i=2;
while(N>1)
{   if( N%i==0)
    {   printf("%d ", i);
        N=N/i;
    }
    else i++;
}

```

Printing prime factors of a given N without duplicates

This process is same as above program but it prints prime factors only once. (not like 2, 2, 5, 5)

ip: N=100

op: 2 5

During the loop cycles, if the current factor is not equal to the previously printed factor, then it is printed. For this, an extra variable 'prev' is used to store the previous prime factor of i. (Initially, set 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 'i'
        { 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 $x^1 + x^2 + x^3 \dots$

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

6) 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

op: 5, 4, 3, 2

ip: 724

op: 4, 2, 7

Logic: Extract digits one by one from right to left in N, and print them on the screen. Let us see following steps

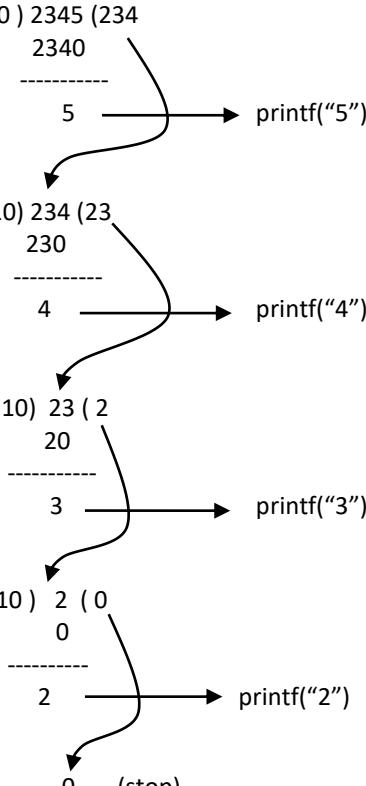
step 1: Divide N by 10 and collect the remainder. The remainder is always the last digit of N when it is divided by 10. (lastDigit=N%10)

step2: now print the last digit

step3: To get the next digit of N, remove the current last digit by updating N as: 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 Iteration-2 Here N is 234 Iteration-3 Here N is 23 Iteration-4 Here N is 2 Here N is 0 (stop)		<pre> int 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 in N } } </pre>
---	--	---

Finding sum of all digits in a given number (N)

ip: 2345

op: 2+3+4+5 → 14

step1: take 'sum' to store the addition of all digits

step2: take 'N' to store the input value

step3: repeat following instructions until N>0

- get the last digit(d) of N by dividing 10, the remainder will be the last digit [lastDigit=N%10]
- add the lastDigit to sum [sum = sum + lastDigit]
- remove the current lastDigit from N, for this update N by dividing 10 as : N=N/10

step4: print(sum)

step5: stop

Note: Implement 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 a deep study of the logic, one may write the program as badly as ...

```
int 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
    }
}
```

input: 2357

output: 5 not exist,	(output when digit 7 is checked)
5 exist,	(when digit 5 is checked)
5 not exist,	(when digit 3 is checked)
5 not exist	(when digit 2 is checked)

Here we don't print the output after checking each digit in a loop. Actually, the output "5 does not exist" can be confirmed only after checking all the digits in N. But in this loop, for every digit comparison, the output is shown on the screen. Therefore, the result is displayed four times. Here, we need to use boolean logic.

If the digit 5 is found during checking in the loop, then set bool = 1 and stop. After the loop, display the result based on the value of bool. Let see steps below

step 1: First of all, assume, digit 5 does not exist in N. So, initialize found = 0 (here found is a Boolean variable).

step2: repeat the 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 d is not 5, then check next digit by continuing the loop.

step3: after loop, print the result based on the '**found**'

```
if(found==1) print(" 5 found");
else print("5 not found");
```

```
int main()
{
    int N, d, found=0;        // found=0 means , assume digit '5' not exist in N.
    scanf("%d", &N);
    while(N>0)
    {
        d = N%10;
        if(d==5)
        {
            found=1;    // now the digit '5' found, so set found=1 and stop the loop
            break;
        }
        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
int main()
{
    int N, evenSum, oddSum, d;
    printf("enter N value :");
    scanf("%d", &N);
    evenSum=oddSum=0;
    while(N>0)
    {
        d=N%10;
        if( d%2==0) evenSum+=d;
        else oddSum+=d;
        N=N/10;
    }
    printf("\n even sum = %d, odd sum = %d", evenSum, oddSum);
}

```

Counting number of digits in a given number (N)

ip: 234	ip: 45
op: count=3	op: count=2

- step1) Take variable digitCount to count the digits in N
- step2) Remove the last digit in N by doing N=N/10 and count the removed digit as one.
for this increment digitCount++;
- step3) Repeat this step until 'N' becomes zero
- step4) Print output 'digitCount'

```

int 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);
}

```

The variable values in every cycle as		
	digitsCount++	N=N/10
Initially→	0	2345
After 1 st cycle	1	234
After 2 nd cycle	2	23
After 3 rd cycle	3	2
After 4 th cycle	4	0

Finding given number (N) is a Armstrong or not?

The number is Armstrong, if the sum of cubes of all digits is equal to N: 153 → $1^3+5^3+3^3 \rightarrow 1+125+27 \rightarrow 153$

ip: 153	ip: 234
op: Armstrong	op: not Armstrong

```

int 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 $N=N/10$ replaces N to zero by the time the loop ends. Therefore, the original value of N is lost after the loop. To compensate this, the value of N is saved in a variable ‘temp’ before the loop begins. Later, ‘temp’ is compared with ‘sum’, and the result is printed.

Finding reverse of a given number (N)

ip: 234	ip: 4673
op: 432	op: 3764

Let us assume the variable ‘rev’ already has the value 23 (rev = 23).

Now, to insert 4 at the end of ‘rev’, we do: rev = rev * 10 + 4 → 23 * 10 + 4 → 234.

In this way, the digits are extracted one by one from N and appended to the end of ‘rev’.

```

int 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 1 st cycle	$2345 \% 10 \rightarrow 5$	$0 * 10 + 5 \rightarrow 5$	234
After 2 nd cycle	$234 \% 10 \rightarrow 4$	$5 * 10 + 4 \rightarrow 54$	23
After 3 rd cycle	$23 \% 10 \rightarrow 3$	$54 * 10 + 3 \rightarrow 543$	2
After 4 th cycle	$2 \% 10 \rightarrow 2$	$543 * 10 + 2 \rightarrow 5432$	0

Finding given number (N) is a 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 input value and print output

```
int 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 the digits in N. If any digit is greater than bigg, then copy that digit into bigg. Finally, the largest digit will be stored in bigg. Remember to initialize bigg to zero at the beginning.

```
int 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
op: 3

ip: 2468
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'.

```
int 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 4 digit single number(N) in English words

This program extracts all the digits of N from left to right and prints each digit separately in English words.

ip: 2345 op: two three four five	ip: 7246 op: seven two four six
-------------------------------------	------------------------------------

Extract the digits of **N** one by one from left to right. To do this, divide **N** by 1000, 100, 10, and 1.

At each division, store the quotient in Q and update **N** with the remainder.

In this way, Q collects the digits from left to right, while **N** retains the remaining part as the remainder.

```
int main()
{
    int N, D, Q;
    scanf("%d", &N);
    D=1000;
    while( D > 0 )
    {
        Q=N/D;
        if( Q==0 ) printf("zero");
        else if( Q==1 ) printf("one");
        else if( Q==2 ) printf("two");
        -----
        N=N%D;
        D=D/10;
    }
}
```

N=2345, D=1000		
D) N (Q		
Iteration1 N→2345 D→1000	1000) 2345 (2 → printf("two") 2000 ----- 345 ↓	
Iteration2 N→345 D→100	100) 345 (3 → printf("three") 300 ----- 45 ↓	
Iteration3 N→45 D→10	10) 45 (4 → printf("four") 40 ----- 5 ↓	
Iteration4 N→5 D→1	1) 5 (5 → printf("five") 5 ----- 0 ↓	
		Stop

Printing each digit of a number(N) in English words

The previous program works only for a 4-digit number. The following program is an updated version that can be used for numbers with any number of digits.

It is difficult to extract the digits of N from left to right, because the input N may have any number of digits entered by the user (not exactly four digits as in the earlier example).

For example, to get the first digit 3 from 3456, the number must be divided by 1000. Similarly, to get 7 from 724, it must be divided by 100. Depending on the number of digits in N, it should be divided by $1000/100/10/1$.

As per above program logic, here, the value D must be generated according to the number of digits in N. If N has 4 digits, then D = 1000. If N has 3 digits, then D = 100.

In this way, we need to generate D value to $10^{(x-1)}$, where x is the number of digits in N.

Let us see the following code.

```
D=1;
while(N/D>9)      // loop to generate 'D' value to  $10^{x-1}$ 
{   D=D*10;
}
```

Let us see the complete program.

```
int main()
{   int N, Q, D=1;
    printf("enter N value:");
    scanf("%d", &N);
    while(N/D>9)      // generating 'd' value to  $10^{x-1}$ 
    {   D=D*10;
    }
    while(D>0)
    {   Q=N/D;        // 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%D;        // removing first digit from N
        D=D/10;
    }
}
```

Finding decimal value from a given binary input

ip: 1101

op: 13

ip: 1111

op: 15

1. multiply all digits of N with $2^0 \ 2^1 \ 2^2 \ 2^3 \dots$ from right-to-left
2. The sum of such all products forms a decimal number.

$$\begin{array}{cccc} 1 & 1 & 0 & 1 \\ 2^3 & 2^2 & 2^1 & 2^0 \end{array} \rightarrow \boxed{1*2^3 + 1*2^2 + 0*2^1 + 1*2^0} \rightarrow 13$$

$$8 + 4 + 0 + 1$$

```
int 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

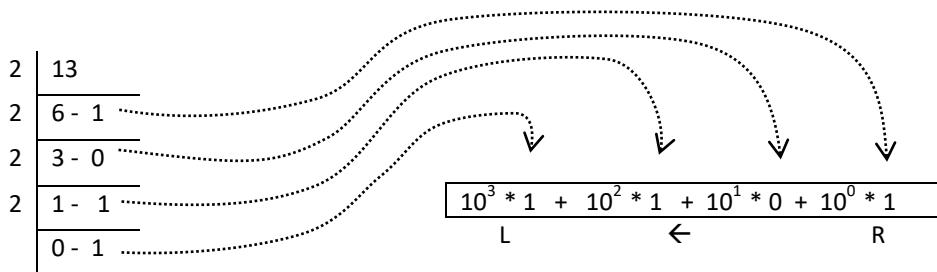
ip : 13

op : 1101

ip: 10

op: 1010

1. let us suppose the given number is 13 ($N=13$)
2. to get the 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 remainder with 10^i and sum up

step2: the binary number is formed as **1000*1+100*1+10*0+1*1 → 1101**

```
int 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

step4: 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 → 0 <u>7 11 06 10 05</u>

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.

```

int 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 and 1, and the remaining terms are generated by adding the previous two terms. This is an endless series, but here we are printing only the 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;

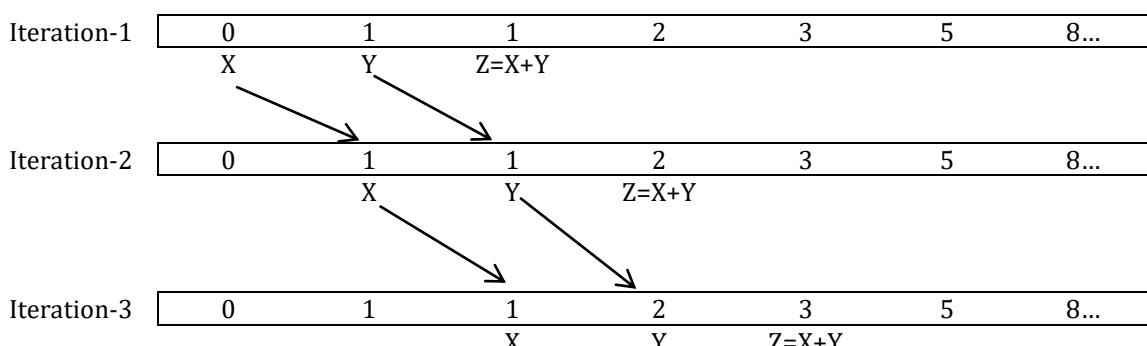
step2: Print the term X.

step3: Generate the next term by adding X+Y to Z

step4: Now advance X to Y and Y to Z 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



```
int main()
{
    int X, Y, Z, i;
    X=0; Y=1; i=0;
    while(i++ < 10)
    {
        printf("%d\t", X);
        Z=X+Y;           // generating next fibo number
        X=Y;             // advancing X to Y and Y to Z
        Y=Z;
    }
}
```

Finding GCD of two numbers

ip: 12 18

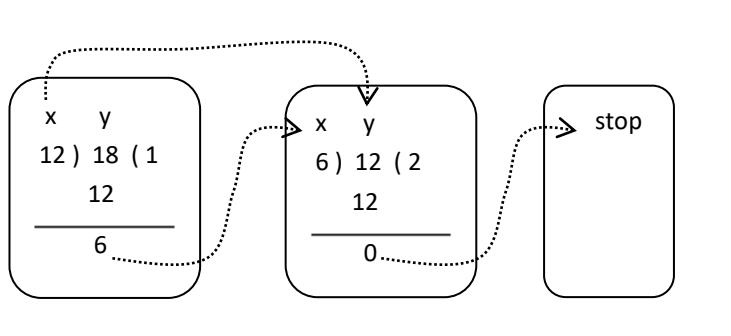
op: gcd = 6

step 1: Let X and Y be the input values.

step 2: Divide Y by X and store the remainder in R (it does not matter whether X>Y or X<Y).

step 3: If the remainder (R) is zero, then stop and print X as the GCD.

step 4: If R is not zero, then move X value to Y, and R value to X. Continue this process until R=0.



```
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

step1: Let X, Y, Z be three input numbers.

step2: Start finding the LCM of the three numbers by dividing with the first factor, 2.

step3: Now divide each of X, Y, Z by 2.

step4: If any of X, Y, Z is divisible by 2, then take 2 as a factor in the LCM ($LCM=LCM*2$), and replace each divisible number with its quotient. For example, if X is divisible by 2, then update: $X=X/2$.

step5: Repeat step4 as long as 2 divides any of X, Y, Z.

step6: If 2 no longer divides any of X, Y, Z, then try with the next factors: 3, 4, 5, ... and continue the same process. Repeat this until all of X, Y, Z become 1. For example: while ($X > 1 \mid\mid Y > 1 \mid\mid Z > 1$) { ... }

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	35
	1	1	1

```

int 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 given 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

```
int 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. Read two dates from the keyboard: date1(d1, m1, y1) and date2(d2, m2, y2).
2. If date1 > date2, then swap them.
3. In the loop, increment date1 step by step until it reaches date2.
4. For each cycle, increment the ‘count’ by 1 to keep track of the number of days between the two dates.

```
int 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 a while-loop is simple and is used when the loop construct contains only the condition part. However, in most cases, loops also involve initialization and incrementing or decrementing. In such situations, the for-loop is more suitable. In a while-loop, the initialization of the loop variable ($i = 1$) must be written before the loop, and the increment part ($i++$) must be placed inside the loop. Thus, initialization, condition, and increment have to be written in three different places.

In contrast, a for-loop allows all three parts initialization, test condition, and increment to be written together in a single place, within the loop header. This makes the for-loop more compact and convenient.

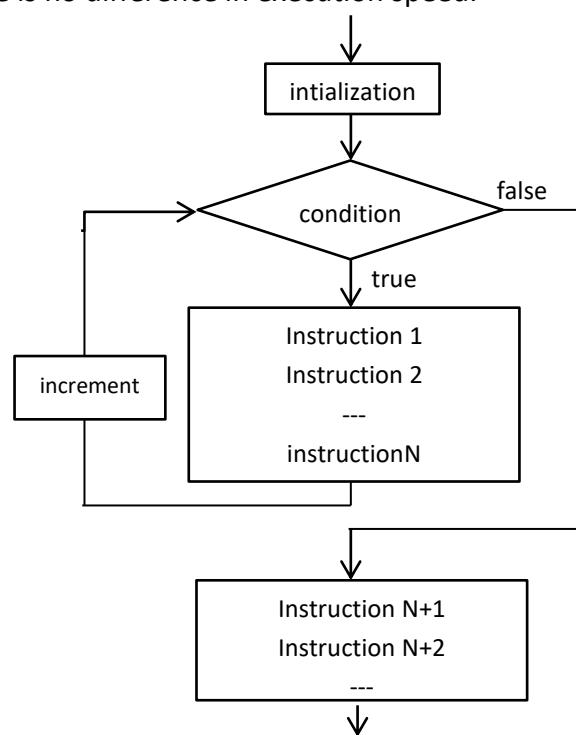
When a loop needs to be repeated a fixed number of times (such as 10 times or N times), the for-loop is the better choice. Otherwise, the while-loop is more appropriate. Remember, when your loop has parameters like $i = 1$ (initialization) and $i++$ (increment), then the for-loop is the preferred choice.

The for-loop is more flexible as it clearly shows the complete structure of the loop, including the initial value, condition, and increment. Ultimately, it is the programmer's choice which loops to use in a given situation, and it largely depends on personal convenience. Some people believe that the for-loop is faster than the while-loop.

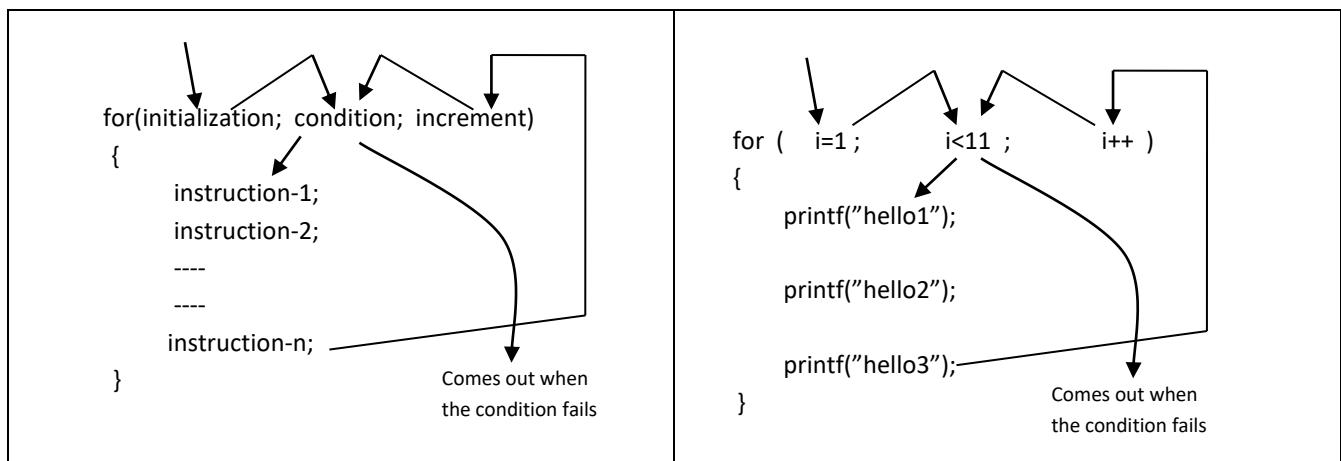
In reality, both generate the same machine code, so there is no difference in execution speed.

for-loop syntax:

```
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. In the first cycle, control enters through the initialization part. From the second cycle onward, control passes through the increment/decrement part. Let us see:



Let us have some examples,

Loop to print 1 to 10 numbers	Same code in while loop structure
<pre>for(i=1; i<11; i++) { printf("%d ", i); }</pre> <p>op: 1 2 3 4 ... 10</p>	<pre>i=1; while(i<11) { printf("%d ", i); i++; }</pre> <p>op: 1 2 3 4 ... 10</p>
Loop to print multiples of 5 for 10 times	Loop to print 10 to 1 numbers
<pre>for(i=5; i<=50; i=i+5) { printf("%d ", i); }</pre> <p>op: 5 10 15 20 25 ... 50</p>	<pre>for(i=10; i>0; i--) { printf("%d ", i); }</pre> <p>op: 10 9 8 7 ... 1</p>

eg1) loop with more initializations

we can initialize more than one variable in the loop header. In such cases, use a comma (,) to separate them (do not use semicolon).

```
for( i=1 , j=2, k=3; ----- ; ----- )
{
    -----> separate by coma (not semicolon)
}
```

eg2) loop with more increments or decrements

we can increment/decrement more than one variable in header, but use comma(,) to separate them

```
for(----- ; -----; i++ , j=j+2 , k-- )
{
    -----> separate by coma (not semicolon)
}
```

eg3) loop with more 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
}
```

eg4) following loops are same

<pre>for(; 1 ;) { -----</pre>	→	<pre>while(1) { -----</pre>
-------------------------------------	----------	---------------------------------

// these loops turns into infinite-loop

eg5) infinite loop: if no condition is provided then compiler takes as infinite loop

```
for( ; ; )      // this is same as: while(1) { ... }
```

eg6) All three parts of a for-loop header may or may not be present. We can omit any part simply by leaving it blank. However, the two semicolons must always be present, as they indicate the separation of the three parts.

```
int main()
{
    int i=1;
    for( ; i<11 ; i++) ————— observe here, initialization part is omitted
    {
        printf("%d ", i);
    }
}
```

eg7) int main()

```
{    int i;
    for( i=1; i<11; ) ————— observe here, increment part is omitted
    {
        printf("%d ", i);
        i++;
    }
}
```

eg8) i=1;

```
for( ; i<11; ) ————— observe here, both initialization & increment parts are omitted
{    printf("%d ", i );
    i++;
}
```

eg9) for(i=1; ; i++) ————— if condition part is omitted, then loop turns into infinite-loop

```
{    printf("%d ", i );
}
```

eg10): A for-loop header must contain exactly two semicolons. If not it will result in a syntax error.

```
for( i=1 ; i<10; i++ ; j++ ) —————→ unfortunately three semicolons found here , so syntax error.
{ }
```

eg11): i=1; —————→ unfortunately no semicolons found here , this is also syntax error.

```
for( i<10 )
{
    printf("hello");
    i++;
}
```

Displaying natural numbers between two given limits

```

ip: 23 45
op: 23 24 25 ....45
int 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: A user may enter any input value. To find the factors of that value, divide N by each number from 1 to N.

Print the number whenever N is divisible by it.

```

int main()
{ int N,i ;
  printf("enter N value :");
  scanf("%d", &N);
  for(i=1; i<=N; i++)
  { if( N%i==0)           // if N divisible by 'i' 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);

```

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);
}

```

Finding given number is a 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"

```
int 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 alphabets and its ASCII codes

op: alphabets (A B C D E F G Z, a, b, c, d....)

The upper case A-Z ASCII codes are: 65 66 6790

The lower case a-z ASCII codes are: 97 98 99122

```
int main()
{
    char ch;
    printf("\n upper case alphabets :");
    for (ch='A'; ch<='Z'; ch++) OR for (ch=65; ch<=90; ch++)
        printf("%c",ch);                                                 printf("%c",ch);

    printf("\n lower case alphabets :");
    for (ch='a'; ch<='z'; ch++) OR for (ch=97; ch<=122; ch++)
        printf("%c",ch);                                                 printf("%c",ch);

    printf("\n the ASCII values are:");
    for (ch='A'; ch<='Z'; ch++)                                        // this loop prints the ascii codes of A - Z
        printf("%d ",ch);                                                    // here %d prints the ascii codes.
}
```

In the above program, the variable '**ch**' internally (in memory) contains the **ASCII** value of characters. Therefore, we can perform any arithmetic operations like incrementing and decrementing. When '**ch++**' is done then it attains next character **ASCII** code.

do-while loop

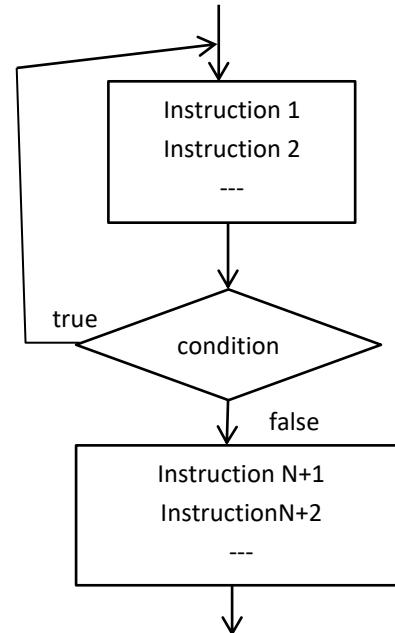
The do-while loop is the opposite of the while loop in terms of condition checking. In a while loop, the condition appears at the top of the loop body, whereas in a do-while loop, the condition appears at the bottom of the loop body. Thus, the while loop is a top-test construct, and the do-while loop is a bottom-test construct.

As a result, the body of a do-while loop executes at least once, regardless of whether the condition is true or false. In contrast, the while loop may or may not execute at all—it can fail at the beginning without executing even once. But **do-while** loop executes at least once irrespective of condition is true/false.

This is rarely used in the programming, 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 with the 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

```
int 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 the keyboard until the last value is zero. When zero is entered, it stops scanning and prints the sum of all the values.

```
int 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
    printf("sum = %d", sum);
}
```

above program using while-loop

```
int main()
{
    int n, sum=0;
    while(1)
    {
        printf("enter values one by one :");
        scanf("%d", &n);
        if( n==0) break;      // if input value is zero then program terminates by break;
        sum = sum+n;
    }
    printf("sum = %d", sum);
}
```

break & continue statements

Jumping out of loop using “break”

A loop performs a set of operations repeatedly until the given condition is satisfied. However, in some situations, the loop needs to be stopped unexpectedly when the required result is obtained earlier.

The break statement is a condition-less control statement used to terminate the loop immediately when the desired result is found earlier. It jumps the control out of the loop body, thereby terminating the loop.

syntax

```
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, certain iterations of a loop need to be bypassed for specific data. For example, when calculating the rank of each passed student in a class, the record of a failed student must be skipped.

The continue statement is also a condition-less control statement. It is used to skip certain instructions in the loop when a specified condition is met. When ‘continue’ is executed, control immediately jumps back to the beginning of the loop for the next cycle, causing the remaining instructions in the current iteration to be bypassed.

```
while ( condition )
{   instruction 1;
    instruction 2;
    if( condition )
        continue;
    instruction 3;
    instruction 4;
    instruction 5;
    -----
    instruction N;
}
```

// instructions-3 to instruction-N are bypassed by continue

examples for “break”

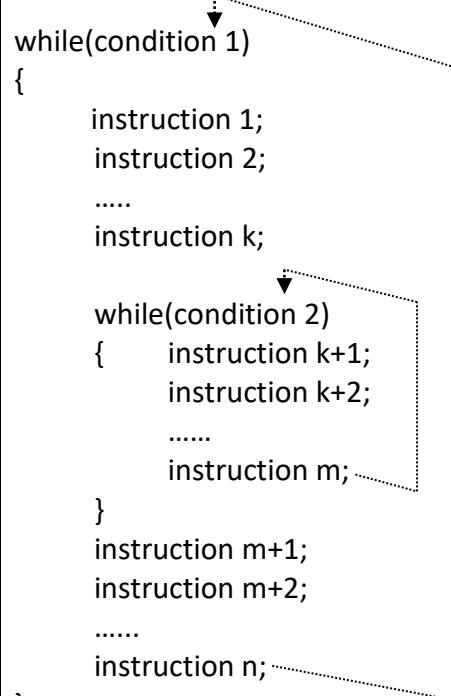
```
int main()
{ int i;
    for(i=1; i<=10; i++)
    {   if(i==5)
        break;
        printf("%d ", i);
    }
}
output: 1 2 3 4
```

example for “continue”

```
int 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 within another loop is known as a nested loop. The inner and outer loops can be of the same type or of different types. For example, a while-loop can be nested inside a for-loop. Similarly, loops can be nested multiple times as needed in a program. Some examples of nested loop structures are as follows:

<pre>while(----) { while(----) { --- --- } }</pre>	<pre>For(----; ----; ----) { while(----) { --- --- } }</pre>
<pre>for(----; ----; ----) { for(----; ----; ----) { --- --- } }</pre>	<pre>while(----) { for(----; ----; ----) { while(----) { --- --- } } }</pre>
 <pre>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; }</pre>	<pre>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; }</pre>

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, which is made up of rows and columns. We know that to display numbers in a single row, we need a loop. Similarly, to display such rows multiple times (for example, 10 times), we need an extra loop. This extra loop is called the outer loop, and its role is to repeat the inner loop for 10 times.

```
int main()
{ int i, j;
  for( i=1; i<=10; i++)
    {
      for(j=3; j<=8; j++)
        printf("%d ", j);
      printf("\n");
    }
}
```

Here the outer-loop is used to repeat the pattern for 10 rows and inner-loop is to print 3 to 8 of each row.

```
8 7 6 5 4 3
8 7 6 5 4 3
.....
10 rows
```

```
int main()
{ int i, j;
  for( i=1; i<=10; i++)          // loop to print 10 rows
  {
    for( j=8; j>=3; j--)        // loop is 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 is to displays numbers from 8 to 3.

<pre>123456 12345 1234 123 12 1</pre>	<pre>for(i=6; i>=1; i--) { for(j=1; j<=i; j++) printf("%d ", j); printf("\n"); }</pre>	<p><i>when i=6, the j-loop is</i></p> <pre>for(j=1; j<=6; j++) //1 2 3 4 5 6 printf("%d ", j);</pre> <hr/> <p><i>when i=5, the j-loop is</i></p> <pre>for(j=1; j<=5; j++) //1 2 3 4 5 printf("%d ", j);</pre> <hr/> <p><i>when i=4, the j-loop is</i></p> <pre>for(j=1; j<=4; j++) //1 2 3 4 printf("%d ", j);</pre>
---------------------------------------	---	--

654321 54321 4321 321 21 1	for(i=6; i>=1; i--) { for(j=i; j>=1; j--) printf("%d ", j); printf("\n"); }	when i=6, the j-loop is for(j=6; j>=1; j--) // 6 5 4 3 2 1 printf("%d ", j); ----- when i=5, the j-loop is for(j=5; j>=1; j--) // 5 4 3 2 1 printf("%d ", j); ----- when i=4, the j-loop is 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"); }	when i=1, the j-loop is for(j=1; j<7; j++) // 1 2 3 4 5 6 printf("%d ", j); ----- when i=2, the j-loop is for(j=2; j<7; j++) // 2 3 4 5 6 printf("%d ", j); ----- when i=3, the j-loop is for(j=3; j<7; j++) // 3 4 5 6 printf("%d ", j);
---	---	--

1 1 2 1 2 3 1 2 3 4 1 2 3 4 5 1 2 3 4 5 6	for(i=1; i<7; i++) { for(j=1; j<=i; j++) printf("%d ", j); printf("\n"); }	when i=1, the j-loop is for(j=1; j<=1; j++) // 1 printf("%d ", j); ----- when i=2, the j-loop is for(j=1; j<=2; j++) // 1 2 printf("%d ", j); ----- when i=3, the j-loop is for(j=1; j<=3; j++) // 1 2 3 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"); }	when i=1, the j-loop is for(j=1; j<=7; j++) // 1111111 printf("%d ", i); ----- when i=2, the j-loop is for(j=1; j<=7; j++) // 2222222 printf("%d ", i); ----- when i=3 the j-loop is 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"); }
8 7 6 5 4 3 2 1 8 7 6 5 4 3 2 8 7 6 5 4 3 8 7 6 5 4	for(i=1; i<=8; i++) // loop to print 8 rows { for(j=8; j>=i; j--) // loop is to print 8 to i column values of each row printf("%d ", j); printf("\n"); }
8 8 7 8 7 6 8 7 6 5	for(i=8; i>=1; i--) // loop to print 8 rows { for (j=8; j>=i; j--) // loop is to print 8 to i column values of each row printf("%d ", j); printf("\n"); }
8 8 7 8 7 6 8 7 6 5	for(i=1, k=8; i<=8; i++) { for(j=1; j<=k; j++) printf("%d ", j); printf("\n"); k--; }
8 7 7 6 6 6 5 5 5 5	for(i=1; i<=8; i++) { for(j=1; j<=i; j++) printf("%d ", 9-i); printf("\n"); }
6 6 6 6 6 6 6 5 5 5 5 5 5 4 4 4 4 4	for(i=1; i<=6; i++) { for(j=1; j<=7-i; j++) printf("%d ", 7-i); printf("\n"); }
6 6 6 6 6 6 6 5 5 5 5 5 5 4 4 4 4 4	for(i=6; i>=1; i--) { for(j=1; j<=i; j++) printf("%d ", i); printf("\n"); }

5 blanks →	<table border="1"><tr><td></td><td></td><td></td><td></td><td>1</td><td></td><td></td><td></td><td></td></tr></table>					1				
				1						
4 blanks →	<table border="1"><tr><td></td><td></td><td></td><td>2</td><td>2</td><td></td><td></td><td></td><td></td></tr></table>				2	2				
			2	2						
3 blanks →	<table border="1"><tr><td></td><td></td><td>3</td><td>3</td><td>3</td><td></td><td></td><td></td><td></td></tr></table>			3	3	3				
		3	3	3						
2 blanks →	<table border="1"><tr><td></td><td>4</td><td>4</td><td>4</td><td>4</td><td></td><td></td><td></td><td></td></tr></table>		4	4	4	4				
	4	4	4	4						
1 blanks →	<table border="1"><tr><td>5</td><td>5</td><td>5</td><td>5</td><td>5</td><td>5</td><td></td><td></td><td></td></tr></table>	5	5	5	5	5	5			
5	5	5	5	5	5					
0 blanks →	<table border="1"><tr><td>6</td><td>6</td><td>6</td><td>6</td><td>6</td><td>6</td><td>6</td><td></td><td></td></tr></table>	6	6	6	6	6	6	6		
6	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(" ");
    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

To get this pattern, some spaces need to be added before each row as,

7 blanks →	<table border="1"><tr><td></td><td></td><td></td><td></td><td></td><td>1</td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>						1	1							
					1	1									
6 blanks →	<table border="1"><tr><td></td><td></td><td></td><td></td><td>1</td><td>2</td><td>2</td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>					1	2	2	1						
				1	2	2	1								
5 blanks →	<table border="1"><tr><td></td><td></td><td></td><td>1</td><td>2</td><td>3</td><td>3</td><td>2</td><td>1</td><td></td><td></td><td></td><td></td><td></td></tr></table>				1	2	3	3	2	1					
			1	2	3	3	2	1							
4 blanks →	<table border="1"><tr><td></td><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>4</td><td>3</td><td>2</td><td>1</td><td></td><td></td><td></td><td></td></tr></table>			1	2	3	4	4	3	2	1				
		1	2	3	4	4	3	2	1						
3 blanks →	<table border="1"><tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td></td><td></td><td></td></tr></table>		1	2	3	4	5	5	4	3	2	1			
	1	2	3	4	5	5	4	3	2	1					
2 blanks →	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td></td><td></td></tr></table>	1	2	3	4	5	6	6	5	4	3	2	1		
1	2	3	4	5	6	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 →	<table border="1"><tr><td></td><td></td><td></td><td></td><td>*</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>					*								
				*										
4 blanks →	<table border="1"><tr><td></td><td></td><td></td><td>*</td><td>*</td><td>*</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>				*	*	*							
			*	*	*									
3 blanks →	<table border="1"><tr><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td></td><td></td><td></td><td></td><td></td></tr></table>			*	*	*	*	*	*					
		*	*	*	*	*	*							
2 blanks →	<table border="1"><tr><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td></td><td></td><td></td><td></td></tr></table>		*	*	*	*	*	*	*	*				
	*	*	*	*	*	*	*	*						
1 blanks →	<table border="1"><tr><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td></td><td></td><td></td></tr></table>	*	*	*	*	*	*	*	*	*	*			
*	*	*	*	*	*	*	*	*	*					
0 blanks →	<table border="1"><tr><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td></td><td></td></tr></table>	*	*	*	*	*	*	*	*	*	*	*		
*	*	*	*	*	*	*	*	*	*	*				
...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

```
int 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 the keyboard until zero the end of program. When zero is entered, the loop is terminated using the break. For every input, the corresponding table is printed.

```
int 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

```
int 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

Twin primes are pairs of prime numbers whose difference is 2. For example: (3, 5), (5, 7), (11, 13), (17, 19).

Initially, take the first prime number as 2 (say previous = 2).

Now, start checking numbers for prime from 3 to 100 (N = 3 to 100).

If N is prime, then check whether N - previous == 2. if yes, then print previous and N as twin primes.

Finally, assign the current N to previous for the next cycle.

```
int main()
{
    int n, i, prev, bool;
    prev=2; // let us take first prime 2 for 'prev'
    for(n=3; n<=100; n=n+2) // loop to check for prime from 3-100 (Generally, odds could be prime, except 2)
    {
        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

```
int 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 in n
    {
        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 divisors other than 1 and itself

Armstrong: sum of cubes of digits equal to given number ($153 \rightarrow 1^3 + 5^3 + 3^3 \rightarrow 153$)

Perfect: sum of factors equal to given number like 6 ($1+2+3 \rightarrow 6$)

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]:

```
int 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("=====");
        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: // checking odd or even
                if( N%2==0 ) printf("Even");
                else printf("Even");
                break;

            case 2: // checking palindrome or not
                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: // checking prime or not
    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: // Armstrong or not
    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

Let us take a problem where we need to handle 10 values, such as finding their sum and the largest among them. One might write the code without using an array as:

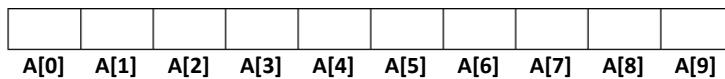
```
int main()
{   int a, b, c, d, e, f, g, h, i, j, k, sum, big;
    printf("Enter 10 values: ");
    scanf("%d%d%d%d%d%d%d%d", &a, &b, &c, &d, &e, &f, &g, &h, &i, &j);
    sum = a + b + c + d + e + f + g + h + i + j; // finding sum of all
    if (a > b && a > c && a > d && a > e && a > f && a > g && a > h && a > i && a > j && a > k) // finding big
        big = a;
    else if( ----- )
    -----
}

```

This code looks awkward and complex. If we need to handle more values, it becomes almost impossible to manage. Let us see above program using arrays

int A[10];

This declaration creates 10 variables in a sequence, with the names: A[0], A[1], A[2], ..., A[9]; All creates as given below picture. These 10 variables together are called an array (array means collection).



here, A[0] accesses the first location in the array.

here, A[1] accesses the second location in the array.

here, A[2] accesses the third location in the array.

in general, A[i] accesses the (i+1)th location of the array.

For example, we can print all the values as:

```
printf("%d %d %d %d %d %d %d %d", A[0], A[1], A[2], A[3], A[4], A[5], A[6], A[7], A[8], A[9]);
```

using loop, the above printf() can be written as:

```
for (i=0; i<10; i++)
{   printf("%d ", A[i]);
}
```

Now we can write the above program using arrays as

```
int main()
{   int a[10], i, sum=0, big;
    printf("Enter 10 values: ");
    for (i = 0; i < 10; i++) // scanning 10 values using loop
    {   scanf("%d", &a[i] );
    }
    for (i = 0; i < 10; i++)
    {   sum = sum + a[i];      // finding all values sum.
    }
    -----
    printf("Sum = %d , big = %d ", sum, big);
}
```

By observing the above code, we can understand how arrays simplify the program.

Let us now learn them step by step.

In the real world, we often need to manage collections of data such as marks of all students in a class, the prices of items in a supermarket, bank accounts data, etc. To handle these collections, arrays provide an effective solution. The word “array” refers to a collection of homogenous items arranged sequentially, one after another.

In computer science, an array is a collection of adjacent memory locations used to store and access multiple similar values using a single name. It is a mechanism for handling collections of similar values with a single name. The new definition is, array is a collection-type data-type designed to handle a collection of same type values. According to mathematics terminology, array is a **vector** while single item 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. However, in practice, we most often use one-dimensional or two-dimensional arrays. Let us see how memory is allocated for arrays.

`int X[5];` → single dimensional array

X[0]	X[1]	X[2]	X[3]	X[4]

`int Y[5][6];` → two dimensional array

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][2]	Y[1][5]
Y[2][0]	Y[2][1]	Y[2][5]
Y[3][0]	Y[3][1]	Y[3][5]
Y[4][0]	Y[4][1]	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 integer items. The declaration of array is same as other 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. The index ranges from **0** to **arraysize-1**.

The syntax to access array element is: **array-name[index]**

For example,

13	26	7	15	10
x[0]	x[1]	x[2]	x[3]	x[4]

The expression `x[0]` accesses the 1st element in the array. Here, `x[0] = 13`.

The expression `x[1]` accesses the 2nd element in the array. Here, `x[1] = 26`.

The expression `x[i]` accesses the $(i+1)^{\text{th}}$ element in the 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 elements after the array declaration is said to be **assignment**.

Example for initialization of array cells	Example for assigning values to array cells
<pre>int a[5]={10, 20, 30, 40, 50};</pre>	<pre>int a[5]; a[0]=10; a[1]=60; a[4]=70;</pre> <p>un-assigned cells contain garbage values.</p>

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 { }

Let us see following examples one by one.

`int A[5]={10,20,30};`

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]

Here compiler takes array size as count of initializing values. Here it is 5

`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 values

`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 < initial values count

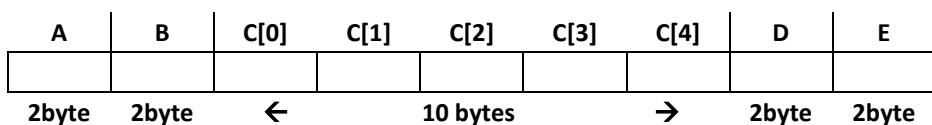
`int A[3];`

garbage	garbage	garbage
A[0]	A[1]	A[2]

If no values are initialized then default values are garbage.

About array size

At the time of compilation, the memory space for variables, including arrays, is logically arranged in the order in which they are declared in the program. For example: `int A, B, C[5], D, E;`



A total of 18 bytes of memory is allocated for all these variables, including the array. Generally, these variables occupy contiguous memory locations, as shown in the diagram. If the array size 5 is not specified, the compiler cannot allocate memory for the subsequent variables (D, E). This is because, at compile time, the memory for all variables is arranged logically in the order of their declaration. Without knowing the array size, the compiler cannot determine how much space to reserve, making it impossible to lay out memory for the following variables. So array-size must be given while declaration and it must be integer 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 is not allowed)

eg3) int n=5;
     int a[n];      // invalid declaration, the size must be a constant, the 'n' value '5' is available at the time of running, but
                      // space is allocated 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 (or fixed-size arrays). They are used when the size of the array is known at coding time. For example, a phone number may require 12 bytes, a name 30 bytes, a pincode 6 bytes, and an address 50 bytes.

However, sometimes the array size (i.e., data size) cannot be determined during coding. In such cases, dynamic arrays are used as an alternative. We will cover these at the end of the chapter, using functions such as `malloc()`, `calloc()`, `realloc()`, and `free()`.

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++)
{
    scanf("%d", &a[i] );
}

input:
Enter 5 values: 7 18 12 10 15 ↴
  
```

scanning 'n' values to array

Sometimes we cannot predict the exact number of input values in advance, as the count may vary from time to time, In such cases, we first scan (read) the number of input values, and then scan the corresponding array elements.

For example, in a college, the classroom capacity may be 40, but the number of students joining can vary each year, say between 20 and 40. In this case, we first read the actual number of students who joined, and then read their data. The following example demonstrates how to read the marks of n students and calculate the average marks of the class:

```
int main()
{ int a[40], i, n, sum=0;
  printf("enter no.of students joined:");
  scanf("%d", &n);
  if( n>60 )
  { printf("error, array size <40 ");
    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
```

Checking array boundaries

The compiler does not check whether an array index is within a valid range or not, because such a provision is not available in C. Therefore, it is the programmer's responsibility to ensure that the index value is valid before using it.

The index must always lie within the range 0 to arrSize - 1. Otherwise, the program may attempt to access unauthorized memory outside the array's range, which can cause unpredictable results or even a crash.

For example, **int A, B[3], C, D;**

A	B[0]	B[1]	B[2]	C	D	
← →						← This is beyond array limits memory →

```
A=10; C=20; D=40;
B[1] = 300; // valid instruction
B[2] = 400; // valid instruction
B[3] = 500; // in-valid instruction
B[4] = 600; // in-valid instruction
B[-1] = 700; // in-valid instruction
```

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 or instructions are damaged and processor tries to execute such damaged instructions then it leads to program crash.

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.

```
int 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 because we are dealing with the marks of only 4 students. However, if the number of students increases, the program becomes difficult to write and maintain, as it requires repeated use of many `printf()` and `scanf()` statements. This is called hard coding and makes the program cumbersome.

For such problems, arrays provide an effective solution. Let us now try the same program using arrays:

```
int main()
{
    int marks[4], total, avg, i;
    for( i=0; i<4; i++)
    {
        printf("enter marks %d ::", i );
        scanf("%d", &marks[i]);
    }
    for(i =0; i<4; i++)
        total = total + marks[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.

Demo: Converting all even numbers to next odd numbers

Let an array A[] contain 5 initial values. Converting all even values into their next odd values.

```
int main()
{
    int A[5]={10, 14, 20, 16, 19 };
    for( i=0; i<5; i++)
    {
        if( A[i]%2==0)      // if even
            A[i]++;          // A[i]++ changes to next odd
    }
    for( i=0; i<5; i++)
    {
        printf("%d ", A[i] );   output → 10 15 21 17 19
    }
}
```

Finding how many 3 divisibles exist in the array. Let array has 5 values.

Logic: Divide each A[i] by 3 and count them when they divided perfectly, like if A[i]%3==0 then count++;

```
int main()
{
    int a[5], i, count=0;
    printf("enter 5 values to array:");
    for( i=0; i<5; i++)
    {
        printf("enter value of a[%d] : ", i );
        scanf("%d", &a[i]);
    }
    for( i=0; i<5; i++)
    {
        if( a[i]%3 ==0 )
        {
            count++;
        }
    }
    printf("no.of 3 divisible are: %d", count);
}
```

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] : 11

output: no.of 3 divisible are: 2

Finding biggest in the array

Here, to find the largest value in the array, we used a variable bigg. Initially, we assigned bigg = 0. Then, each element A[i] was compared with bigg. If bigg < A[i], we updated bigg with the value of A[i].

At the end of the loop, bigg contained the largest value in the array.

```
int main()
{
    int a[5], bigg, i ; // here bigg is to hold big-value;
    printf("enter 5 values to array : ");
    for( i=0; i<5; i++)
    {
        scanf("%d", &a[i]);
    }
    bigg=0; // to clean the garbage with zero
    for(i=0; i<5; i++)
    {
        if( bigg < a[i] )
            bigg=a[i];
    }
    printf("\n biggest is= %d ", bigg);
}
```

input: enter 5 values to array : 4 12 43 6 19 ↵

output: biggest is 43

If all the input values are negative (for example: -12, -45, -6), the condition if (bigg < A[i]) will always fail, because the initial value of bigg is 0. Since no array element is greater than 0, the program will incorrectly print 0 as the largest value.

To solve this problem, instead of initializing bigg to 0, we should initialize any one element in the array, better initializing with the first value (bigg = A[0]). This way, the comparison will work correctly even all values are -ve.

Checking at least one value in the array is –ve or not?

```
int main()
{
    int a[5], i, bool=0 ;
    for(i=0; i<5; i++)
    {
        printf("enter value of a[%d] : ", i );
        scanf("%d", &a[i]);
    }
    for( i=0; i<5; i++)
    {
        if( a[i] < 0 )
        {
            bool=1;
            break;
        }
    }
    if(bool==1) printf("yes, -ve value exist");
    else printf("no, -ve is not exist");
}
```

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[3] : 12

output: yes, -ve value exist

Checking all values are in ascending order or not?

Let array A[] has 5 values, if any pair of elements satisfies $A[i] > A[i+1]$, then the array is not in ascending order.

The following logic is partially correct; correct the mistake with Boolean logic.

```
int main()
{
    int a[5], i, bool=0 ;
    for(i=0; i<5; i++)
    {
        printf("enter value of a[%d] : ", i );
        scanf("%d", &a[i]);
    }
    for( i=0; i<4; i++) // observe i<4, should not be i<5
    {
        if( a[ i ] < a[ i+1 ] )
            printf("yes, in order");
        else
            printf("not in order");
    }
}
```

if input is: 13 12 18 3 55

output: In first cycle of loop, the output is : $13 < 12 \rightarrow \text{false} \rightarrow \text{not in order}$

In second cycle of loop, the output is : $12 < 18 \rightarrow \text{true} \rightarrow \text{yes, in order}$

In third cycle of loop, the output is : $18 < 3 \rightarrow \text{false} \rightarrow \text{not in order}$

The output is displayed for each pair of comparisons inside the loop. However, in this case, we should use Boolean logic to display a single output after the loop. Try it yourself. The output should be "yes" or "no"

Searching for element in the array ?

Let array A[] has 10 values, all elements are in ascending order and initialized; our task is to search for a given input element. If searching element is found then print “found” or else “not found”.

```
int main()
{
    int a[10]={24, 28, 29, 34, 45, 78, 89, 99, 100, 120 };
    int searchValue;
    printf("enter element to search:");
    scanf("%d", &searchValue);
    for( i=0; i<10; i++)
    {
        if( a[ i ] == searchValue )
            break;
    }
    if( a[i] == searchValue )
        printf(" found");
    else
        printf("not found");
}
```

If the search value is 45, then the output is displayed smoothly (“found”).

If the search value is 55, the program unnecessarily compares with all elements, even when $a[i] > searchValue$. Another issue may arise: after the loop, the value of ‘i’ becomes 10. So, the comparison $if (a[10] == searchValue)$ causes the program to crash, because $a[10]$ is out of the array limits. Correct this program yourself.

Printing 2nd odd value in the array

Scan N values to array, where $N < 10$, and our task is to print 2nd odd value. If odd is not found then display error

```
int main()
{
    int a[10] , N, count=0, i ;
    printf("enter How many input values: " );
    scanf("%d", &N );
    if( N>10)
    {
        printf("\n error, N should be <10 ");
        return 0;
    }
    printf(" enter %d values to array : " );
    for( i=0; i<N; i++)
        scanf("%d", &a[i] );
    for( i=0; i<10; i++)
    {
        if( a[i]%2 == 1)
        {
            count++;
            if( count==2)
                break;
        }
    }
    if(count==2) printf("the odd value is %d", a[i] );
    else printf("the second odd not found" );
}
```

Finding no.of days in a given month

```

ip: 2 2003          ip: 4 2009
op: 28 days         op: 30 days
int 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[] is pre-initialized with the number of days in each month relative to the month index.

For example:

```

a[1] → January → 31 days
a[2] → February → 28 days ( ignoring leap-year )
a[3] → March → 31 days
a[0] → 0 (dummy value)

```

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
int 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); // if y%4==0 is true then it gives 1. Leap-year has 29 days.

    if( m<0 || m>12 || d<1 || d>arr[m] )
        printf(" invalid date");
    else printf( " valid date");
}

```

Finding ${}^N C_R$ of two numbers

```

ip: 4 2              ip: 7 1
op:  ${}^N C_R = 6$       op:  ${}^N C_R = 1$ 

int main()
{   long int a[10]={ 1,1,2,6,24,120,720,5040,... }; // array pre-initialized with factorial values.
    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 is pre-initialized with all factorial values. Therefore, the nCr value can be easily obtained by substituting these values into the equation. Finally, the result is printed.

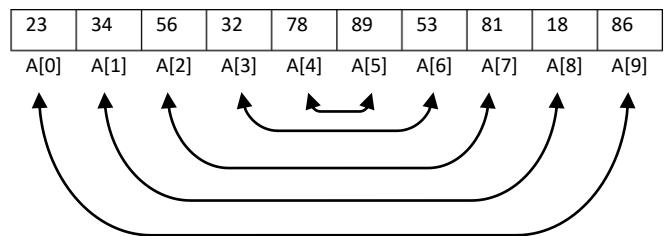
Reversing array elements

```
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

```
int main()
{
    int a[100], i, j, n, temp;
    printf("enter no. of input values :");
    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)
```

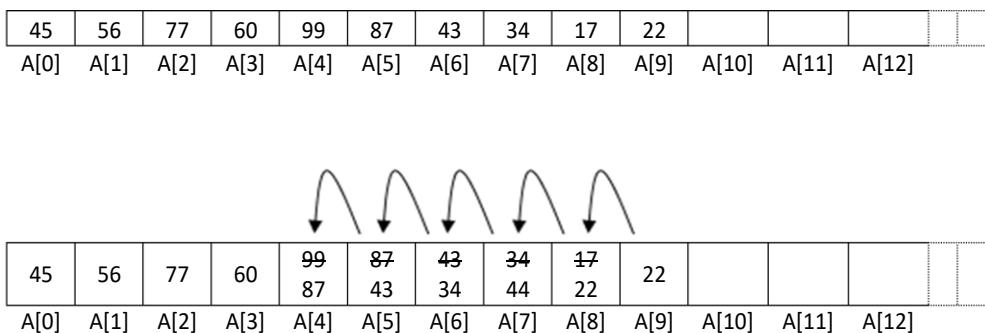
The following array contained 10 values, and it gives demo how to delete 5th element.

The code to delete 5th element is as follows

```
for(i=5; i<10; i++)
    a[i-1] = a[i];
```

Here, this loop replaces a[4] by a[5], a[5] by a[6], ...etc.

Finally 5th element (a[4]) will be deleted by replacing with successive elements.



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

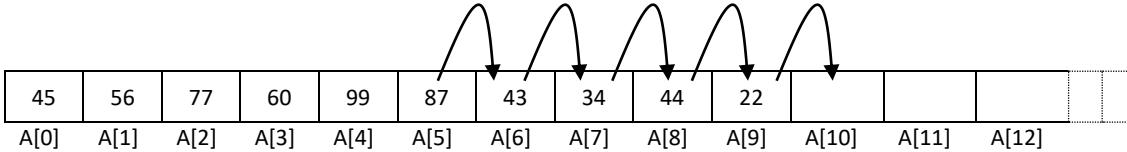
The program inserts a new element at the kth position in an existing array of n elements, where k < n.

To insert a new element at A[k], shift all existing elements A[k], A[k+1], A[k+2], ... A[n-1] one position to the right.

This creates a gap at A[k], where the new element can be inserted.

For example, to insert a new element 11 at A[5], shift all elements 22, 44, 34, 43, 87 one position to the right.

This creates a gap at A[5], where 11 can be inserted.



```
int main()
{
    int a[100], 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++;                  Since one element is added, the count of elements 'n' should be increased.
    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 ...

```
int main()
{
    int a[5]={14, 16, 13, 11, 24 };
    int i, j ;
    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
int main()
{ int a[100], i, bool, j;
printf("enter the no.of input values of array :");
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

```

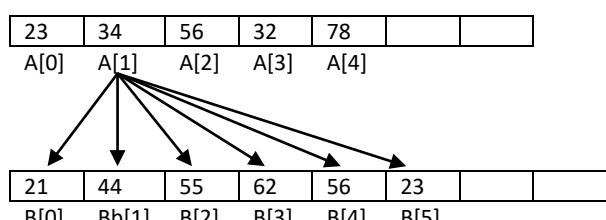
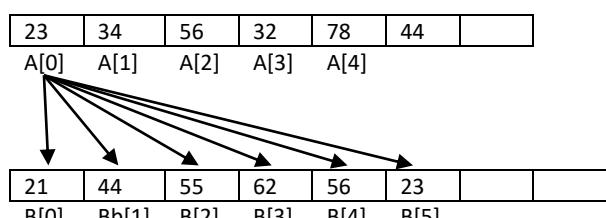
Let us say, two arrays a[] & b[] contained n1 & n2 number of distinct elements respectively.

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.

The following figure and code shows how the comparisons are made 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;
    }
  }
}

```

complete the total program yourself.

Printing all digits of N in ascending order

30) Let N value has non repeated digits like 9265 , now just print all digits in ascending order 2569

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 9265, 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	0	0	1	1	0	0	1

```
int n, i, a[10]={0};
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

$$sd = \sqrt{((A_0-M)^2 + (A_1-M)^2 + (A_2-M)^2 + (A_3-M)^2 + \dots + (A_n-M)^2) / N}$$

```
int 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);
}
```

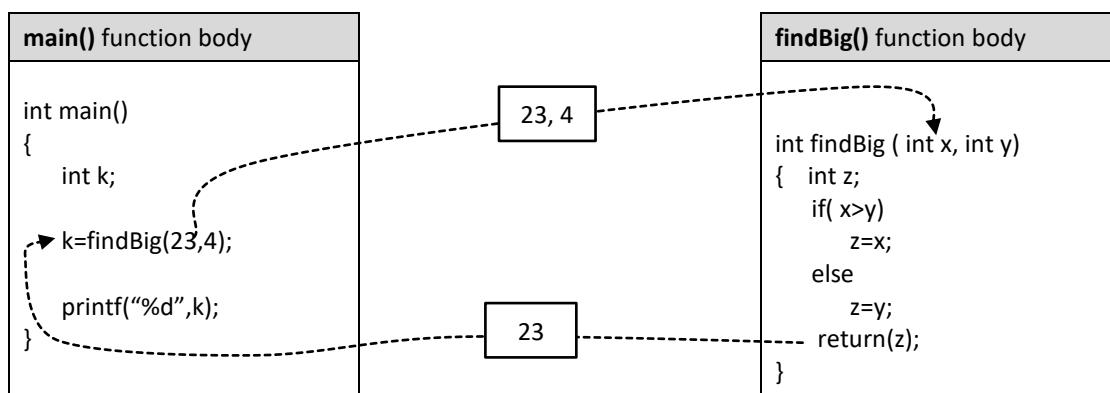
Introduction to functions

In real world, many tasks cannot be accomplished by a single person; instead, several people are often involved in completing one task.

For example, when your vehicle breaks down, you call a mechanic and hand over the vehicle to him. After repairing it, he returns the vehicle to you. In the same way, in programming, a function works like a mechanic. For instance, the **power()** function takes two input values—base and exponent—and after performing the calculation, it returns the result, just like the mechanic returning the repaired vehicle.

A function is a subprogram that performs a specific subtask in a program and is executed when called from another part of the program. In C, **main()** itself is also a function, and it serves as both the starting and ending point of the program. Here, the **main()** function can be compared to the vehicle owner, while the **power()** function represents the mechanic.

In functions, there are two key components: the **function body** and the **function call**. The function body defines the task of the function what it does, and it is executed only when it is called from another part of the program. Let us look at the following picture, which illustrates how functions are executed in a program.



The instruction **k = findBig(23,4);** is a function call statement in **main()**. When it is executed, the control jumps to the body of **findBig()** along with the values (23,4). These values are assigned to the variables (x, y). After calculating the bigger value (z), the function returns it, and it is then assigned back to k. This is how functions are executed, as shown in the picture above.

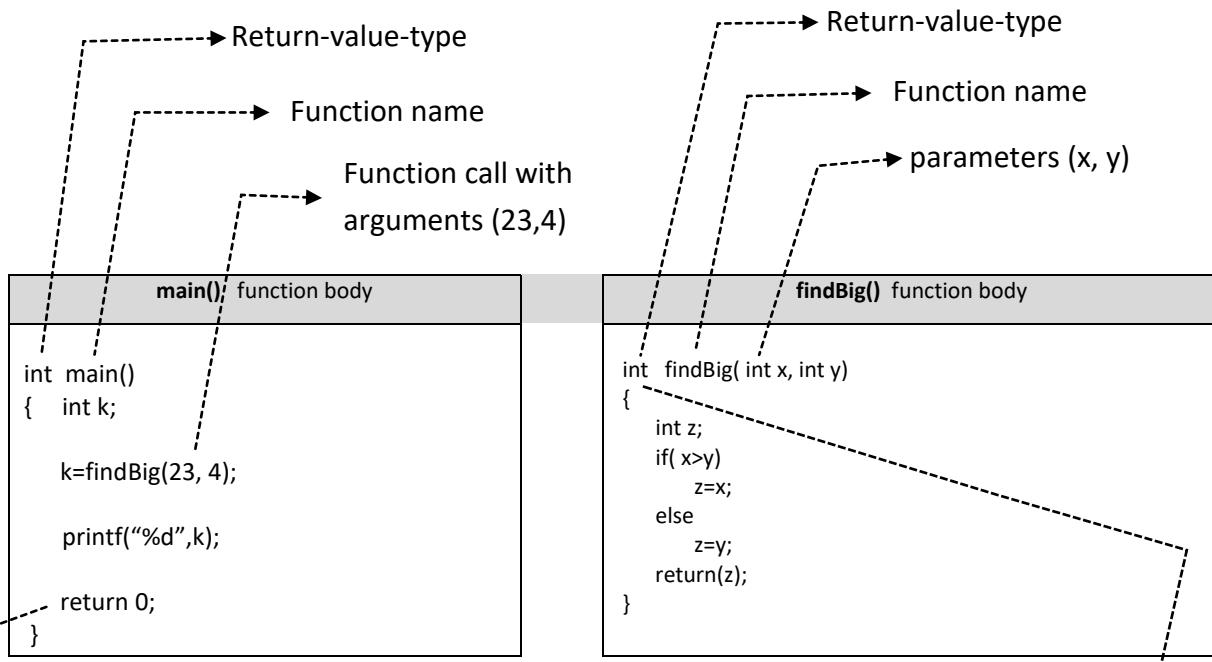
The values passed to a function are called **arguments**, whereas the variables that receive these values are called **parameters**. In this example, the values (23, 4) are arguments, and (x, y) are parameters. The number of arguments and parameters must match, and their types must be compatible.

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 data type **int** before the function name **int findBig(){ }** is called the **return-value-type**. In this example, the function is returning the value of ‘z’ as an integer, so the return-value-type is being declared as **int**. The return-value-type tells what kind of value the function returns. And it is useful for the compiler to check syntax errors as well as to the programmer for usage.

Thus, every function does some task and returns a calculated value to the calling-function. All functions, including main(), follow the same syntax rules with one or two optional statements. The following figure shows each entity of a function's syntax.



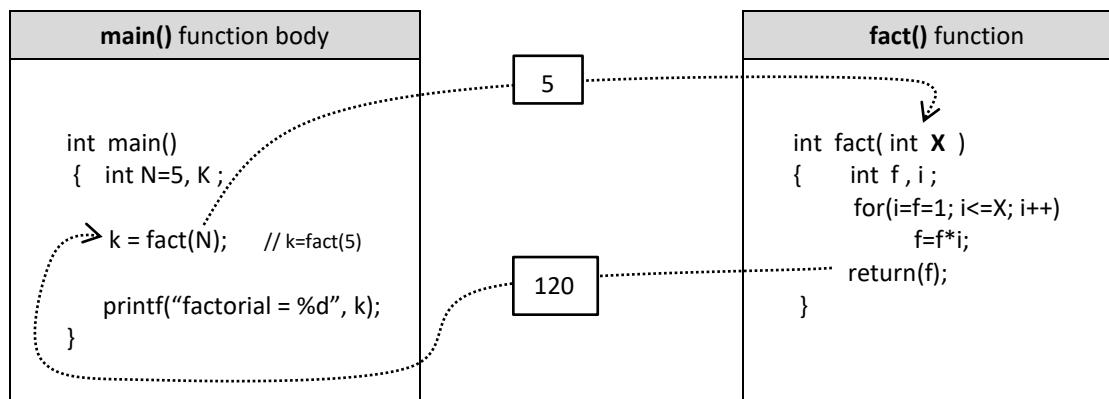
This is “**return-value-type**” of a function, this type must be matched with the return-value of a function. In this example ‘z’ is returning as int, therefore return-value-type declared as **int**.

The **main()** function also returns a value (0). This value is sent back to the operating system(OS), because the OS calls **main()** when we run a program. A return value of 0 indicates that the program finished successfully without errors, but we can return other values as well. If our program is called by another program, the OS passes this return value to that program.

Finding factorial of a given value using function

input: 5

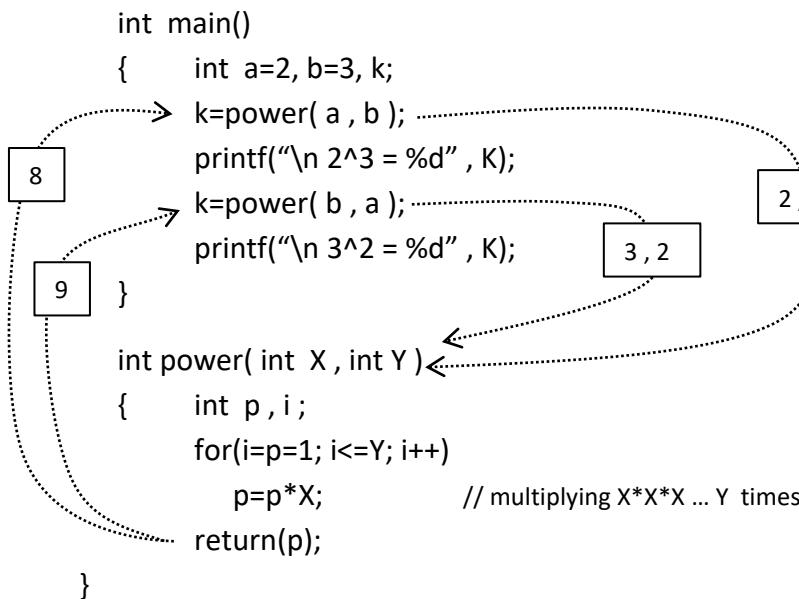
output: 120



The statement **k = fact(N)** is a function-call statement. When it is executed, control jumps from the **main()** body to the **fact()** body with the value 5, as shown above. This value is stored in the parameter **X**, and the function then returns the calculated factorial value (120). This value is assigned back to **K**. (like **k=f**). In this way, the function can be called whenever its calculation is needed.

Finding power of 2^3 and 3^2 using power() function

The power() function takes two arguments(X,Y) as base and exponent and returns the X^Y value.



Here the power() function is called 2 times for 2^3 and 3^2

at 1st call, the values 2,3 is passed as an arguments like: $k=\text{power}(a,b) \rightarrow k=\text{power}(2,3)$; since $a=2, b=3$

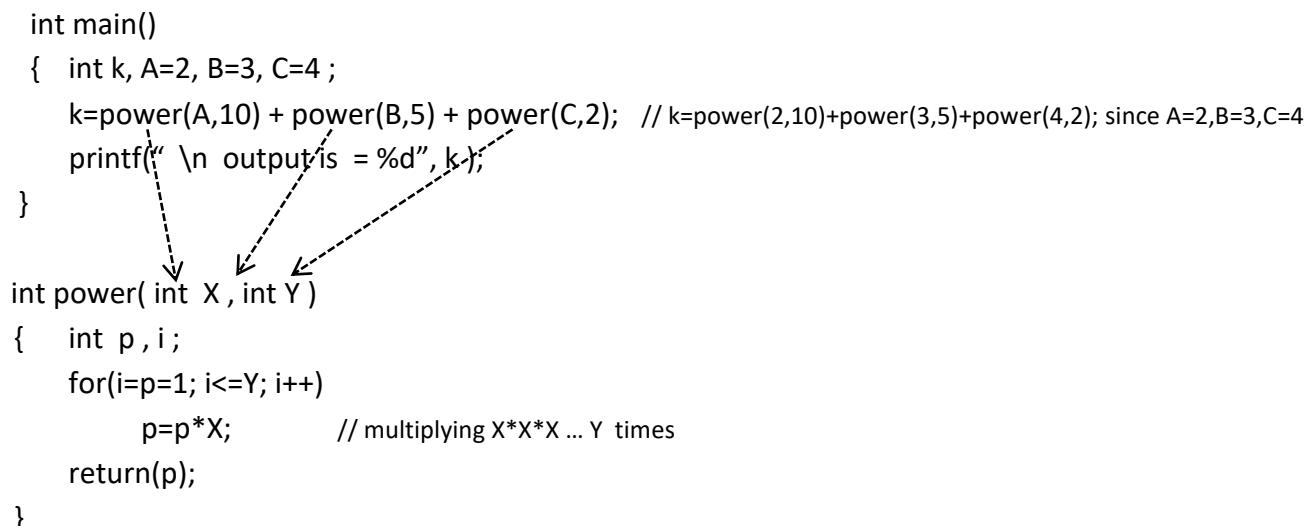
at 2nd call, the values 3,2 is passed as an arguments like: $k=\text{power}(b,a) \rightarrow k=\text{power}(3,2)$;

In this way function can be called any number of times with different arguments.

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 X^Y value.

Later the main() function calls power() function to find value of equation: $A^{10} + B^5 + C^2$



Let us see the above instruction: **$k=\text{power}(A,10) + \text{power}(B,5) + \text{power}(C,2)$** ;

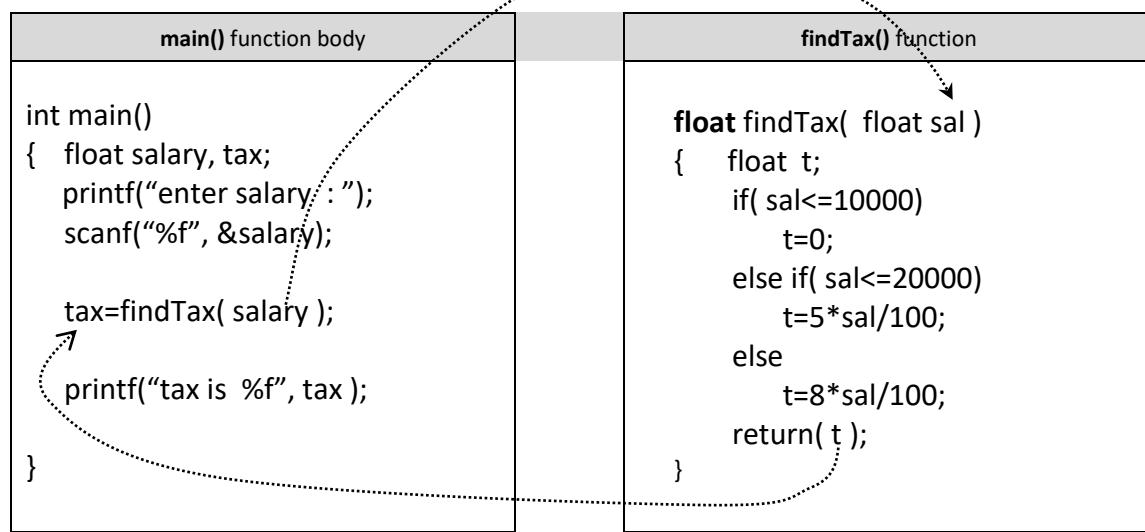
The call statement is composed as: **$k=\text{power}(2,10) + \text{power}(3,5) + \text{power}(4,2)$** ; // since $A=2, B=3, C=4$;

So this statement calls the power function 3 times for A^{10} , B^5 , C^2 ;

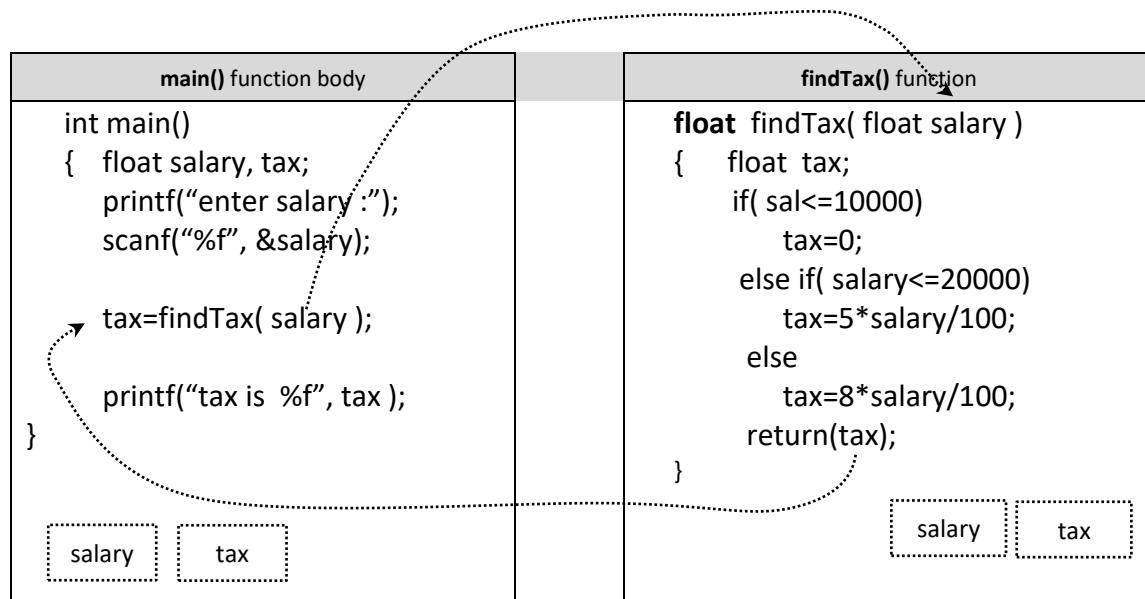
the 1st call with arguments(2,10), the 2nd call with arguments(3,5), and 3rd call with (4,2).

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%
```



Here, the argument name is '**salary**' while the parameter name is '**sal**'. When the function is called, the value of **salary** is assigned to **sal** (like **sal=salary**). Sometimes the argument and parameter names can be the same, but they are different copies.(compiler manages with different memory address).Let us see the following program.



Here, the argument and parameter variables have the same name **salary**, but they are actually two different copies. Beginners often get confused and think a single variable is shared between two functions.

Let us consider these two statements:

1) Suresh has given a pen to Ramesh. 2) Suresh has given a pen to Suresh.

The second statement seems confusing because both the giver and taker have the same name, **Suresh**. But in reality, they are two different people with the same name. In the same way, argument and parameter can be same name, but we don't need to worry, compiler manages with different addresses. The argument 'salary' belongs to `main()` function whereas parameter 'salary' belongs to `findTax()` function.

Actually, between functions, only values are exchanged while calling and returning—not the variable names.

In this program, there are two sets of variables (`salary` & `tax`), totally 4 variables. One set belongs to `main()`, and the other set belongs to `findTax()`, as shown in the picture above.

Finding biggest of 2 & 3 integers using 2 functions

In the following example, big2() finds the biggest of 2 numbers, and big3() finds the biggest of 3 numbers.

```
int main()
{
    int a=4, b=76, c=10, k ;
    k=big2( a , b );
    printf("\n biggest is %d", k );
    k=big3( a , b , c );
    printf("\n biggest is %d", k );
}

int big2( int a , int b ) <
{
    if (a>b) return a;
    else return b;
}

int big3( int a , int b , int c ) <
{
    if (a>b & a>c)
        return a;
    else if( b>c)
        return b;
    else return c;
}
```

Finding sum of 1 to N using function ($1+2+3+4+5+\dots+N$)

The following findSum() function calculates the sum of $1 + 2 + 3 + \dots + N$ without using a formula. This function takes N as an argument and returns the sum from 1 to N. Later, the main() function checks whether the value returned by findSum() is equal to the formula $N(N+1)/2$. If it is equal, the program prints “Program is correct”; otherwise, it prints “Program has some logical errors..”

```
int 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;
}
```

More about functions

A function can be defined as a subprogram, or an independent block of code with a name, designed to accomplish a particular task. In computer science, a function is also called a routine, subroutine, subprogram, procedure, method, or module.

Functions break down a large program into several smaller subprograms, making it easier to write, understand, debug, modify, and maintain, while also improving portability and reusability. Thus, a collection of functions together makes a well-structured C program.

The main advantages of functions are reducing repetition of code and reusability of code.

Reduces repetition of code: If a piece of code is needed in several places in the program, we can put it into a function and then call that function whenever required.

Reusability of code: A well-developed function can be reused in many programs by many people, just like the built-in functions `printf()` and `scanf()`.

Generally, a large application cannot be developed by a single person, where several people are involved. When developing a large application, the program should be divided into meaningful modules, and each module can be further divided into several sub modules called functions.

For example, consider a student automation project in a school. It can be divided into modules such as admission, attendance, exam-score, fees, and games. Each module is made up of several functions to accomplish individual tasks. Typically, every module is developed in a separate file like “admission.c”, “attendance.c”, “examScore.c”, etc. All these files may be developed by different 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 manufacturers to provide solutions for basic and routine tasks in programming. For example I/O functions `printf()` & `scanf()`, mathematical functions `pow()` & `sqrt()`, etc.

The vendor of C, supply these predefined functions in compiled form along with the 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: User-defined functions: We can write our own functions as per our requirements, just like any library function. Conceptually, there is no difference between user-defined and library functions; both work in a similar manner.

Moreover, our functions can be added to an existing library and used just like built-in library functions. If we have a larger collection of such functions, we can even create our own library.

In C, the `main()` function is also a user-defined function. The word `main` is reserved by the compiler, but its body must be implemented by the programmer. The `main()` function acts as the starting and ending point of a program. Other functions are executed by temporarily transferring the control from `main()`.

There is no syntactical difference between `main()` and other functions; all functions work in a similar way.

Syntax of function

Function has 3 syntaxes

1. **Function definition/body** (defines the code of function)
2. **Function calling/invoking** (executes the task of the function)
3. **Function proto-type** (like declaration a variable, 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 the function—that is, what it does. It specifies the task using input and output values (arguments and return value). The body executes only when the function is explicitly called from another part of the program. Every function follows the same syntax, with one or two optional statements.

② Syntax of function-call statement

variable = function-name(arguments-list);

Calling a function means asking it to perform its task. When a function call is executed, the control moves to the function's body along with the arguments. After all the instructions in the body are executed, the control returns to the same point where the function was called.

To understand the mechanism of how a function call works, consider this example: Suppose the owner of a house wants some groceries. He gives money to his assistant and asks him to buy the groceries. Later, the assistant returns with the groceries. In this example, the owner is like `main()`, and the assistant is like `findBig()`. When `main()` wants to find the bigger of two numbers, it calls `findBig()` and gives it the arguments `(23, 4)`. The `findBig()` returns 23 as result. This is how functions work.

Function name: The function name should be relevant to its task and should express what the function does. For example, if a function finds the square root of a value, it is better to name it `sqrt()`.

Calling vs Called: Here, `main()` calls the `findBig()` function. In this context, `main()` is called the calling-function, while `findBig()` is called the called-function.

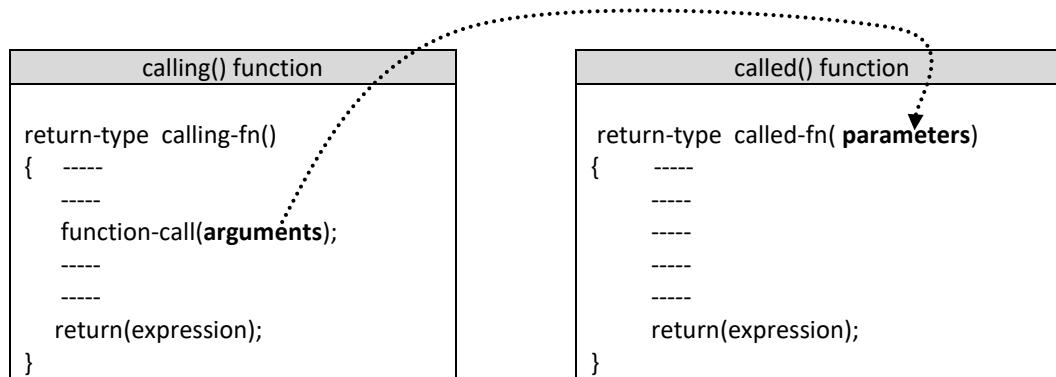
Arguments VS Parameters: Arguments are the input values given to a function. When a function is called, these values are passed from the calling function to the called function along with control. Parameters are the variables in the called function that receive (hold) the argument values. (In short, arguments are values, whereas parameters are variables.)

Arguments belong to the calling function, while parameters belong to the called function.

The count of arguments and parameters must match, and their types must be compatible. For example, if an argument is `int`, the parameter can be `int`, `long int`, or `float`.

In **older definitions**, arguments were called actual arguments or actual parameters, while parameters were called formal arguments or formal parameters. (Actual → Formal)

* The following figure illustrates the relation between argument and parameters.



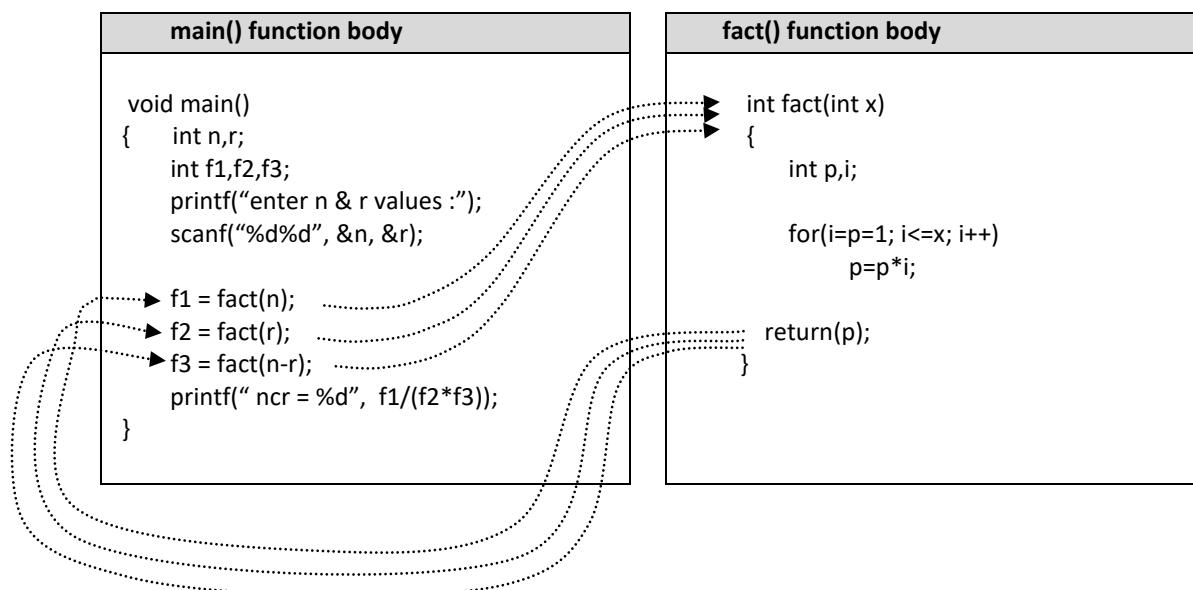
return statement: The return statement ends the current function's task and returns control to the calling function, with or without a value.

Function's limitation: A C-function can return only one or none values using the 'return' statement because the syntax allows only that. To return more values, pointers are used (we will see this later).

Note: You can write functions in any order. For example, main() can be written at the bottom. Don't worry, the compiler automatically moves it to the beginning in the executable file, because it is the starting point of the program. Senior programmers usually write the main() function at the bottom of the program. Let us have some more example programs.

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() function as given below



Let input $n=7, r=3$; then fact() function calls as given below

```

f1 = fact(7);      // here the value 7 is passed as an argument to fact()
f2 = fact(3);      // here the value 3 is passed as an argument to fact()
f3 = fact(7-3);    // here the value 4 is passed as an argument to fact()

```

In the first call, the control goes with the argument 7, and the function returns 5040 (7!), which is assigned to f1. In the second call, the control goes with the argument 3, and the function returns 6 (3!), which is assigned to f2. In the third call, the control goes with the argument 4, and the function returns 24 (4!), which is assigned to f3. In this way, reusability of code is possible with the functions whenever it is required.

Finding sum of $(1) + (1+2) + (1+2+3) + \dots N$ terms

```

int findSum ( int N ) <----- dashed oval
{
    int i , sum=0;
    for(i=1; i<=N; i++)
    {
        sum=sum+i;
    }
    return sum;----- dashed oval
}
int main()
{
    int i , N, sum=0;
    N=4;          // or scan from keyboard
    for(i=1; i<=N; i++) ----- dashed oval
    {
        sum = sum + findSum(i); ----- dashed oval
    }
    printf("\n sum is %d" , sum );
}

```

Note: Functions can be written in any order in a program. The main() function can be written at the bottom as shown above. However, execution always starts from the main() function.

Finding summation of $x^1/1! + x^2/2! + x^3/3! + \dots + x^N/N!$

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.

```

int main()
{
    float sum=0, v1, v2;
    int i;
    for(i=1; i<=N; i++)
    {
        v1=power(x , i );----- dotted oval
        v2=fact(i);----- dotted oval
        sum=sum+v1/v2;
    }
    printf("sum is %f", sum);
}

```

```

int fact( int N ) <----- dotted oval
{
    int i , f;
    for(i=f=1; i<=N; i++)
        f=f*i;
    return f;
}

```

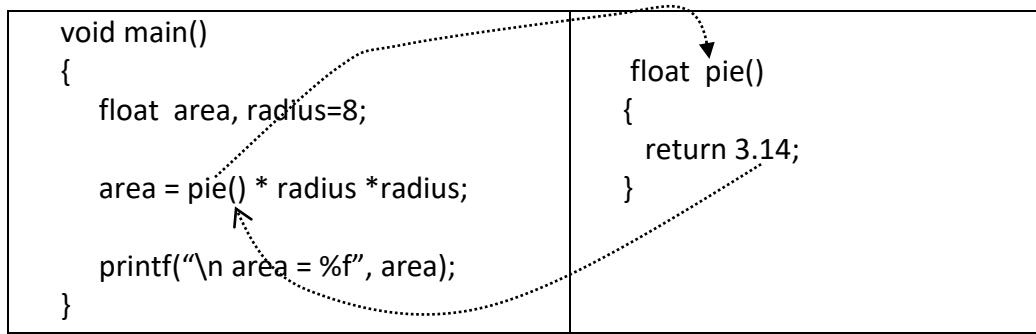
```

int power( int x , int y ) <----- dotted oval
{
    int i , p;
    for(i=p=1; i<=y; i++)
        p=p*x;
    return p;
}

```

A demo program how return-value substitutes at fn-call statement

If a function returns a value, then the function-call statement can be used as a variable or value in the program, because the return value substitutes in place of the function call. Let us see how this works.



When the `pie()` function is called in `main()`, it returns the float value 3.14 and substitutes it at the function-call statement, so the expression becomes `area = 3.14 * radius * radius`.

Actually, the function's return value is first stored in a temporary variable (the temp is automatically created by the compiler). This temporary variable value then substitutes at the function-call statement. Thus, the above function call can be imagined as:

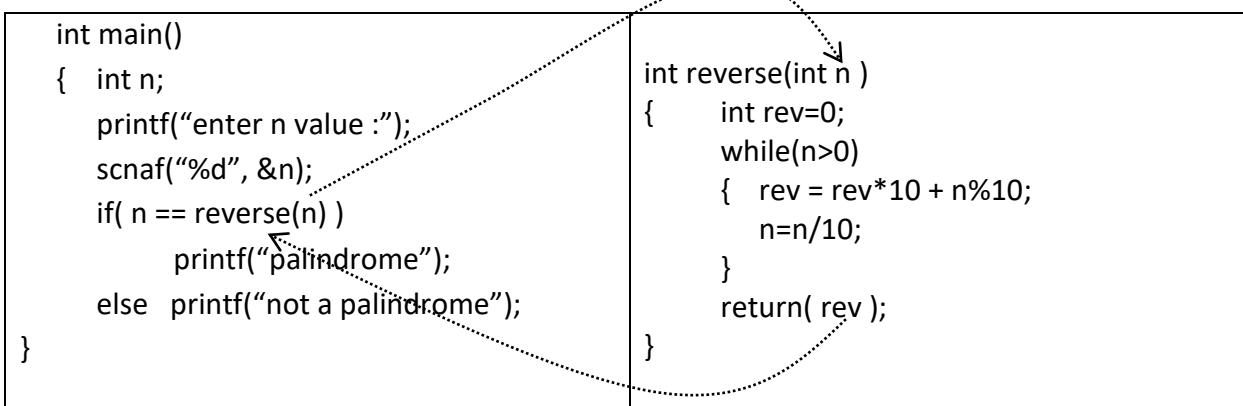
```

int main()
{
    float area, radius=8;
    temp=pie(); // here this 'temp' will be created by compiler
    area = temp * radius *radius;
    printf("\n area = %f", area);
}
  
```

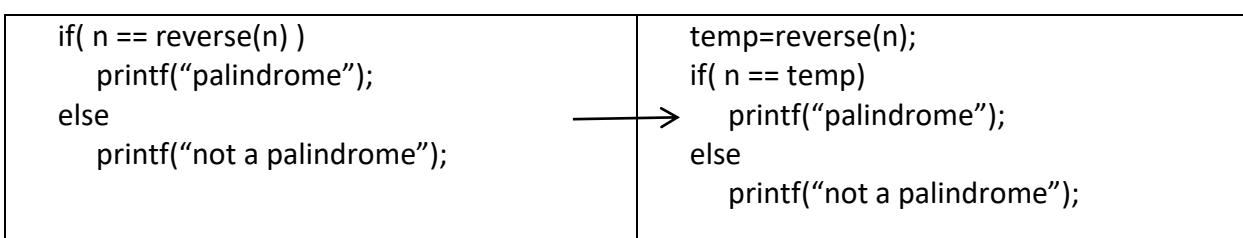
Note: The temporary variable `temp` is a nameless variable, created and managed internally by the compiler.

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?



As explained in the above example, the function's return value is stored in a temporary variable, and this value substitutes in place of the function call. Thus, the function call can be imagined as shown below:



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 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 combined into a single line as: k=big(x, big(y,z))

Let x = 23, y = 42, and z = 31. Then the function calls proceed as follows

k = big(23, big(42,31)) → k=big(23,42) → k=42

First, the inner call big(42, 31) is executed, which returns 42. This value is then substituted into the outer call as the second argument. Finally, the outer call is executed as k = big(23, 42);

Finding given number is Armstrong or not?

This function takes an integer N as an argument and checks whether it is an Armstrong number.

If it is, the function returns 1; otherwise, it returns 0. Later, the main() function reads N as input and prints whether it is an Armstrong number or not.

```
int isArmstrong(int n)
{
    int t = n, sum = 0, r;
    while (n > 0)
    {
        r = n % 10;
        sum = sum + r * r * r;
        n = n / 10;
    }
    if (t == sum) return 1;
    else return 0;
}

int main()
{
    int n;
    printf("Enter n value: ");
    scanf("%d", &n);
    if ( isArmstrong(n) == 1) printf("Yes, it is Armstrong\n");
    else printf("No, not Armstrong\n");
    return 0;
}
```

Printing list of Armstrong numbers between 1 to 1000 (using above fn)

Output: 1, 153, 370, 371, 407.

```
int main()
{
    int n;
    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); // the return statement stops the loop and returns the control with value 0 as not prime
    }
    return(1); // not at all divided, so returning '1' as prime
}
int 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.

```
int 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 3 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=3; n<100; n=n+2) // even numbers can't be primes except 2, so it is wise to check only odd numbers
{ if( isPrime(n) && isPrime(n+2) )
    printf("\n %d → %d ", n, n+2 );
}
```

Function proto-type

A function prototype is a function declaration, similar to a variable declaration. Before learning about prototypes, we should first understand why they are required.

```
line 1 → int main()
line 2 → {     float k;
line 3 →         k =add(20, 30);
line 4 →         printf("%d ", k );
line 5 → }
```

```
line 6 → float add(float a, float b)
line 7 → {     return a+b;
line 8 → }
```

In C, functions are compiled in the order they appear in the program. Here, `main()` is compiled first, and `add()` comes later. When the compiler encounters the function call `k = add(20, 30)` at line 3, it does not yet know about the `add()` function and its definition, which appears later at line 6. However, based on the arguments (20, 30), the compiler assumes that the function takes two int arguments and returns an int. The default return type is int.

So, based on the function call statement at line 3, the compiler assumes the function definition to be:

`int add(int a , int b) { ... }`. But when compiling the actual function definition at line 6, the compiler encounters it differently as: `float add(float a , float b) {...}`. This causes an error called **type mismatch** for the `add()` function.

To solve this problem, we have two options:

- 1) Declare the function prototype before the first call.
- 2) Write the called-function above the calling function.

The English word prototype means a model to an actual implementation, similar to a building plan for a building. Prototype is used to **introduce the function to the compiler**, giving information about the return type, function name, parameter count, and parameters types.

For example, `int findBig(int , int);` → this is the proto-type of the `findBig()` function.

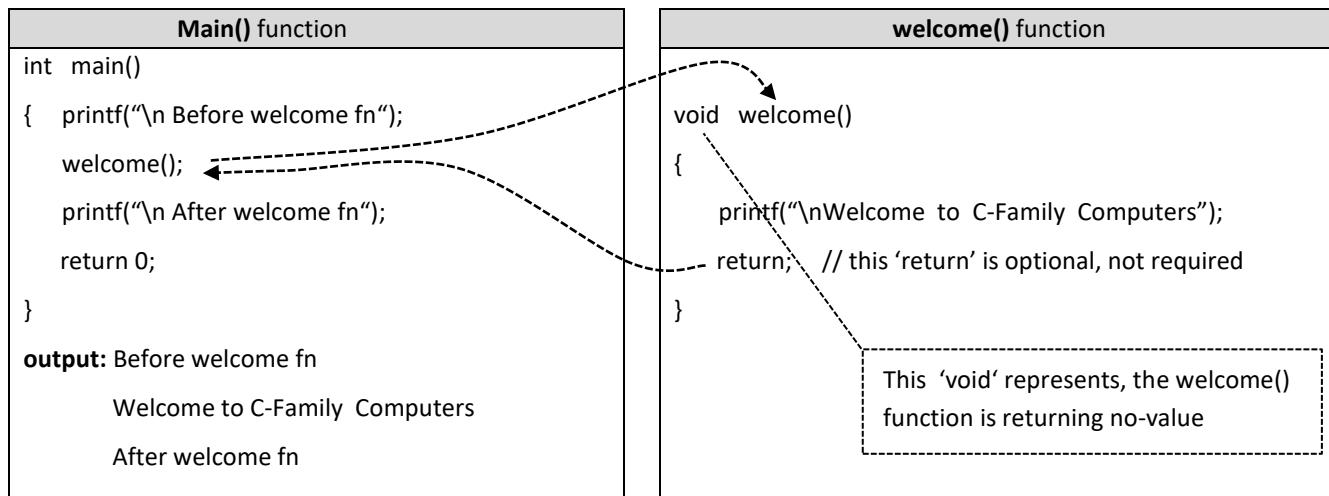
A function prototype is the function header followed by a semicolon (a function header without a body). Here, this prototype tells us that the function takes two integer arguments and returns an int value. So, the prototype gives complete information to the compiler about the function. In this way, pre-assumptions and mismatches are avoided. Let us now see the solutions for the above program.

Solutions: with Proto-type	Solution2: function-body before function-call
<pre>float test(float , float); // function proto-type int main() { --- k=test(); --- } float test(float a, float b) { return(a+b); }</pre>	<pre>float test(float a, float b) // this body itself works as proto-type { return a+b; } int main() { --- k=test(); --- }</pre>

About “void” data-type

If a function does not return any value, then its **return-value-type** should be declared as **void**.

The keyword **void** is also a data type like int, float, char. It represents empty/null type.



The above **welcome()** function is just displaying a message on the screen, here it is taking no argument and returning no-value. Therefore, the return-value-type '**void**' is used in this example. The last closing braces of a function works as a 'return' statement, so writing 'return' at the end of a void function is optional.

Printing multiplication table(s) up to 10 terms

The **printTable()** function given below takes the table number N as an argument, prints the first 10 terms, and function returns nothing because it does not perform any calculation. The **main()** function calls this function to print the tables of 3, 6, and 12.

```

void printTable( int n )
{
    int i;
    if( n<0 )
    {
        printf("error, table Number should not be -ve");
        return; // this returns the control to main() function without printing table.
    }
    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
} // this closing braces works as 'return' , so above 'return' statement is not required

int 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 does not calculate any value; it only prints the table on the screen. Since it does not return any value, the return type **void** is specified here.

Printing all multiplication tables from 3 to 19

```

void printTable( int n )
{
    int i;
    for( i=1; i<11; i++ )      // loop to print 10 terms in a table
        printf(“\n %d * %d = %d”, n, i, n*i);
    return;      // we know this ‘return’ statement is optional
}
int main()
{
    int i;
    for( i=3; i<20; i++ )
        printAllTables( i );
}

```

Program prints 3-digit number in English words

The function `printDigit()` prints the given digit in English words. For example, if the input argument is 4, it prints “four” as an English word. This function returns nothing because it does not perform any calculation; it simply prints the given digit on the screen. Now, using this function, we can write a `main()` function that scans a 3-digit number (like 247) and prints the output as “two-four-seven”.

```

void printDigit( int n);           // function proto-type
int 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;
    }
}

```

exercise: update above `main()` function using loop to print 4-digit number.

A demo program, how multi-level calls are made

Example showing how the control jumps through several levels of function calls and then returns to the same point of call.

```
int 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

The function x() is considered both a called-function as well as calling-function.

The main() function calls x(), and x() calls y(). Therefore, depending on the context, x() acts as both a called-function and a calling-function.

The order in which functions are written in a program does not have to match the order in which they are called.

In fact, any function can call any other function, regardless of the order in which they are written.

Functions cannot be nested like control structures, because each function body is independent.

A program can have any number of functions.

In programming, to accomplish a task, one function may take the assistance of other functions, and those functions may in turn take the assistance of still other functions. In this way, software is designed and developed as a collection of functions, often contributed by many people in the industry.

Scope of variables

Scope means a region, boundary, or area that specifies a limited or enclosed space. In C, scope is defined using braces { }. If a variable is declared inside a block scope, it becomes local to that block and cannot be accessed outside it. If we try to access it outside the block, the compiler shows an error. For example

```
int main()
{
    { int k;
        k=100;
    }
    printf("%d ", k); // error, undefined symbol 'k'
}
```

Here compiler shows an error message called “**undefined symbol k**” at the printf() statement, because the variable ‘k’ is declared inside the inner block, and we are trying to access print it outside. Thus, ‘k’ cannot be used outside that block. In this way, variables have block-scope. Let us see one more example

```
int main()
{
    int k=100;
    test();
    printf("the k value =%d", k);
}
void test()
{
    k++; // error, undefined symbol 'k'
}
```

In this case also compiler shows same error message “**undefined symbol k**”. The variable ‘k’ is declared inside the main() function block, and we are trying to access it in the test() function block. In this way, function body block is also considered a block, and we can’t access one block variables in another block.

Let us one more example

```
int main()
{
    { int k;
        k=100;
    }

    { int k;
        k=200;
    }
}
```

In this program, two separate copies of the variable k exist, and their memory addresses are different. The first k belongs to the first inner block, and the second k belongs to the second inner block. Let us one more example

```
int main()
{
    int k=90;
    test(k); // remember, here 90 is passed as an argument ( not 'k' is passed to the function)
}

void test( int k )
{
    printf("k value is %d ", k);
}
```

In this program, two copies of the variable ‘k’ exist. The first ‘k’ belongs to the main() function, and the second ‘k’ belongs to the test() function. Here, the first ‘k’ serves as an argument, whereas the second ‘k’ serves as a parameter. We will discuss the scope and accessibility of such variables in the chapter on **Storage Classes**.

Demo program, how arguments VS parameters behave

If the argument is a variable, then its value is passed and assigned to the parameter. As we know, the parameter is a copy of the argument. If any changes are made to the copy (the parameter), they do not affect the original (the argument). The following examples explain this in two ways.

Example 1	Example 2
<pre>int main() { int n=10; printf("\n before calling =%d", n); test(n); printf("\n after calling = %d ", n); } void test(int x) { x++; }</pre> <p>Output: before calling = 10 after calling = 10</p>	<pre>int 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++; }</pre> <p>Output: before calling = 10 after calling = 10</p>
<p>1) When the test() calls, the argument n value 10 passes and assigns to parameter x.</p> <p>Now, x is said to be copy of n. (like above picture) If any changes made in x, it doesn't affect the n.</p> <p>if one wants to change the n through x, then pointer are required. We will see in next chapter.</p>	<p>Here names of argument & parameter are same; both variable names are n, but they are different copies, So, if any changes made in parameter n, it doesn't affect to argument n.</p> <p>Both programs show same output.</p>

Swapping of two values

```
int 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 values of x and y in the main() function; instead, it only swaps the copies of x and y within its own function. As we know, if the parameter (copy) changes, it does not affect the argument (original) values. To swap the values of x and y in main() using the swap() function, the concept of pointers is used. We will discuss this in the next chapter.

Passing array values one by one to function and printing on the screen

```
int main()
{
    int a[5]={ 55, 64, 67, 89, 32 };
    int i;
    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

```
int 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. In this example, we use x, y, and z to receive/hold value of a[0], a[1], and a[2].

Generally, this logic is not recommended when an array has many elements. For this problem, we have a better solution using the concept of pointers.

More about functions

1) return & return-value-type

return statement: The job of the return statement is to terminate the execution of the current function and transfer the control back to its calling function, with or without a value.

syntax 1: **return;** // returning the control without value
 syntax 2: **return(value);** // returning with value, here parenthesis are optional

Functions can contain any number of return statements, but only one of them is executed. Once a return statement is executed, the control immediately transfers back to the calling function.

return-value-type:

The return-value type is the data type of the value that a function returns. If the return type is not mentioned explicitly, the compiler assumes the default type as **int**. Specifying the return type explicitly indicates what type of value the function returns. This is useful for the compiler to check for syntax errors and also for the programmer to clearly understand what value the function is returning. Let us see some code examples

<pre>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.</pre>	<pre>float test() { return(14); } Here, compiler returns 14 as 14.00 Because the return-type is 'float'</pre>
<pre>void test() { return 10; } If return-type is 'void' then we can't return a value. It shows syntax error.</pre>	<pre>int test() { return; //observe no value } Here compiler shows an error/warning. Because return-type is int, but we are not returning any value;</pre>
<pre>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.</pre>	<pre>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.</pre>

2) about main() function

One may wonder: if main() is also a function, then who calls it, or from where is it called?

Actually, the main() function is called by the operating system (OS). Indirectly, it is initiated by us when we run the program—for example, by pressing F9 in an IDE, typing the .exe file name in the OS shell, or using some other method.

In some compilers, the main() function must return a value. This return value is passed back to the OS, which may then take certain actions, such as closing streams, files, and other resources allocated for the program.

The main() function can be written in any of the following ways:

<pre>void main() { --- --- --- }</pre> <p>Turbo C and other compilers allow this syntax.</p>	<pre>int main() { --- if(condition) return 1; --- --- return 0; // here '0' indicates normal termination }</pre>	<pre>int main() { --- if(condition) return 1; // equal to exit(1) --- --- return 0; //equal to exit(0); }</pre>
--	--	---

The instruction return(0); in the main() function indicates the normal termination of the program. In other words, the program has successfully executed with 0 errors. This statement also signals the operating system to close all files, I/O streams, and other resources before the program ends.

Values like return(1);, return(2);, etc. indicate abnormal termination of the program. In such cases, the operating system records the error code in log files for future reference (similar to how history is stored in web browsers).

If we forget to write return(0); at the end of the main() function, the compiler automatically inserts return 0; before the last closing brace. This ensures normal termination of the program.

But in case of other non-void return functions, if the return is missed, then compiler returns a garbage value.

4) Types of value exchanges between functions

- 1) function with arguments and return value
- 2) function with arguments and no return value
- 3) function with no arguments but returns a value
- 4) function with no arguments and no return value

1) function with arguments and return value: Most calculation-based functions in C need arguments (inputs) and return a value (output). Example functions include, fact() , power(), sqrt().

<pre>int main() { int k; k=fact(4); printf("factorial is %d", k); }</pre>	<pre>int fact(int n) { int f=1; while(n>1) f=f*n--; return f; }</pre>
---	--

2) Function with arguments and no return value: This type of function accepts arguments but does not return any value. Instead, it simply performs a task, such as printing output to the screen.

<pre>int main() { printTable(7); }</pre>	<pre>void printTable(int n) { int i; for(i=1; i<11; i++) printf("\n %d * %d = %d", n,i,n*i); return; }</pre>
--	---

3) Function with no arguments but returns a value: This type of function does not take any arguments but returns a value. Although less common, such functions are frequently used to provide information or results from the system. Examples include: rand() → generates a random number, getTime() → returns current time , getDate() → returns current date, and getGPS() → returns GPS coordinates.

<pre>void main() { int m1,m2; m1=scanMarks(); m2=scanMarks(); if(m1>50 && m2>50) printf("passed"); else printf("failed"); }</pre>	<pre>int scanMarks() { int m; printf("enter marks:"); scanf("%d", &m); return m; }</pre>
--	--

4) Function with no arguments and no return value: This type of function does not take any arguments and does not return any value. It is typically used for fixed tasks, such as clearing the screen or printing common messages

<pre>int main() { showAddress(); }</pre>	<pre>void showAddress() { printf("C-Family Computers"); printf("Vijayawada"); printf("AP,India"); printf("pin:520010"); }</pre>
--	---

5) Rules and regulations applied to functions

- 1) The number of arguments and parameters must match.
- 2) The types of arguments and parameters must be compatible. For example, if an argument is of type int, the parameter can be int, long int, float, etc.
- 3) In C, all functions are independent and defined globally, so any function can be called anywhere in the program.
- 4) A function can return only one value (or none) using the return statement. To return multiple values, pointers are required.
- 5) If the return type is not specified, the compiler assumes the default type as int. However, in some compilers it may be treated as void.
- 6) Some people think that parameters are a special kind of variables that cannot be used other than for receiving arguments. In reality, they are like any other normal variable in the program.
- 7) Passing arguments and returning values makes a function independent, reusable, easy to write, and easier to understand. Always try to divide code into independent blocks of functions.

6) Advantages of functions

Reusability of code: A well-developed function can be reused in many programs by many people, just like the built-in functions printf() and scanf().

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. In practice, programmers often build applications using existing code rather than starting from scratch. For example, functions like pow(), sqrt(), printf(), and scanf() are pre-existing functions that are used repeatedly in almost 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 the function call statements (function names), one can understand what the function does. Functions are often written by different people for different purposes. Generally, a programmer does not need to know how another person's function is implemented. For example, we call sqrt() without knowing its internal working. Thus, when a program is made up of a collection of functions, it becomes easier to understand the overall logic. Simply by looking at the function calls in the program, one can quickly grasp the entire logic.

Easy to modify: if one function is modified, that does not affect the other functions since they are independent.

Portability: In computer science, C has become a vital language, and its usage has extended to almost all fields in the real world. Today, C software is available in most computing environments. If a function is developed on one operating system, it can be easily adapted to other operating systems with little or no modification. Such functions are called portable and generic.

outline of pointers

Before exploring pointers in depth, we will first see how they work in programming.

In programming, every variable has its own memory address, and this address is assigned by the compiler.

For example:

<code>int K = 67 ;</code>	K	→ variable name
	67	→ variable value
	2000	→ variable address (assumed address)

in C, the symbol ‘&’ is called the address or reference operator, it gives the address of a variable, For example

```
printf("%d", K);      // output: 67
printf("%d", &K);     // output: 2000 (address of K)
```

The expression ‘&K’ gives the memory address of ‘K’, where the variable is resided in memory.(here, 2000 is an assumed address). Here the expression ‘&k’ is pronounced as: “**address-of-k**”.

Another operator, ‘*’ is used in the pointers concept. The symbol ‘*’ is called the de-reference or value operator. This gives the value at a given address. Aactually, the symbol * is used in **three ways** in programming:

1) multiplication operator(*) : use to multiply two values (2*3 → 6)

2) de-referencing operator(*) : used to get the value from a given address, for example

`*&k → *2000 → 67`, so the operator(*) gives value at 2000 address

3) pointer declaration operator(*) : used to declare a variable as a pointer.

for example: `int *P;` // here ‘*’ indicates ‘P’ is a pointer variable (not a normal variable)

this declaration creates ‘P’ as a pointer variable. It can store the address of another variable.

so pointer variable is a variable used to store the address of another variable.

Demo program

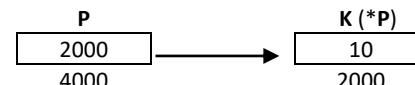
```
int main()
{
    int K = 10;
    int *P;      // here '*' is used to declare 'P' as a pointer variable
    P = &K;      // assigns the address of 'K' to 'P'
    printf("\n output1 = %d", *P ); // here *P accesses K's memory; since *p → *&K → *2000 → 10
```

`*P = 20;` // here also ‘*P’ accesses K’s memory, so this equal to: k=20

`printf("\n output2 = %d", *P);`

`printf("\n output3 = %d", K);`

}



Output: output1 = 10

output2 = 20

output3 = 20

The instruction: `int *P;` declares **P** as a **pointer variable**. Here the symbol * indicates that **P** is not a normal variable but a pointer variable.

The statement `printf("%d ", *P);` prints the **K** value, since `*P` is **K**, that is `*P → *2000 → 10`

If the reference (&) and the dereference (*) appear side by side, they cancel each other. `*P → *_&K → K`

Finally, the expression `*P` accesses the memory of **K**. So, `*P` is nothing but **K**.

Let use come to know point by point

Pointers

Pointers play an important role in the C language; the powerful features of pointers have made C a vital language in the world of computers. In some other languages, such as Pascal or BASIC, pointers are not as programmer-friendly as they are in C. In C, pointers are simple to use, allowing programs to be written efficiently and compactly. They give tremendous power to the programmer. Experienced programmers often feel that, without pointers, no application could be implemented as efficiently as expected. However, pointers can also be dangerous. For example, a pointer containing an invalid address can cause a program to crash.

In programming, it is **not** always possible to access data directly using a variable name. An alternative is to access data indirectly through its address, which is achieved using pointers. Pointers provide quick and dynamic access to arrays. In fact, array values are implicitly managed through this indirect access technique, because we cannot assign a separate name to each element in the array. Instead, all values are accessed through a single pointer. Pointers are especially useful for manipulating arrays, strings, lists, tables, files, and more. They also provide strong support for implementing dynamic data structures such as array-list, linked-list, stacks, queues, sets, trees, and graphs.

Definition of pointer

A pointer is a special type of variable used to store the address of a memory location, so that the memory location can be accessed indirectly through the pointer from any point in the program. In this way, pointers provide indirect memory access.

Memory is a collection of bits, where each group of eight bits constitutes one byte. Every byte has a unique location number called its **address**. This location number (address) is simply a serial number in the byte sequence. For example, the first byte has address 1, the second byte has address 2, and so on. To store this address, we need a special kind of variable called a **pointer variable**.

Since the address of a memory location is just a serial number in the byte sequence, a normal integer variable could technically store it. However, indirect access becomes difficult, because the compiler cannot determine whether the variable is holding an address or a value. To work with pointers, we use **two unary operators**.

- Reference or address operator (**&**)
- De-reference or value operator (*****)

The '**&**' and '******' operators work together for referencing and de-referencing.

Reference operator (**&**)

This referencing operator is also called the address operator, gives the address of the memory location where a variable resides. Add the symbol ampersand (**&**) before a variable name to get its address. For example

<pre>int k=10;</pre>	K ← variable name <div style="border: 1px solid black; padding: 2px; display: inline-block;">10</div> ← value 2000 ← assumed address
----------------------	--

syntax to get address is : &variable-name
for example : &k → 2000

```
printf(" k value is %d ", k );        // output: K value is 10
printf(" k address is %u ", &k );     // output: K address is 2000
```

- ⊕ The address of a variable is not always a fixed location such as 2000, as shown above; it is assigned randomly by the computer. Here, we assumed that the address of K is 2000.
- ⊕ 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.
- ⊕ We know that the variable ‘K’ occupies two bytes in memory. Suppose, it is stored in memory locations 2000 & 2001. In this case, only the first byte address (2000) is considered to be the address of ‘K’ by the compiler. Since the compiler is aware that ‘K’ occupies two bytes, if the first byte address is 2000, then the second byte address will obviously be 2001. Therefore, we always take the ***first byte address as the address of a variable*** regardless of the variable’s size.
- ⊕ Remember that, address is an **unsigned-int** type value. (positive value)

De-Reference operator (*)

You may misunderstand this operator with the multiplication operator, since the same symbol * is used for both. However, there is no confusion, compiler can easily distinguish between the two because they perform completely different actions.

The **dereferencing operator (*)** is used to get the value stored at a given address.

syntax to get a value from address is: * addressable_expression
for example : * 2000 → 10

```
printf("\n the value of K = %d ", *&K ); // output = 10
```

```
*&K → *2000 → 10
*&K → K → 10
```

If the reference (&) and the dereference (*) appear side by side, they cancel each other. $*\&K \rightarrow K$

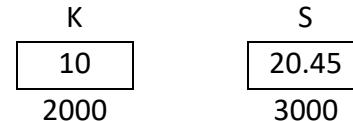
The action of the dereference operator is the exact opposite of the reference operator.

The reference operator (&) gives the address of a variable, whereas the dereference operator (*) gives the value of the variable at that address.

The operator(*) is sometimes informally called the “value operator” because it gives the value at a given address. However, it is also called the ‘indirection operator,’ since it changes the direction of access control from one location to another (from the pointe-variable P to the value-variable K).

```
int main()
{
    int K = 10;
    float S=20.45;

    printf("\n K Address is = %d", &k ); → 2000
    printf("\n S Address is = %d", &S ); → 3000
    printf("\n K Value = %d", *&K ); → 10
    printf("\n S Value = %f", *&S ); → 20.45
}
```



Data type of address

If a variable is of type **int**, then its address is of type **int***. Similarly, if a variable is of type **float**, then its address is of type **float***. In this way, we can easily identify the address type of a variable. Let us see an example:

```
int X=10; float Y=20.45; char Z='A';
```

X	Y	Z
10 →int	20.45 →float	'A' →char
2000 →int*	3000 →float*	4000 →char*

Address types are derived types. By adding the star (*) symbol to any existing data type, it becomes an address type. For example:

- The data type **int*** → is derived from → **int**
- The data type **int**** → is derived from → **int***
- int***** → is derived from → **int****

In this way, pointers can be extended any number of times (theoretically, infinitely).

Similarly, we can define pointer types for all other data types as well. Notice that 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 (we already discussed above)
- 3. To declare variable as a pointer**

Now let us see how to declare a pointer variable:

Declaring a pointer is just like declaring a normal variable, except that the star (*) symbol is prefixed to the variable name. The star (*) symbol differentiates a normal variable from a pointer variable.

Syntax to declare pointer variable is: **data-type *pointer-name1, *pointer-name2, ... ;**

For example: int *p; // this is integer pointer, holds the address of an int variable
 float *q; // this is float pointer, holds the address of a float variable

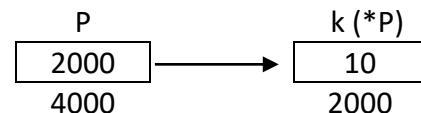
Here, p and q are pointer variables, capable of holding the addresses of int and float variables, respectively. In other words, p is called a pointer to int, and q is called a pointer to float. An integer pointer is used to access an integer value indirectly, while a float pointer is used to access a float value indirectly.

Note: An address is an integer-type value. To store such an address, a pointer requires 2 bytes of memory. Therefore, all types of pointers occupy 2 bytes each. Here, both p and q occupy 2 bytes. (We will understand this more clearly later.)

Demo 1

```
int main()
{ int k = 10;
  int *P; // here '*' is used to declare 'P' as a pointer variable
  P = &k; // assigns the address of k to p

  printf("\n output1 = %d", *P); // here *P accesses k's memory; *P is nothing but K
  *P = 20; // here also '*' is used to access K's memory, this is K=20
  printf("\n output2 = %d", *P);
  printf("\n output3 = %d", K);
}
```



Output: output1 = 10
output2 = 20
output3 = 20

The purpose of the instruction `int *P;` is to declare p as a **pointer variable**. Here, the * symbol specifies that P is a pointer rather than a normal variable.

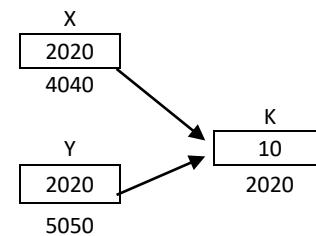
The expression `*P` used in a `printf()` statement accesses the memory of K, i.e., it retrieves the value stored at the address of K.

When P is assigned the value `&K`, we say "**the pointer P is pointing to K**". This relationship is often illustrated with a diagram like shown above, where P holds the address of K and `*P` gives the value of K.

Demo 2

This demo program tells how two pointers can point to same location.

```
int 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 point to the same variable (the same memory location). If the value is changed through one pointer, the other pointers will also reflect the changed value.

Here both pointers x and y are pointing to k. Hence k's memory can be accessed in three ways: direct using k, indirectly using `*x`, and indirectly using `*y`; All the expressions `*x`, `*y`, and `k` access the same location(k).

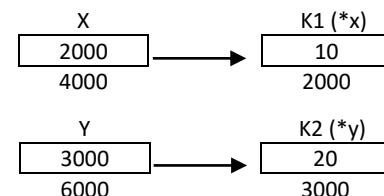
Demo 3

This demo program swaps two values using pointers.

```
int 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

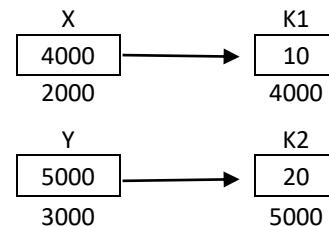
Demo program for swapping two pointers and it is just like swapping two integer values.

```
int 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 );
}
```

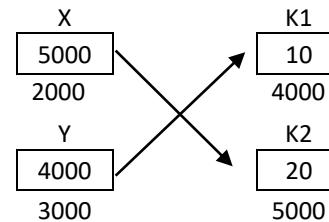
Output:

```
output1 = 20 10
output2 = 10 20
```

Before swapping



After swapping

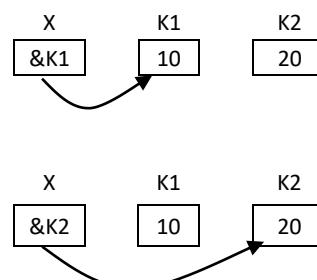


Initially, x and y hold the addresses of k1 and k2, respectively. After swapping, x points to k2 and y points to k1. In this way, we can change the pointing address of a pointer while the program is running.

Demo 5

Expect the output of following program

```
int main()
{
    int k1=10 , k2=20 , *x ;
    x=&k1;
    *x=30;
    x=&k2;
    *x=40;
    printf("\n %d %d %d", *x, k1, k2);
}
```

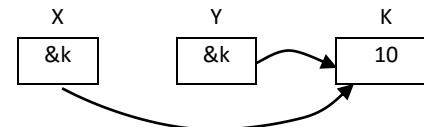


Demo 6: expect output

```
int main()
{
    int k1=10, k2=20, *x, *y ;
    x=&k1; y=&k2;
    *x = *x + *y;
    *y = *x + *y;
    printf("\n %d %d %d %d", k1, k2, *x, *y );
}
```

Demo 7: expect output

```
int main()
{
    int k=10, *x, *y;
    x=&k;
    y=x;
    *y=55;  *x=66;
    printf("\n %d %d %d", *x, *y, k);
}
```



Demo 8: expect output

```
int main()
{
    int k1=10, k2=20 , *x , *y;
    x=&k1;   y=&k2;
    printf("\n enter any two values ");
    scanf("%d%d", x, y ); // this call can be taken as: scanf("%d%d", &k1, &k2);
                           // passing x,y values to scanf(), which are nothing but &k1, &k2;
    printf("\n %d + %d = %d", k1, k2, *x+*y);
}
```

----- keyboard -----
Let input is : 50 80 ↵

Demo 9

```
int main()
{
    int K1=10 , K2=20, *X, *Y;
    X=&k1;   Y=&k2;
    K1++;   K2++;
    printf("\n %d %d %d %d", k1, k2, *X, *Y );
}
```

Demo 10

```
int main()
{
    int K=100 , *X, ;
    X=&K;
    (*X)++; // this is different from *X++
    printf("\n %d %d %d %d", K, *X );
}
```

The instruction `K = *X++;` is interpreted by the compiler as two separate operations: `K = *X;` followed by `X++;`. However, if our intention is to increment the value of K through the pointer X, we must use parentheses: `(*X)++;`. We will discuss more about this concept later.

Types of pointers (address types)

Some people get confused about the difference between a pointer and an address. In reality, the relationship is similar to that of a variable and its value. A pointer variable holds an address, just as an ordinary variable holds a value. For this reason, the terms pointer and address are sometimes used interchangeably in conversation. This is to note that some people use these words interchangeably.

Earlier, we discussed addresses and their types. Now, let us look at address or pointer in more detail.

Pointers are **derived types** because they can be created from any existing data type. By adding the star (*) symbol to a data type, it becomes a pointer type. For example

```
int*    is derived from int
int**   is derived from int*
int***  is derived from int**
```

In this way we can extend infinitely. Similarly all other pointer types can be defined as well.

For example:

```
int *p; // here 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*, respectively.

If there is any data type named XYZ, then XYZ* will represent a pointer to that type.

While declaring a pointer, we should be careful with the placement of the * operator. For example:

- ⊕ In the first version of C, the syntax to declare a pointer was:

```
int* X,Y, Z; // here X, Y, Z are all of type int*
```

- ⊕ In the next version, the syntax was modified as:

```
int *X, *Y, *Z; // the * is written before each variable name
```

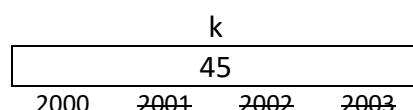
- ⊕ Now if you declare carelessly like: int* X (or) int * X;

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 compiler takes as: int *X, Y;

How much memory occupied by a pointer?

- As we know, an address is like an integer number. Hence, in early C compilers (on 16-bit systems), 2 bytes were enough for pointer to store an address. In modern Windows/Unix-based C compilers, however, pointers usually occupy 4 bytes (on 32-bit systems). In this book, we have taken 2 byte for pointers.
- A variable may occupy multiple bytes, but only the first byte's address is considered the address of the variable. Therefore, to store the address, 2 bytes of memory are sufficient for a pointer (in a 16-bit system). Let, long int k=45; Here K takes 4 bytes memory but compiler consider only first byte address(2000).



- In other words, regardless of the type or size of a variable, only the first byte's address is assigned to the respective pointer. That is why, in older 16-bit systems, 2 bytes were enough for any type of pointer, whereas in new systems 4 is enough for pointer.

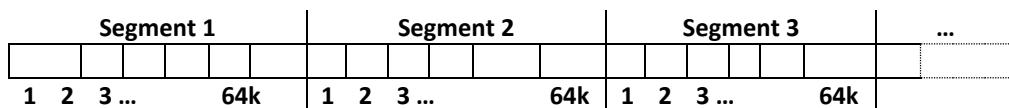
```

int 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



Suppose the variable k is allocated at the 2000th offset of the 3rd segment. Then, the full address of k would be represented as 3:2000. However, in most programs, this address is printed as a single number, such as 32000.

K	← variable name
10	← value
3 : 2000	← allocated in 2000 th offset of 3 rd segment

Conclusion: based on compiler and operating system, the pointer takes 2/4 byte.

Difference between one pointer to another pointer

Even though all pointers occupy the same size of 2 bytes to store an address, they can differ in two ways.

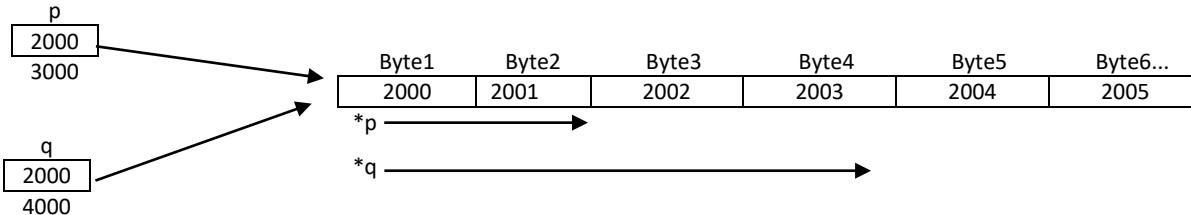
- 1) While accessing the value being pointed to
- 2) While incrementing the pointer (for example, when accessing array elements through a pointer).

```

int *p;
float *q;

```

Let us say p and q have both been assigned the same address, 2000. Now, let us see how they access the memory they are pointing to



The expression `*p` accesses only the first 2 bytes of the memory as int-type, like above picture(2000+2001)

The expression `*q` accesses only the first 4 bytes of the pointed memory as a float type (2000 to 2003).

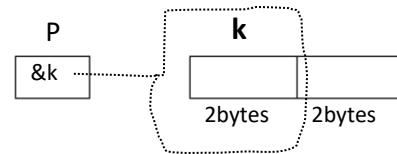
Since p is an `int*` pointer, it accesses only the first 2 bytes of memory as an int. Hence, `int* → points to → int`
 Since q is a `float*` pointer, it accesses only the first 4 bytes of memory as a float. Hence `float* → points to → float`
 In this way, a pointer accesses the memory according to its **type**.

`sizeof(p) → 2 byte whereas sizeof(*p) → 2 byte`
`sizeof(q) → 2 byte whereas sizeof(*q) → 4 byte`

Precautions with pointers

A variable and its pointer must be of the same type. If they don't match, the pointer may access more or fewer bytes than the value it points to, 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.

```
int main()
{
    int *p;
    long int k=10;
    p=&k;
    printf("\n output = %ld", *p);
}
```



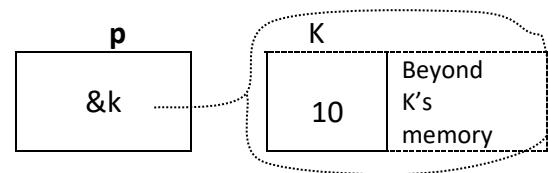
Did you observe? Here, **p** is of type **int***, but **k** is of type **long-int**. Even though **k** is a long-int, its address can still be assigned to the **int*** pointer **p**. (because address is **int-value**) At compile time, this generates a **warning** but not an **error**. However, when we access ***p**, it retrieves only the **first 2 bytes** of **k** (out of its 4 bytes), because **p** is an **int*** pointer.

note: **pointers blindly access memory bytes based on pointer-type and not on pointed-value-type.**

Example 2:

```
int 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 pointed to by the pointer, first **2bytes** belong to **k** and the remaining **2 bytes** belong to unknown memory beyond **k**. This is considered illegal access, often referred to as a **memory leak error** or **pointer leak error**. Such programs should not be executed, as they may crash (terminate abnormally in the middle of execution).

Example 3:

```
int main()
{
    int *p;
    *p=20; // assigns 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 memory of **K**. But 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, while debugging a program, we need to check the address of a variable to verify whether the pointer is pointing correctly or not. In this case, we need to print address of variable in the program.

To print an address, we can use the format specifiers **%u** or **%p**:

%u → prints the address as an *unsigned integer*

%p → prints the address in *hexadecimal format*

```
int main()
{
    int k=10, *P;
    P=&k;
    printf("\n &k=%u , p = %u ", &k , p ); // &k = 2000 , p = 2000
    printf("\n &k=%p , p = %p ", &k , P ); // &k = 0x7D0 , p = 0x7D0
}
```

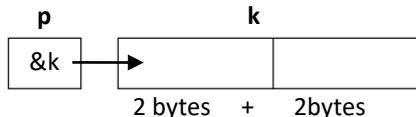
Pointer conversion

In C programming, sometimes we need to access one type of data through a pointer of another type, which requires pointer conversion. In such cases, the pointer must be type-cast according to the type of value it points to. The following example demonstrates how a long-int value can be accessed through an **int*** pointer.

```
int 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
```



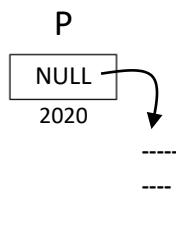
Converting data-type '**p**' from **int*** to **long int***

NULL pointer value

If a pointer contains the value **NULL**, it means it is currently not pointing to any memory location.

The symbol **NULL** is defined in the **stdio.h** file with the value zero. At compile time, every occurrence of **NULL** is replaced by 0 in the machine code (not in the source code)

```
#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;
p = &k;
```

Above pointer assignment can be converted into pointer initialization as:

```
int K=10;
int *P=&k; → here &k is assigned to p not *p.
```

Beginners wrongly identify as , the **&k** is assigned to ***p** (because of its appearance).

Actually, the instruction **int *p = &k** involves two actions.

first box is: **int *p;** and second box is **p = &k;**

We know that the declaration **int *P;** specifies that **P** is a pointer variable of type **int***. However, beginners often misunderstand and think that the pointer's name is ***P**. In fact, the pointer's name is simply **P**; Here the asterisk ***** indicates that **P** is a pointer, not part of its name.

For example, consider the person Mr.Ravi. The person's name is Ravi, not Mr.Ravi. The prefix '**Mr.**' only indicates that he is a male, not part of his actual name.

void* pointer (generic pointer)

It is also called a generic pointer, or a pointer of no particular type. It can hold the address of any data type, but it cannot be dereferenced without explicit type-casting, because the compiler cannot determine the type of data currently the pointer is pointing to. The following example demonstrates how a void* pointer can be used in programming.

```
int 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 an **int*** pointer, it blindly accesses the pointed memory as an **int**. Similarly, if **p** is a **float***, it blindly accesses the pointed memory as a **float**. However, a **void*** pointer cannot determine how many bytes should be accessed in the pointed memory, because it can hold the address of any data type, and the compiler does not know which type it currently points to. As a result, the expression ***p** will cause an error. Therefore, type-casting is required before dereferencing a **void*** pointer.

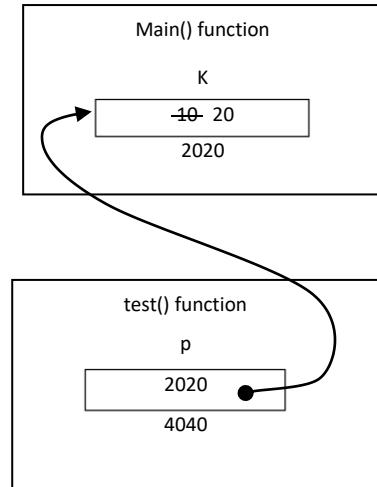
Passing address to function

Passing an **address** to a function is similar to passing a **value** to a function. The only difference is that the receiving parameter must be a **pointer of the same type**. 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); // function proto-type
int 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
```

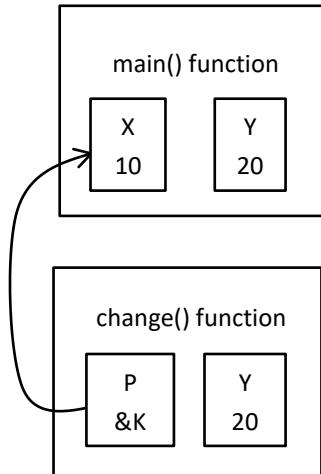


- When the test() function is called in main(), the '**&k**' is passed and stored in the parameter **p**, thereby using **p**, we can read/write/modify the value of **k** from the test() function.
- For example, the expression ***p=20** assigns the 20 to **k**.
- this calling mechanism is known as **call-by-reference**

Example2: Passing address to function

```
#include<stdio.h>
void change( int *p , int q ); // function proto-type
int main()
{ int X=10, Y=20;
  change( &X , Y );
  printf("\n output is: %d %d", X, Y );
}
void change( int *p , int q ) // int *p=&X , int q=Y;
{ (*p)++; // Increments the value of X by 1
  q++; // Increments the local copy of q ( not Y )
}
```

output is: 11 , 20



when above function is called, the address of X and value of Y are passed to the function.

Here P stores the address of X and q stores the value 20 of Y.

(*P)++ increments the value at the address pointed to by P, which is X. So, X becomes 11
q++ only increments the q value by 1 and not the Y value in main() . so the original Y in main() remains 20

As we know, the return statement allows a function to return only a single value. Since most of the time we need to return just one value, the syntax of return was designed that way. However, if we want to return multiple values from a function, pointers are used. Here, we pass the addresses of variables and update their values through pointers. This method is called **call-by-reference**. Let us look at the following examples

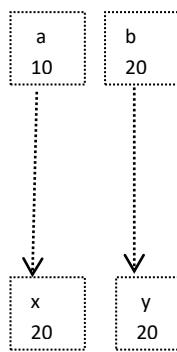
Call by value vs Call by reference

In the **call-by-value** method, the values of arguments are passed and stored into parameters. That is, a copy of the argument's data is created in the parameters of the called function. Now, the arguments are called the **original copy**, and the parameters are called the **duplicate copy**. So, any changes made in the parameters (**duplicate**) do not affect the arguments (**original**). For example, functions like `pow()`, `sqrt()`, `printf()`, etc., follow the call-by-value method.

In the **call-by-reference** method, the addresses of arguments are passed and stored in the corresponding pointer parameters. Using these parameters, we can read, write, or modify the values of the original arguments. For example, functions like `swap()`, `incrementDate()`, etc., follow this method. Let us see an example

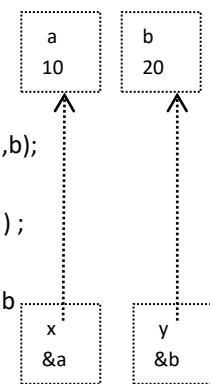
Call by value does not work

```
int 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)
Any changes made to (x, y) that does not affect the values
of (a, b)
```



Call by reference works

```
int 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).
```



The above `swap()` function indirectly returns the swapped values to the `main()` function through pointers.

So using call-by-reference mechanism, we can return multiple values indirectly through pointers.

Finding total and average of 3 number using function.

The following function takes three values and two addresses as arguments. The addresses are used to return the total and average.

```
int main()
{   int a, b, c, total ;
    float average ;
    printf("Enter three numbers: ");
    scanf("%d %d %d", &a, &b, &c);
    findTotalAverage( a, b, c, &total, &average);
    printf("total is %d , average is %f ", total , average );
}
void findTotalAverage(int a, int b, int c, int *p, float *q)
{   *p=a+b+c;           // calculating total and returning
    *q=*total/3.0;       // calculating average and returning
}
```

Finding area and perimeter of circle using function

Following function takes **radius** as argument and returns **area & perimeter** of a circle.

```
int main()
{
    float radius, area, circum;
    scanf("%f", &radius);
    find( radius, &area, &circum );
    printf(" area = %f , circumference = %f", area, circum );
}

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.
}
```

Here, the **find()** function takes the radius as an input argument and calculates the area and circumference. These two calculated values are indirectly assigned to the **main()** function's variables **area** and **circum** through the pointers **p** and **q**.

Finding sum & product of 1+2+3...+N, 1*2*3...*N using single function.

This example explains how the **find()** function returns the sum and product of 1 to N to the **main()** function. This function returns these values indirectly through pointers. Here **find()** function takes the argument **N** along with the addresses of the variables that will receive the returned values.

```
int 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;
    }
    *x=s;   *y=p;      // sum=s , product=p
}
```

Scanning 3 values and printing using functions.

```
int main()
{
    int a, b, c;
    scanThree( &a, &b, &c ); // call-by-reference
    printThree( a, b, c ); // call-by-value
}

void scanThree( int *p, int *q, int *r)
{
    printf("enter 3 values :");
    scanf("%d%d%d", p, q, r); // → scanf("%d%d%d", &a, &b, &c);
}

void printThree( int p, int q, int r)
{
    printf(" %d %d %d ", p, q, r );
}
```

Finding big & small of three values using function

The function `findBigSmall()`, which takes 3 values, after finding big and small it returns the big & small of them.

```
#include <stdio.h>
int main()
{
    int a = 15, b = 29, c = 7, big, small;
    findBigSmall(a, b, c, &big, &small);
    printf("Biggest is: %d\n", big);
    printf("Smallest is: %d\n", small);
    return 0;
}
void findBigSmall(int a, int b, int c, int *big, int *small)
{
    int min, max;
    min = max = a; // Initialize both with first value

    // Find the biggest
    if (max<b) max = b;
    if (max<c) max = c;

    // Find the smallest
    if (min>b) min = b;
    if (min>c) min = c;
    *big = max; // returning biggest
    *small = min; // returning smallest
}
```

Sorting 3 values (a, b, c)

Here, three functions are used such as `scanThree()`, `sort()` and `swap()`, used to scan, and sort three values (a, b, c) into ascending order. The `sort()` fn sorts 3 values into order with the help of `swap()` function for swapping.

logic for sorting a, b, c is : if(a>b) swap a,b; if(a>c) swap a,c; if(b>c) swap b,c; complete the code.

ip: 23 12 2	ip: 10 2 5
op: 2 12 23	op: 2 5 10

```
int main()
{
    int a, b, c;
    scanThree(----);
    sortThree( ---- );
    printf("\n output is: %d %d %d", a, b, c);
}

void scanThree( -----)
{
    -----
}

void sortThree( ----- )
{
    -----
}

void swap(int *p, int *q)
{
    -----
}
```

Arrays VS pointers

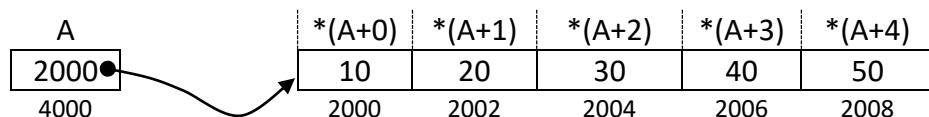
Arrays are implicitly managed as pointers by the compiler. All array expressions are converted into pointer expressions by the compiler. For example, the expression $A[i]$ is converted into $*(A + i)$. We can also interchangeably use array expressions as pointer expressions and vice versa. Here, the array name works as a constant pointer variable, which holds the address of the first element. This is shown below:



The array expression $A[i]$ can be written in pointer form as $*(A + i)$.

So, the expression $A[i]$ is called array notation, whereas $*(A + i)$ is called pointer notation.

We can use these expressions interchangeably, as shown below



The expression $A[0] \rightarrow *(A+0) \rightarrow *A$

$\&A[0] \rightarrow \&*(A+0) \rightarrow (A+0) \rightarrow A \rightarrow 2000$ (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 in pointer style

$A[0] \rightarrow *(A+0) \rightarrow *A \rightarrow *2000 \rightarrow 10$
 $A[1] \rightarrow *(A+1) \rightarrow *(2000+1) \rightarrow *(2002) \rightarrow 20$ (integer-address increments by 2)
 $A[2] \rightarrow *(A+2) \rightarrow *(2000+2) \rightarrow *(2004) \rightarrow 30$
 $A[i] \rightarrow *(A+i) \rightarrow *(2000+i)$

Note: compiler implicitly takes $A[i]$ as $*(A + i * \text{sizeof(int)}) \rightarrow *(A + i * 2)$

This is because, to move from one value to next value in the array, the pointer should be incremented by 2.
If it is a float array, then the pointer is incremented by 4. So pointer increments based on its type.

Point to remember: the expression $\&A[0] \rightarrow \&*(A+0) \rightarrow (A+0) \rightarrow A \rightarrow$ it gives base address of array (2000)

```
printf("%d %d %d ", A, A+1, A+2); → 2000 2002 2004
printf("%d %d %d ", *(A+0), *(A+1), *(A+2)); → 10 20 30
printf("%d %d %d ", A[0], A[1], A[2]); → 10 20 30
```

Let us see one program, printing array values

```
int main()
{
    int A[5] = {10, 20, 30, 40, 50};
    int i;
    for(i=0; i<5; i++)
        printf("%d ", A[i]); // 10 20 30 40 50
    for(i=0; i<5; i++)
        printf("%d ", *(A+i)); // 10 20 30 40 50
}
```

Making a pointer to an array

The C language also provides the facility to access arrays using our own pointers.

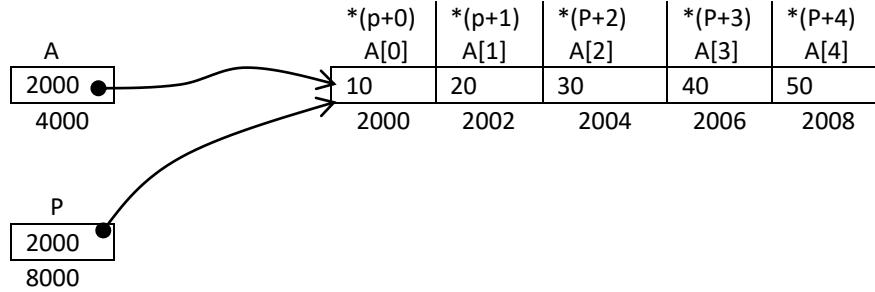
Here, a single pointer **cannot** point to all elements at once; instead, it can point to only one element at a time.

By incrementing the pointer, or by adding an index value to it, the remaining elements can be accessed.

Let us see an 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 points to an array.



The above pointer expression $*(p+0)$, $*(p+1)$, $*(p+2)$... can be written as $p[0]$, $p[1]$, $p[2]$, $p[3]$

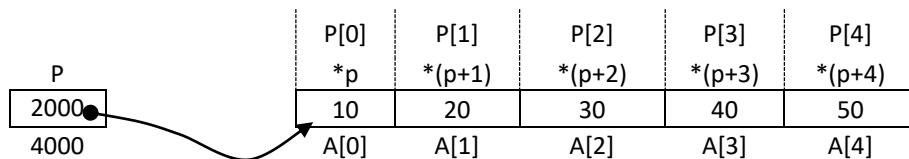
Since $*(p+i)$ can be written in array style as $p[i]$

the expressions $*(p+0) \rightarrow p[0] \rightarrow *p$ accesses the A[0] // now p[0] and A[0] accesses same location
 $*(p+0) \rightarrow *(2000+0) \rightarrow *2000 \rightarrow 10$

$*(p+1) \rightarrow p[1] \rightarrow$ accesses the A[1]
 $*(p+1) \rightarrow *(2000+1) \rightarrow *2002 \rightarrow 20$

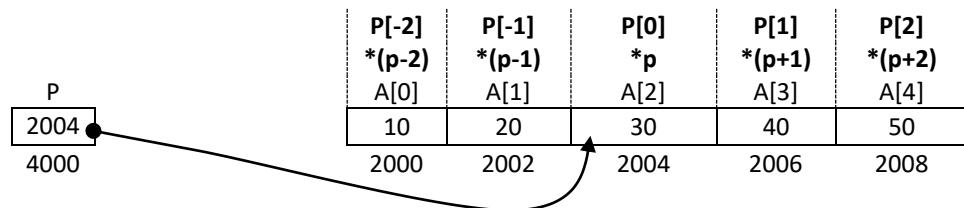
$*(p+2) \rightarrow p[2] \rightarrow$ accesses the A[2]
 $*(p+2) \rightarrow *(2000+2) \rightarrow *2004 \rightarrow 30$

The above figure can be imagined with the pointer "p" as



if the pointer 'p' is set to A[2] then the expressions are as follows

```
p = &A[2];
```



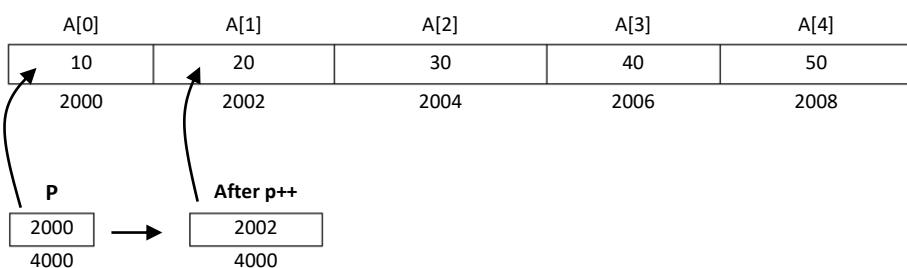
Here p[0] accesses the A[2], p[1] accesses the A[3], p[-1] accesses the A[1], p[-2] accesses the A[0]

Demo program for printing array elements using pointer

```
int 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 P by 2
    }
}
```

The above two loops prints the same output.

In the 2nd for loop, the pointer **p** advances to next element after every **p++** operation, observe following figure



Example for printing address of each element in the array

```
int 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 );
}
```

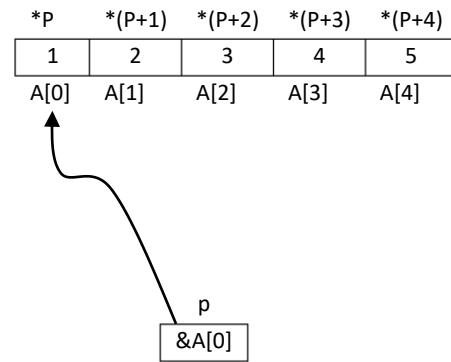
Output: 2000 2002 2004 2006 2008 // address of each element in the array

Passing one dimensional array to function

In a single function call, we cannot pass all the elements of an array to a function at once, as there is no such provision in C. The solution is to pass the base address of the array and then access the elements through a pointer. This method is both simple and fast.

```
int 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 the `display()` function is called, the argument `&A[0]` is passed and stored in the pointer `p`. This is the same as passing an integer address to a function. The parameter `p` can be declared in two ways: `int *p` or `int p[]` (both declarations are equivalent).

The first declaration, `int *p`, may cause some confusion about whether the pointer `p` is pointing to a single-integer or an array-of-integers, it cannot be determined immediately. For this reason, C provides the second form. However, when seeing `int p[]`, one might wrongly assume it represents an empty array, but in reality, it is the same as `int *p`.

This 2nd declaration `int p[]` gives a clearer idea that `p` is receiving the address of an array instead of a single integer address.

When developing large applications, there may be many functions scattered across different files. The called function may not be located immediately below the calling function, and sometimes it exists in another file altogether. For this reason, the array-style pointer declaration `int p[]` is recommended. It clearly indicates that `p` is a pointer to an array rather than a pointer to a single value.

Let us have one more example, filling values to array at called-function.

```
int 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 19
}
void fill( int p[] or int *p )
{
    p[0]=10;    p[1]=16;    p[2]=19;
}
```

following function takes array-base-address and count of elements, returns the sum of all.

```
int findSum ( int p[], int n )
{
    int i, sum=0;
    for( i=0; i<n; i++ )
        sum=sum+a[i];
    return( sum );
}
int 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

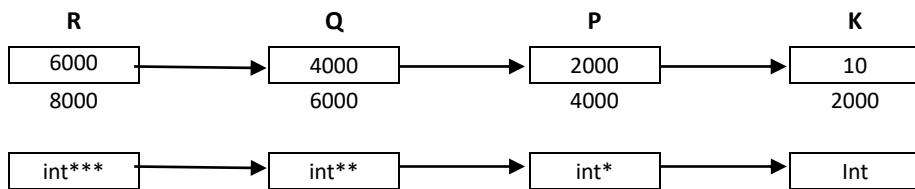
output: sum is 24

The ability to access a calling function's array elements in the called function using the array's base address provides a convenient way to move multiple data items back and forth between functions. This feature witnesses the excellence of pointers in C.

Pointer to pointer (multiple indirections)

A pointer that points to another pointer variable is known as a “pointer to pointer.” For example, to store the address of an int variable, an int* pointer is required; similarly, to store the address of an int* variable, an int** pointer is required. In this way, we can extend the concept to any level of indirection. However, in practice, we usually use only single or double pointers. A pointer to a pointer is mainly used in working with multi-dimensional arrays.

```
int main()
{
    int K=10;
    int *P;      // single indirection pointer
    int **Q;     // double indirection pointer
    int ***R;    // triple indirection pointer
    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 the ‘P’, the ‘***R’ accesses the ‘K’ value.

*R → Q	**R → *Q → P	***R → **Q → *P → K
--------	--------------	---------------------

11) Swapping two values using single and double pointer. Let us see following example

```
#include<stdio.h>
int main()
{
    int a=25, b=35;
    swap( &a , &b );
    printf("%d %d " , a , b);
}

void swap( int *p, int *q )
{
    exchange( &p , &q );
}

void exchange(int **x, int **y)
{
    int t;
    t=**x;
    **x=**y;
    **y=t;
}
```

```
#include<stdio.h>
int main()
{
    int a=25, b=35;
    swap( &a , &b );
    printf("%d %d " , a , b);
}

void swap( int *p, int *q )
{
    exchange( p , q );
}

void exchange(int *x, int *y)
{
    int t;
    t=*x;
    *x=*y;
    *y=t;
}
```

Array of pointers

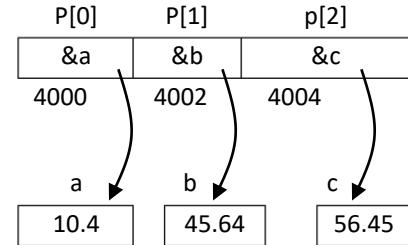
In computer science, arrays of pointers are widely used in programming. I recommend practicing more with this concept, as it is very useful in many applications.

An array that holds a collection of addresses is known as an array of pointers. For example, instead of declaring multiple pointers like `int *p1, *p2, *p3;`, we can declare an array of pointers as `int *p[3];`.

Using this concept, we can handle 2D array data such as matrices and tables in a more flexible and compact way.

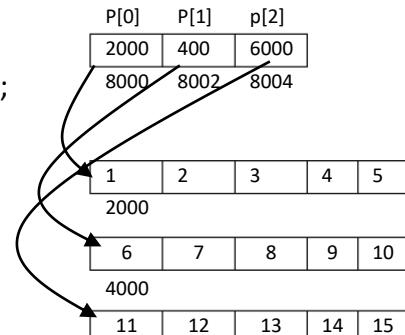
Demo program for array of pointers

```
int 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

```
Int 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) or p[i][j] );
        printf("\n");
    }
}
output: 1 2 3 4 5
       6 7 8 9 10
       11 12 13 14 15 16
```



Advantages of pointers

1. **Call by Reference**
 - We can return multiple values from a function.
 - Passing an array to a function is possible only through call-by-reference.
 - If the data is large (like strings, structures, or files), call-by-reference saves both time and space.
2. **Dynamic Memory Allocation**
 - Dynamic memory allocation is possible only with the help of pointers.
3. **Hardware and OS Interaction**
 - Direct interaction with hardware and the operating system is possible.
4. **Dynamic Data Structures**
 - Pointers are the backbone for implementing dynamic data structures.
5. **Efficient Code**
 - Pointers make the code more flexible, compact, and faster in execution.
6. **Handling Large Data**
 - Through pointers, it is easier to handle large collections of data.

More about pointers

1. Some common ways of pointers 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 the 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 is points to

An `int*` pointer points to an `int` memory location.
A `float*` pointer points to a `float` memory location.
Similarly, all other pointers point to memory locations of their respective data types.

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 pointer/address

- Adding or subtracting integral value to an address is allowed like `p+1`, `p-1`, `p+10` ...etc
- Multiplying or dividing integral value to an 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]` gives 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

A pointer can not only point to data, but it can also point to code. That is, a pointer can be made to point to a function, allowing us to make function calls through the pointer. In C, every function is loaded into a separate memory location as it appears in the program. This memory address serves as the entry point of the function, i.e., the address of its first instruction. Using this address, control can be shifted from the caller function to the called function.

In general programming, this approach may seem unnecessary because in C all functions are global and can be called directly from anywhere in the program. However, function pointers become useful when we need to pass the addresses of different functions at different times to another function. This makes function calls more flexible and supports a higher level of abstraction.

Syntax to declare a pointer to a 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 points to a function of type 'int fn-name(int, int)'
that is, the fn which takes two integer arguments and returns an integer value.
- ② void (*p)() → here the pointer 'p' can points to a function of type 'void fn-name()'
that is, the fn which takes no arguments and returns no value.
- ③ float (*p) (float,float,float) → it can points to a function of type 'float fn-name(float, float, float);'

To get the address of a function, use the function name without parenthesis or arguments that gives starting address of a function. The following program gives a demo.

```
Int main()
{
    void (*p)(); // pointer to a function, which takes no arguments and returns no value.
    p=test; // assigning test() function address to 'p'
    p(); // calling test() function through 'p'
    (*p)();
}
void test()
{
    printf("\n Hello World");
}
```

Example for calling a function based on even/odd input

```
int 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 on 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()	Void x (void (*p)())	int main()
{	{	{
printf("hello");	p(); // calling y() fn through 'p'	x(y); // passing y() fn address to x() fn
}	}	}

Sorting and Searching

The word sorting refers to arranging data either in ascending or descending order. The collection of data may consist of integers, real numbers, strings, etc. In computer science, sorting is a major process in everyday tasks. Therefore, several efficient techniques have been invented, so programmers can relax instead of worrying about how to code and sort elements efficiently.

You may have one doubt: what is the use of sorting? If the data is not in sorted order, searching takes much more time. But if the data is sorted, searching becomes easier. In this case, the search operation takes less than $\log_2 N$ time; otherwise, it may take up to N time. We have three elementary techniques available.

1. Bubble sort.
2. Selection sort.
3. Insertion sort.

We also have several advanced alternative sorting techniques available (which are not covered in this book).

Bubble sort

In bubble sort, in array, each adjacent pair of elements is compared, and if they are not in the order, they are swapped. That is, if $a[i] > a[i+1]$, then the elements are exchanged. In this way, bubble sort gradually arranges the array into sorted order. The name bubble sort comes from the idea that during comparisons, larger elements “bubble up” to the end of the array, just like bubbles rising to the surface of water.

The algorithm is as follows:

Step 1: In the first pass of the loop, if $a[i] > a[i+1]$, swapping takes place. This comparison and swapping continues from the first element up to the last element (for $i = 0$ to $N-1$). After this pass, the largest element is moved to the last position, so it is considered sorted.

Step 2: In the second pass of the loop, only the first $N-1$ elements are considered (since the last element is already sorted). Again, comparisons and swaps are performed as in Step 1. At the end of this pass, the second-largest element will be in the $(N-1)^{th}$ position, and it is considered sorted. In this way all elements are sorted.

Let us consider 5 values: 90 , 10, 70, 30, 20

Observe elements how they are swapped after every pass of inner loop

Before 1 st iteration	90	10	70	30	20
After 1 st iteration	10	90	70	30	20
Before 2 nd iteration	10	90	70	30	20
After 2 nd iteration	10	70	90	30	20
Before 3 rd iteration	10	70	90	30	20
After 3 rd iteration	10	70	30	90	20
Before 4 th iteration	10	70	30	90	20
After 4 th iteration	10	70	30	20	90

After 4 cycles of inner loop, the biggest **90** have moved last to its right place and sorted
Repeat these iterations for all elements. 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];           // swapping elements
                a[j]=a[j+1];
                a[j+1]=temp;
            }
        }
    }
}

```

The main() function to test the bubble sort

```

int 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

Time complexity analysis of Bubble sort

The total number of operations required for bubble sort is: **number of comparisons + number of swaps**

1) Total number of comparisons in the loop (if($a[j] > a[j+1]$))

To sort 1st big element, it takes N-1 comparisons (to move to the last position),

To sort 2nd big element, it takes N-2 comparisons (to move to previous of last position),

Total number of comparisons are: $N-1 + N-2 + N-3 + \dots + 3+2+1 \rightarrow (N-1*N)/2 \rightarrow$ proportional to $N^2/2$.

2) Total number of swappings in the loop:

The total number of comparisons is $N^2/2$, Obviously, the total number of swaps cannot exceed $N^2/2$.

However, a swap occurs only when the comparison $a[j] > a[j+1]$ evaluates to true. But every time, the condition may not be true, so if we take average then it will be $N^2/4$.

Since each swap takes 3 operations, the total cost of swaps will be $3*N^2/4$.

Therefore, the total time $T(N) = \text{number of comparisons} + \text{number of swaps}$. $T(N)=N^2/2 + 3N^2/4 \rightarrow 5N^2/4$

$T(N) = 5N^2/4 ; T(N) = O(N^2)$ approximately

So Bubble sort takes N^2 instructions to be executed. Here N is total number of sorting elements

Bubble sort 2'nd method

Sometimes, the input array may already be almost sorted except for one or two elements. In that 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 inner loop, the total array gets sorted and successive iterations would not be required. Using following algorithm, we can achieve solution for this problem. 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.

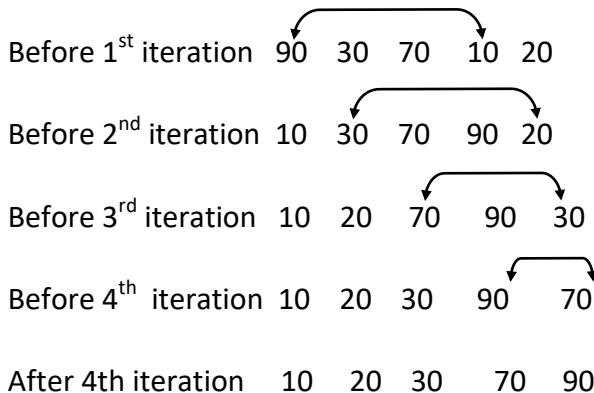
```
void BubbleSort2( int a[], int n )
{
    int flag;
    for(i=n; i>0; i--)
    {
        flag=0;
        for(j=0; j<i; j++)
        {
            if( a[j] > a[j+1] )
            {
                temp=a[j];           // swapping
                a[j]=a[j+1];
                a[j+1]=temp;
                flag=1;
            }
        }
        if(flag==0) break; //if no swapping occurred, then terminate the loop.
    }
}
```

Selection sort

In **selection sort**, the algorithm first finds the smallest element in the array and exchanges it with the element at the first position. Next, it finds the second smallest element and exchanges it with the element at the second position, and so on. 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



```

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] ); // write swap() function yourself
    }
}

```

Note: the main() function code is same as in bubble sort;

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
 $n-1+n-2+n-3+\dots+3+2+1 \rightarrow (n-1)*n/2 \rightarrow$ 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 number 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
 $= n^2/2 + n^2/4 + 3 * n \rightarrow$ 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 choice 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.

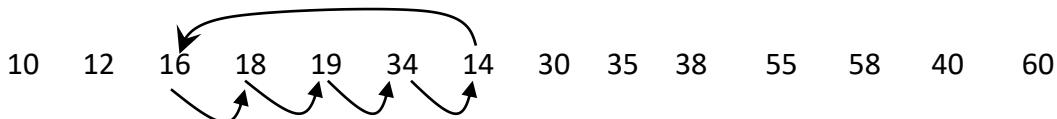
* Record : records is nothing but a collection of fields. for example customer record in a bank: Account holder name

account no	account type	sex	age	address	balance
------------	--------------	-----	-----	---------	---------

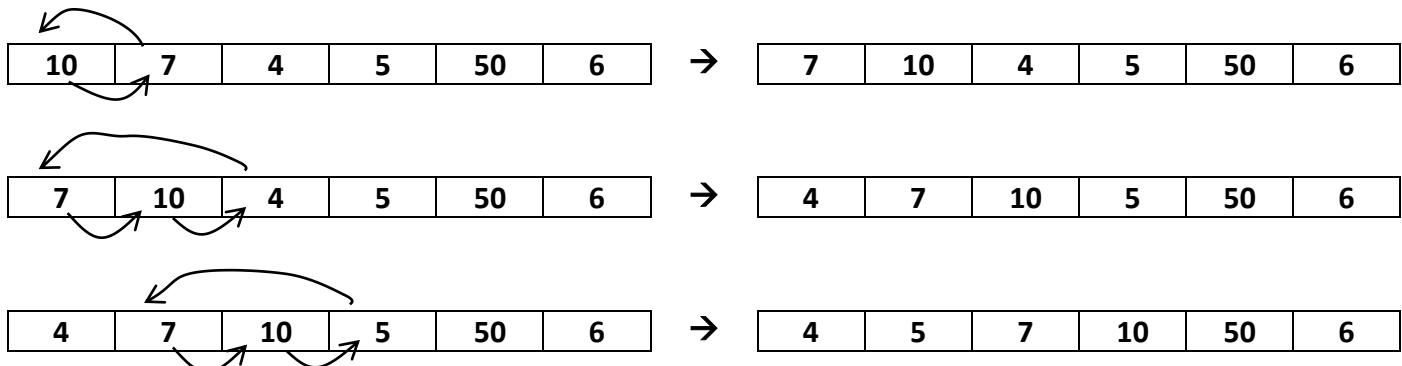
[#]Cost: cost means time, here the time is calculated in terms of cost.

Insertion sort

Let us pickup k^{th} element in the array, compare $a[k]$ with the previous elements one by one, if previous elements are bigger than $a[k]$ then shift them one position to the right. This creates a gap at the correct position where $a[k]$ can be inserted. Let us observe how 14 will be sorted (shifted to right position) in the following elements



Actually, the process starts from $a[1]$ as shown in the below picture. First we take $a[1]$ and compare with the previous element $a[0]$, and values are shifted if needed. Next we take $a[2]$ and compare with the previous elements $a[1], a[0]$. In this way insertion sort works.



Algorithm for insertion sort

Let us take the first ' $k-1$ ' elements in the array (a sub-array) and assume they themselves are already in sorted order (sorted by previous iterations of loop). Now take the k^{th} element, compare it with the previous elements one by one, and shift all bigger elements to right-hand side, so that we get a gap where insert the k^{th} element. These inserted positions may not be the final positions, as they might have to be moved again to make room for smaller elements encountered in subsequent iterations.

Step by step procedure to sort elements

Step 1: pick up $a[1]$, compare it with previous element $a[0]$. If $a[0]>a[1]$, then shift $a[0]$ to the right side and insert $a[1]$ in place of $a[0]$.

Step 2: Now, the first two elements are in sorted order. Next, pick up $a[2]$ and compare it with the previous elements. Shift them to right side if they are bigger than $a[2]$. After shifting, we get a gap where insert $a[2]$. (while comparing with previous elements, if any element is smaller, stop the inner loop because subsequent elements will be even smaller.)

Step 3: Repeat this process for all the remaining elements.

```
void insertionSort ( int a[], int n )
{
    int temp, i, j;
    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 big 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
    }
}
```

Time complexity 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 stop in middle) so It uses about $N^2/4$ comparisons and $N^2/4$ data moves on the average.

So total no.of operations is $N^2/4 + N^2/4 \approx N^2/2$.

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 is : algorithms by Robert Sedgwick**)

Searching techniques

Searching is the process of finding required key element in the group of elements.

We have 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

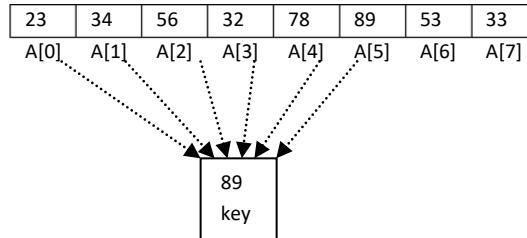
Sequential Search (or Linear Search) means checking elements one after another. In this search, each element of the array is compared with the element we are looking for.

- If the element is found, the search returns the index of that element.
- If the element is not found, it returns -1.
- Note: The array elements may or may not be sorted.

```
int linearSearch( int a[], int n, int key )
```

```
{   int i;
    for(i=0; i<n; i++)
    {   if( a[i]==key )
        return i;
    }
    return -1;
}

int 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 the input file is small or elements are not in sorted order, it takes $O(n)$ time.

If elements are in sorted order, the binary search is the best choice, it takes $<\log_2(n)$ time.

Divide the array into three parts: the left sub-array, the middle element, and the right sub-array.

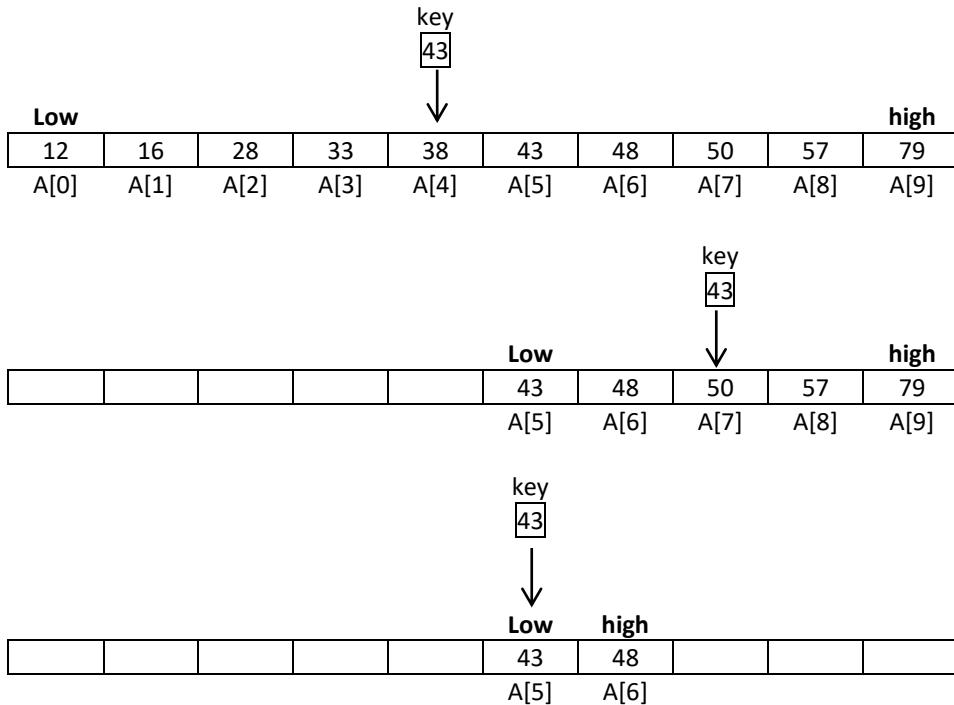
If the middle-element == search key, return the index of the middle element and stop.

If the search-key < middle-element then continue the search in the left sub-array.

If the search-key > middle-element then continue the search in the right sub-array.

Repeat steps 1 to 4 until the sub-array has less than 1 element or the element is found.

The following picture demonstrates searching for 43 using the binary search method:



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
}
```

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 $\rightarrow (2^k)$

At cycle2, the array size is $N/2 \rightarrow (2^{k-1})$ \rightarrow because at cycle2, we search in left/right sub-array.

At cycle3, the array size is $N/4 \rightarrow (2^{k-2})$

After cycle K, the array size is 1 $\rightarrow 2^0$

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

$N \approx 2^k \rightarrow k=\log_2(N)$, so finally binary search takes $\log_2 N$ comparisons in worst-case.

Scope, Accessibility & Storage Classes

Scope means a region, area, block, or boundary that specifies a limited or enclosed space. In C, scope is defined using braces { }. These braces specify the boundary of the scope. A region surrounded by a pair of braces { } in a program is said to have a local scope.

Scope of a Variable: The scope of a variable is the region in a program where the variable is available and accessible. If a variable is declared within a block { }, it becomes local to that block and cannot be accessed outside it. If we try to access it outside the block, the compiler will generate an error.

In C, variables can be declared in any block of a program, but they must be declared at the beginning of the block. Such variables are called local variables, and they cannot be accessed outside the block in which they are declared. Let us see some examples.

Example1: int main()

```

{
    {
        int K;      // here memory space creates for 'K'
        k=100;
    }
    K=200;      // here we get compile time error called "undefined symbol K"
}

```

In the above program, the instruction K=200; shows an error called "**undefined symbol K**", because the variable K is declared inside the inner block. Therefore, it belongs only to that inner block and cannot be accessed or used outside it. The scope of K is limited to the inner block in which it is declared. Such variables are called local variables, and their scope is restricted to the block in which they are defined. Let us see one more example.

Example2: int main()

```

{
    {
        int k;      // here memory space for 'k' creates and scope belongs to this block only
        k=100;
    }
    {
        int k;      // this 'k' is different from above 'k', here one more 'k' gets created in the memory
        k=200;
    }
}

```

Here two copies of 'k' variables will be created, the first 'k' belongs to first inner block and second 'K' belongs to second inner block. Within the same block, two variables cannot be declared with the same name. However, variables with the same name are allowed in different blocks, because each block maintains its own local scope.

Example3: int 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;
int main()
{
    int Y=200;
    show();
}
void show()
{
    printf("%d %d", X, Y);      // error, undefined symbol 'Y'
}
```

Here, the variable Y is declared inside the main() function block, so it is **local to main()** and cannot be accessed inside the show() function. In this way, one function's local variables cannot be accessed directly in another function.

On the other hand, the variable X is declared at the beginning of the 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. Here 'X' is said to be global variable whereas Y is local variable.

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: We have already discussed local scope with examples. Let us now conclude it. Local scope is the region surrounded by a pair of braces {} in a program. Local variables can be declared inside any block of a program, but they must be declared at the beginning of the block. This block may belong to an if–else, while loop, for loop, or a function block. Local scope reduces the variables complexity (confusion)

Global scope is said to be entire program's area. If any variable is declared at the beginning of a program (outside all functions) then it is said to be global variable. Let see one more example,

```
int X=100;           // X is called global variable because it is in global-scope
int main()
{
    int Y=200;       // Y is a local variable, and this belongs to main() fn block

    if( X < Y )
    {
        int Z;       // Z is also a local variable, and belongs to if-block only
        Z=300;
    }

    X=200;
    show( 1000 );
}

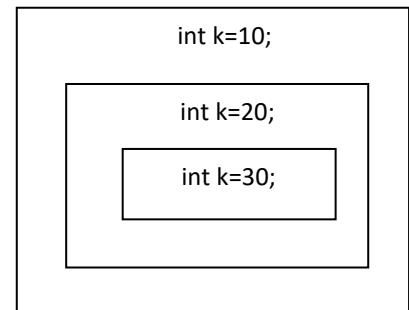
void show( int A )   // here A is also a local variable, belongs to this show() fn block, and also works as a parameter
{
    int B=100;       // here B is also a local variable, and belongs to this show() fn block
    printf( "%d %d %d %d", X, Y, Z, A, B );    //error, here Y, Z are unknown to this 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 or overlapped 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.

example1:

```
int 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
}
```



example2:

```
int X=10;
int main()
{
    int X=20;
    printf( "%d ", X );      // output is 20 (not 10), here global variable X overridden by local variable X
    show();
}
void show()
{
    printf( "%d ", X );      // output is 10 (not 20), here there is no local variable like X , so global X is accessed
}
```

example3:

```
int X=10;
int main()
{
    show();
    printf( "%d ", X );      // output is 10
}
void show()
{
    int X=20;
    printf( "%d ", X );      // output is 20
}
```

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 defined in a program possesses two main characteristics: **data type and storage class**.

Data type tells the memory size and type of values it can hold, 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 either zero or garbage: for temporary variables the default value is garbage, whereas for permanent variables the default value is zero.

* The life span specifies when the variable comes into existence and when it goes out of existence; that is, when the variable's space is created in RAM (life starts) and when that space is 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 variables (automatic)

It is the default storage class for all local variables declared inside a block. That is, if a variable is declared inside a block, its storage class is automatically auto. Hence, the keyword auto is optional for local variables. In fact, 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: The life of an auto variable exists only within a block, as long as the control remains in that block. That is, when the control enters a block, memory space for all its auto variables is created (life starts), and when the control leaves the block, that memory space is destroyed (life ends).

```
int main()
{
    test();           // calling 3 times
    test();
    test();
}
void test()
{
    auto int k=10;   // memory space of k will be created, when the control comes here
    printf(" %d ", k);
    k++;            // no use of incrementing before dying (this is given for demo )
}
                                // memory space of k will be destroyed here, before returning control to main() function
output: 10 10 10
```

The variable 'k' will be created & destroyed 3 times upon calling test() function for 3 times.

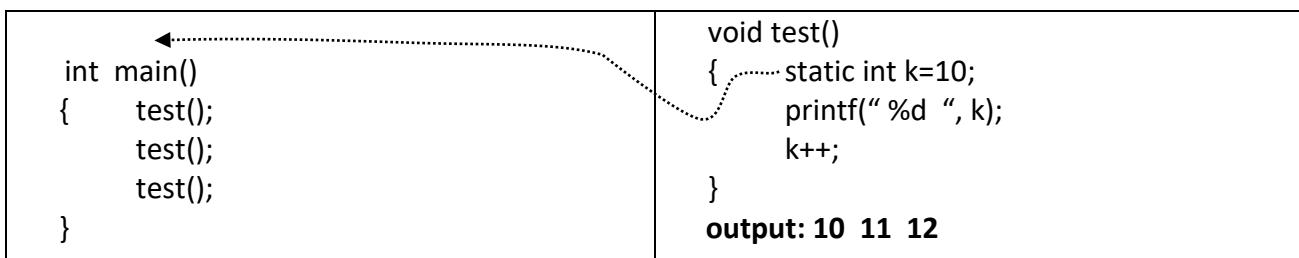
When the control enters into the test() function's body, the 'k' will be created & initialized with 10 and when the control leaves the function, the 'k' will be destroyed. Hence the output is 10 10 10.

It is generally meaningless for k to remain in memory after the function finishes its task. Therefore, all auto variables are destroyed before the function ends. However, in some cases, a variable is required to survive across multiple function calls. In such situations, the static storage class is used.

static variables

This is also a local variable like auto, but its memory space is created before the start of the program (before main() function begins) and it survives until the end of the program. It is used to retain and share values from previous function calls for subsequent calls.

Unlike auto, the static storage class preserves the value of a variable even after the function ends. The variable is not recreated or reinitialized on every function call. Instead, it is created only once and continues to exist until the program terminates. Its scope is local (like auto), but its default initial value is zero.



The instruction **static int k=10** executes only once just before main() fn starts (Compiler shifts this instruction to before main() function in the machine code file, like above shown arrow-line). Here **printf(" %d ", k); k++;** executes 3 times for 3 function calls.

Example2

```

int main()
{
    int i;
    for( i=0; i<3; i++ )
    {
        auto int X=10;
        static int Y=10;
        printf("\n %d  %d", X , Y );
        X++; Y++;
    }
}

```

output: 10 10
10 11
10 12

The variable X (auto) is created and destroyed 3 times, once during each iteration of the loop. Every time it is reinitialized to 10. Hence output for X is: 10 10 10

The variable Y (static) is created **only once** and initialized only once (with 10). Its value is preserved across iterations. Hence, output for Y is: 10 11 12

register variables

Registers are small memory units available inside the processor's ALU. These are inbuilt memory locations used by the CPU for performing operations. Before carrying out any arithmetic operation, input values are moved from RAM to registers, and then the operation takes place. Thus, all arithmetic operations are performed with the help of registers.

Since a processor usually has only 4 or 5 registers, and 1 or 2 are always busy, the remaining unused registers can be assigned to program variables. This provides faster access, as it avoids repeatedly transferring variable values between RAM and registers. Hence, the register storage class is particularly suitable for frequently used variables, such as loop counters.

Only **local integral types** (char, int, long) are allowed as register variables.

Floating-point types, arrays, pointers, and static/global variables **cannot** be declared as register variables.

The keyword register is **not a command**, but only a **request to the compiler**.

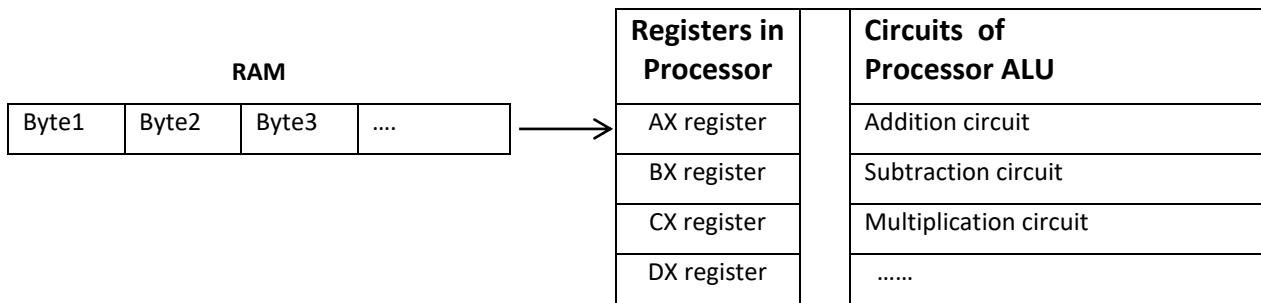
If no registers are available, the variable will simply be allocated in RAM.

Other behaviors such as **scope**, **life**, and **default initial value** are the same as the auto storage class.

Example: register int a, b, c, d, e;

```
register float x, y, z; // error, float types are not allowed
```

In the above declaration, all register variables (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



global variables

We have already discussed global variables while studying scope. Let us now look at them in more detail.

As we know, a function's local variable cannot be accessed in another function. However, sometimes some variables need to be accessible in many functions, in such cases variables are declared in the global scope (outside of all blocks and functions).

Global scope is the region that does not belong to any specific function or block; instead, it belongs to the entire program. Global variables are usually declared at the beginning of the program or in a separate file.

The life span of global variables starts before the main() function begins and continues until the end of the program (same as static). static variables can be accessed only within the block in which it is declared, whereas a global variable can be accessed anywhere in the program.

The word global is not a keyword in C, and therefore it is not required when declaring a global variable.

```
int k; //now the 'k' is global and default value is zero
int 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-span & 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.

extern (proto-type of global variables)

If a program is very large, its code may be distributed across several files, and each file may contain multiple functions. Each file can be compiled independently to produce an object file, and later all such object files are linked together. During this compilation and linking process, some problems may arise. For example, if one file uses a global variable that is declared in another file, the compiler will report an undefined symbol error.

To avoid this type of error, we declare a prototype of the global variable using the 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 the static keyword in the global scope. Usually, file-scope variables are declared at the beginning of a file so that they can be accessed by all functions within that file. So these variables are local to a single file and cannot be accessed from outside that 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.

storage place of variables (stack area and heap area)

The program's memory is divided into code area and data area. The code area contains the program's instructions. The data area contains variables, string constants, and other data.

The data area is further classified into:

Stack – where temporary variables are stored and managed.

Heap – where permanent variables are managed.

The stack and heap are predefined data structures at the operating system 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
```

Preprocessor Directives

Before learning about preprocessor directives, let us see one example. The following program calculates the area and perimeter of a circle:

```
int main()
{
    float area, perimeter, radius;
    -----
    area = 3.14 * radius * radius;
    perimeter = 2 * 3.14 * radius;
    -----
}
```

Now, suppose we are writing a big application where the value 3.14 (π) is used many times, say more than 50 times. After some days, if we need to change this value from 3.14 to 3.145, it will take a lot of effort to modify all occurrences. If we accidentally miss one or two places, it may give **wrong results**.

In such situations, it would be much better if we could define this value in one place, and that change would automatically affect the whole program. For this purpose, preprocessor directive is the alternative, here we use **#define** statement like given below

```
#define PIE 3.14          // do not give semicolon at end
int main()
{
    float area, perimeter, radius;
    -----
    area = PIE *radius*radius;
    perimeter = 2*PIE*radius;
    -----
}
```

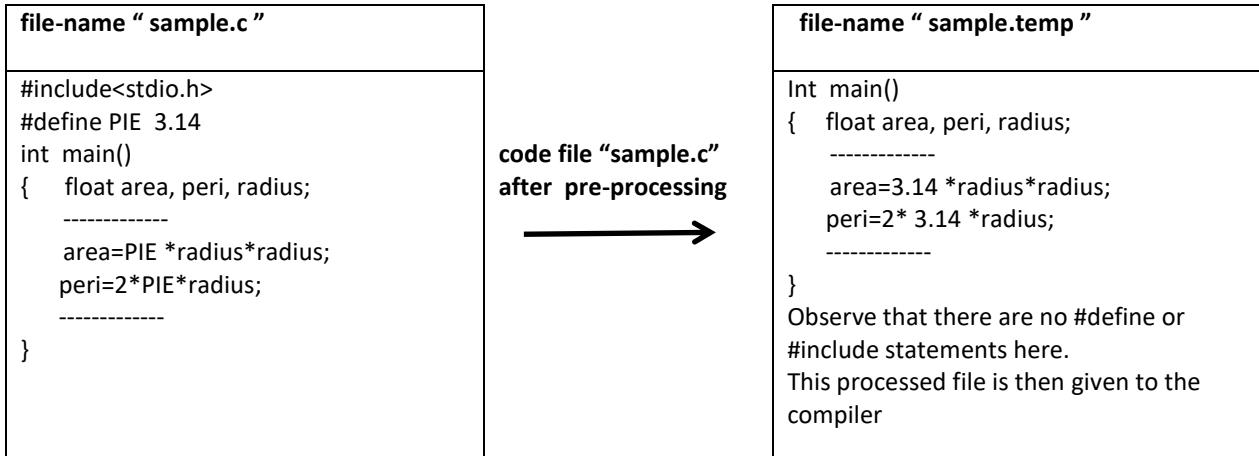
At compile time, the symbol ‘PIE’ is replaced by 3.14 (π) in every place it is used in the program. This replacement happens internally by the compiler and affects the executable file, not in our source code file. If any changes are needed in the future, we simply modify the value in the macro definition and recompile the program. The compiler will again then generate a new ‘exe’ file by replacing old ‘exe’ file. This new exe file updated to new value. In this way, we can easily automate the repeated constants using #define directive.

It is mandatory to symbolize the repeating constants, so that they can be easily remember, manage, and modify later. In this way wherever the pie value 3.14 is required, we can use symbol **PIE**.

What is the preprocessor? The preprocessor, as its name implies, is a program that performs some kind of text manipulations just before compilation begins. It provides facilities such as automating repeated constants, simplifies complex math equations, providing conditional compilation, and combining two or more files, etc. Actually, this is not an **inbuilt** feature of the c-compiler. This is an additional tool kit that helps the programmer.

Note: All preprocessor directives start with the **#(hash)** symbol, and we can use these statements anywhere in the program. Common 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 then given to the compiler (as shown in the picture below). It does not modify our original source file. It creates new temporary file where it replaced PIE with 3.14. Thus compiler does not know about **#define**, **#include** statements.

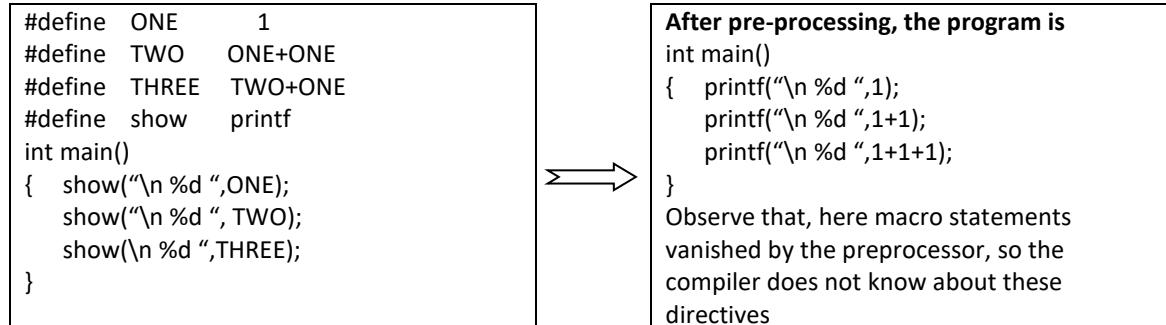


More about macro directive #define

#define is called a macro directive. It defines symbolic constants and expressions.

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 the macro-name in the source file. The replacement-expression can be anything like constant, equation, string, etc. Let us have one more example.



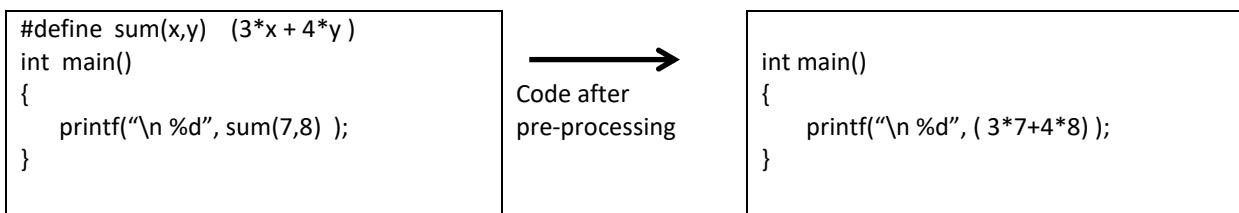
Note: It is good programming practice to write macro names in all capital letters, so that they can be easily identified in a program. In fact, this convention is widely followed in the software industry.

I strongly recommend you follow this rule for clarity and convenience.

Defining macro like function

The #define directive has another powerful feature: it can be used as a simple function. Generally, it is used for simple mathematical equations, complex expressions, or small functions. A macro function can have parameters, and these parameters are substituted during preprocessing.

Syntax is: **#define macro-name(parameters) replacement-expression-with-parameters**



Advantages of macro over function

- **No function call overhead:**

Macros do not make any function calls; instead, the replacement expression is directly pasted wherever the macro is used. Hence, we can avoid the overhead of function calls and returns. As we know, a function call consumes extra space and time for jumping and returning control, as well as for creating and destroying local variables.

For very simple functions (one or two lines of code), this overhead can dominate the actual work. In such cases, it is better to use a macro to save both time and space.

- **No need to specify data types for parameters:**

Macros are not type-dependent, so the same macro can be used with different types of data. For example, a macro like `sum(x, y)` will work with integers, floats, or doubles, such as `sum(4.5, 5.4)` values are floats.

Disadvantages with macros

- If macro replacement-expression is large, 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 pastes the replacement-expression. For example

```
#define sum(x) (x$x)
int main()
{ printf(" output = %d", sum(5)); // here macro pasts as ( 5$5 )
}
```

Here by mistake, the user has given the '\$' instead of '+', the macro blindly 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
int 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 large application into several functions and write them in multiple files. Later, these files are combined using the `#include` directive. This directive merges two or more files into a single file.

It is used to join the contents of one file into another file. There are three syntaxes for `#include`:

`syntax1: #include "file name"`

`syntax2: #include "path + filename"`

`syntax3: #include <file name>`

Syntax 1: Used when the included file is in our current working directory.

Syntax 2: Used when the included file is in some other directory.

Syntax 3: Used when the included file is in the standard C software director

For example:

Here, the file ‘main.c’ contains the main() function code, and the file ‘sub.c’ contains a sub-program.

The sub-file should be included as:

main file “main.c”	sub name “sub.c”	Code after pre-processing	File name “main.temp”
#include “sub.c” int main() { printf("%d", fact(5)); }	int fact(int n) { long int f=1; while(n>1) f=f*n--; return(f); }	After pre-processing, the sub.c file’s code is pasted in main.c as	int fact(int n) { int f=1; while(n>1) f=f*n--; return(f); } int main() { printf("%d", fact(5)); } this file is given to compiler

These two files will be combined into a single file and then given to the compiler (as shown above).

In this way, we can attach as many files as needed in a program.

The included sub-file can contain many functions, but only the functions that are actually called will be part of the executable file.

In this manner, C header files (.h files) contain the prototypes of all functions, and we must include these files to avoid errors during compilation. These header files are used by the compiler to check for syntax errors and ensure that functions are called correctly in the program.

Conditional Compilation

It is possible to compile selected portions of a program’s source code. This process is called conditional compilation and is widely used by commercial software companies that provide and maintain multiple customized versions of the same program.

Nowadays, commercial software is designed to accommodate the needs of different organizations. For example, one bank’s software can be adapted for another bank with only a few modifications. We cannot determine or finalize all the details of an application at the time of development—such as the operating system, hardware, or available services.

Therefore, while developing software, programmers must anticipate all possibilities that may arise in later stages. This requires writing code for all potential scenarios so that sections can be added, removed, or modified according to the customer’s requirements before installation.

In this way, a generic program can be adjusted as needed before delivery with the help of the preprocessor.

Let us see an example.

```
#define COUNTRY 1      // 1-India, 2-US, 3-France, 4-England, ...
#if COUNTRY==1
    char curr[]="Rupees";
#elif COUNTRY==2
    char curr[]="Dollar";
#else
    char curr[]="No currency defined";
#endif

int main()
{   printf("our currency = %s", curr);
}
```

By changing country code in first line, the corresponding currency name is applied to the ‘curr[]’

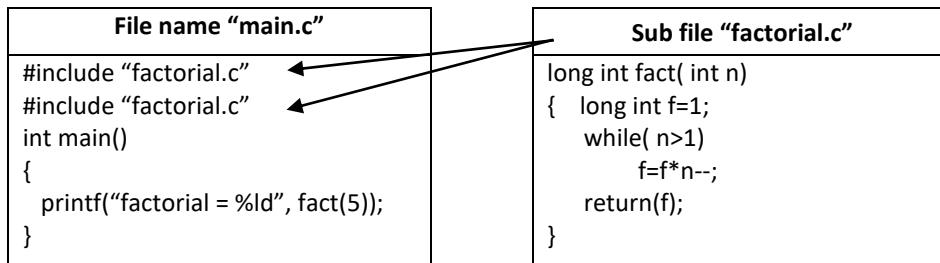
#ifdef, #ifndef and #undef

There is another method of conditional compilation, which is used to check whether a given symbol has already been defined using #define or not. Based on this check, we can include or exclude certain parts of the code during compilation. The Syntax is as follows

```
#ifdef macro-name
    statement1
    statement2
    ...
#endif
```

example1	example2
<pre>#define SIZE 100 #ifndef SIZE #define SIZE 200 #endif int main() { printf("\n the size = %d", SIZE); } Output: the size = 100</pre>	<pre>#ifndef SIZE #define SIZE 200 #endif Int main() { printf("\n the size = %d", SIZE); } Output: the size = 200</pre>

Example 3: Let us see one more practical example



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 ...

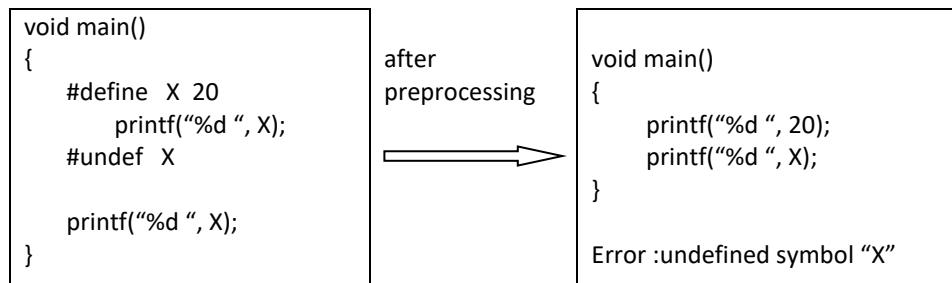
Sub file "factorial.c"
<pre>#ifndef FACT #define FACT 1 long int fact(int n) { long int f=1; while(n>1) f=f*n--; return(f); } #endif</pre>

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



#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

```

#ifndef 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-dimensional or three-dimensional array data structures, especially when working with matrices, tables, strings, or files. In C programming, we can declare and access as many dimensions as needed.

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 the **row-size**, 5 represents the **column-size**.

Then the 2D array is as follows

		Columns				
R	O	x[0][0]	x[0][1]	x[0][2]	x[0][3]	x[0][4]
o	w	x[1][0]	x[1][1]	x[1][2]	x[1][3]	x[1][4]
s		x[2][0]	x[2][1]	x[2][2]	x[2][3]	x[2][4]

Scanning 2x4 matrix data and displaying on the screen

```
int 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 an 1D-array, second one is by enclosing each row with { }. The second method is clear and easy to understand, and 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
```

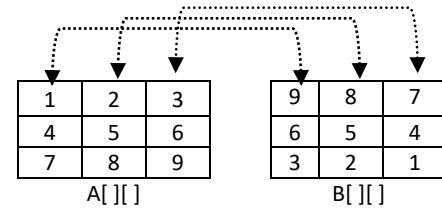
In the last two declarations, the row size is automatically calculates by the compiler. Here, the row-size equal to (number-of-elements)/(column-size). The column size is mandatory while declaration of array.

```
int a[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 size 3x3 and prints the result. Consider the source matrices A and B to be already initialized with some assumed values.

```
int 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");
    }
}
```



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.

```
int 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

```

int 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 means changing values from rows to columns and from columns to rows."

```

#define M 3
#define N 4
int 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 $m1xn2$.

```
int 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 \geq 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 \leftarrow

enter marks of 4 subjects:

60 49 40 50 \leftarrow

10 20 30 40 \leftarrow

70 50 90 80 \leftarrow

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	

```

int 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 %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} };`

1 st row					2 nd row					3 rd 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 2000	2 2002	3 2004	4 2006	5 2008	6 2010	7 2012	8 2014	9 2016	10 2018	11 2020	12 2022	13 2024	14 2026	15 2028

In one dimensional arrays, the expression `x[i]` can be written in pointer form as `*(x+i)`

Similarly, 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

$$x[i][j] \rightarrow *(x+i*column-size+j).$$

$$x[0][0] \rightarrow *(x+0*5+0) \rightarrow *(2000+0) \rightarrow *2000 \rightarrow 1$$

$$x[0][1] \rightarrow *(x+0*5+1) \rightarrow *(2000+1) \rightarrow *2002 \rightarrow 2 \quad // \text{ integer address increments by 2}$$

$$x[1][0] \rightarrow *(x+1*5+0) \rightarrow *(2000+5) \rightarrow *20010 \rightarrow 6$$

$$x[1][1] \rightarrow *(x+1*5+1) \rightarrow *(2000+6) \rightarrow *2012 \rightarrow 7$$

$$x[2][0] \rightarrow *(x+2*5+0) \rightarrow *(2000+10) \rightarrow *2020 \rightarrow 11$$

$$x[2][1] \rightarrow *(x+2*5+1) \rightarrow *(2000+11) \rightarrow *2022 \rightarrow 12$$

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.

$$x[0] \rightarrow (x+0*5) \rightarrow x \rightarrow 2000 \rightarrow \text{gives 1st row address}$$

$$x[1] \rightarrow (x+1*5) \rightarrow x+5 \rightarrow 2010 \rightarrow \text{gives 2nd row address}$$

$$x[2] \rightarrow (x+2*5) \rightarrow x+10 \rightarrow 2020 \rightarrow \text{gives 3rd row address}$$

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

```

int 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.

```

int 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

```

int 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)

```

int 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 necessary to handle single-character data items such as employee's sex (M|F), the marks grade of a student (A|B|C), or pass/fail status (P|F), etc. These characters may be an alphabet, digit, or any other symbol, and they are treated as single-character items in a program.

In a computer's memory, characters are stored in terms of their ASCII values, not in the picture form that we see on the screen. For example, the ASCII value of 'A' is 65, and the ASCII value of 'B' is 66. These ASCII values are stored in binary form, since that is what the computer understands. ASCII values are standard serial numbers assigned to each alphabet, digit, and all symbols in the keyboard. This standard is called **American Standard Code for Information Interchange**, proposed by the international computer association. The term "information interchange" refers to exchanging data between devices or between computers in the form of ASCII values.

Hence, every character is internally represented in binary format, and when it is displayed on the screen, it is converted into its symbolic picture form (in pixel format). Internally, special software continuously performs this graphical conversion task while displaying contents on the monitor. Such software is usually provided by the hardware manufacturers. The following table shows the ASCII codes of some character sets:

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 is enclosed within a pair of single quotes, it is known as a character constant. Unlike integer or float constants, character constants must always be enclosed within single quotes (not double quotes). Remember, the value of a character constant is nothing but its 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 ASCII values of characters lie between 0 and 255, so to store such values, C's developer, Dennis Ritchie, provided a suitable data type called char. The data type char occupies 1 byte of memory, where we can store values in the range 0 to 255.

```
char variable1, variable2, ...variableN;           // this is the syntax to declare character variables
char gender, marksGrade, accountType;           // here gender, marksGrade are char type variables
```

Initialization of char variable:

```
char gender='M';          // Let 'M' male
char gender=77;           // this is also valid but informal
int gender='M';           // this is also valid but informal
char color='Y';           // Let 'Y' is yellow
```

char gender='M' and char gender=77; Both of these declarations are the same, since the ASCII code of 'M' is 77. However, the number 77 is difficult to remember or recognize as an ASCII code, so using the symbolic representation ('M') is always better. The third declaration int gender='M' is also valid but 1byte gets unused in 2bytes of int memory. Because only 1 byte is actually needed to store the ASCII value of a character.

Character library functions for I/O

`scanf()`, `getchar()`, `getch()`, `getche()`
`printf()`, `putchar()`, `putch()`.

We know that `printf()` and `scanf()` are generic i/o functions used to print and scan any type of data. In contrast, other functions like `getchar()`, `getch()`, `putchar()`, `putch()` are specialized to handle only characters data. These are light weight functions used to read/write single characters.

getchar(): This is light-weight alternative to `scanf()` function, it accepts a single character from keyboard.

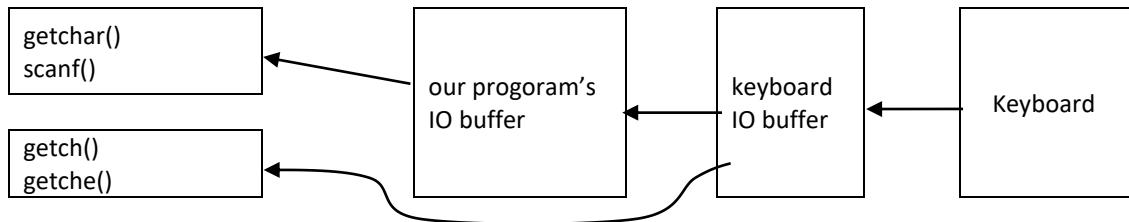
```
char ch;
ch=getchar(); // this is equal to scanf("%c", &ch );
```

getch(): the `scanf()` and `getchar()` do not support reading all keys such as up-arrow, down-arrow, pg-up, pg-dn, home, end, F1, F2, etc. But, the `getch()` can read all keys in the keyboard. It reads values directly from keyboard buffer, not from program's io buffer, as shown below.

The keyboard buffer is a small memory space used to store user input values before sending them to a program. It acts as a mediator between the keyboard and the program. Whatever we type on the keyboard is first placed into this buffer, and later the program reads the input values from it. Thus, the buffer works as an intermediary between the keyboard and the program.

For example, a milk delivery boy places milk packets in our basket at home, and we collect them whenever we are free. Similarly, a keyboard stores input values in a buffer, and the CPU reads from it when it is available. Since the CPU performs many tasks, it may not be immediately ready to read input while the user is entering it. This is why buffer technology exists—to temporarily hold data until the CPU can process it.

In this analogy, the buffer acts like the basket. Let us see the following picture.



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 the Enter key is pressed before reading the input. But `getch()` (and `getche()`) reads the key immediately, without waiting for Enter.

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.

```
int 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'; ch++)      // 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.

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 converts 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");    v
else   printf("\n lower case alphabet");  x

if( isupper( 'a' ) ) printf("\n upper case alphabet");  x
else   printf("\n lower case alphabet");  v

if( isdigit( '9' ) ) printf("\n digit");      v
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.

<pre>int 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); }</pre> <p>always we can't remember the ascii difference, this code is not recommended</p>	<pre>int 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); }</pre> <p>Always we can't remember ascii codes, this is not recommended</p>	<pre>int main() { char ch; printf("enter a alphabet :"); ch=getchar(); if(isupper(ch)) ch=tolower(ch); else ch=toupper(ch); printf("opposite is: %c", ch); }</pre> <p>This is the best choice to understand easily.</p>
---	--	---

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 are 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 produce only ascii-codes, whereas special keys produce two values: 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, here we have to scan again for 2nd code.

```
int main()
{
    char ch1,ch2;
    printf("enter any key in the keyboard :");
    ch1=getch();
    if( ch1 != 0 )          // user entered normal-key
        prnifff("\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 ↵
output: ascii code = 65

input: up-arrow ↵
output: ascii code = 0, scan code = 72
```

Printing following patterns

ABCDEF	void main()	A	void main()
ABCDE	{ int i , j;	AB	{ int i , j;
ABCD	for(i=0; i<6; i++)	ABC	for(i=0; i<6; i++)
ABC	{ for(j=0; j<6-i; j++)	ABCD	{ for(j=0; j<=i; j++)
AB	printf("%c", 'A'+j);	ABCDE	printf("%c", 'A'+j);
A	printf("\n");	ABCDEF	printf("\n");
	}		}
	}		}

About fflush(stdin)

```
int main()
{
    char ch1,ch2;
    printf("enter character1:");
    ch1=getchar(); or scanf("%c", &ch1);
    printf("enter character2:");
    ch2=getchar(); or scanf("%c", &ch2);
    ----
}
```

input: enter character1: A ↵ (A + enter-key)

unfortunately or unknowingly, we entered two input characters, first one is 'A', and second is 'enter-key'.

In case of characters input, the 'enter-key' is also considered to be input value. Here first character 'A' will be scanned by ch1=getchar(); whereas second character 'enter-key' will be scanned by ch2=getchar().

Here we need to clear the KB buffer to avoid such unwanted character scanning. If you use fflush(stdin) before reading 2nd character then it clears the 'enter-key' from the buffer so that next new character can be scanned.

Let us see following program.

```
int main()
{
    char ch1,ch2;
    printf("enter character1:");
    ch1=getchar(); or scanf("%c", &ch1);
    printf("enter character2:");
    fflush(stdin); // clears the enter-key from keyboard buffer
    ch2=getchar(); or scanf("%c", &ch2);
    ----
}
```

enter character1: A ↵

enter character2: B ↵

stdin → represents keyboard buffer, stdout→represents monitor buffer, stdprn→represents printer buffer.
fflush(stdin) clears the total keyboard buffer if anything there.

Example2:

```
printf("enter character 1&2:");
ch1=getchar();
ch2=getchar();
```

input: enter character 1&2: AB↵ (A + B + enter-key) it works fine, here: ch1='A' and ch2='B'

Example3:

```
printf("enter character 1&2:");
ch1=getchar();
ch2=getchar();
```

input: enter character1&2: **A space B↵** (A + space + B + enter-key) then here: ch1='A' and ch2=space
Remember space is also a character, ASCII value of space is 32, so 32 gets inserted in ch2.

Points to remember: before scanning a character or string as second input value, then you need clear the unwanted characters like enter-key or space or tab from keyboard buffer.

Strings

A string is a collection of characters arranged sequentially, one after the other in memory. Strings are also called arrays of characters and are used to represent names of persons, items, countries, cities, and sometimes messages. Generally, any non-computable data is handled as strings. That is, arithmetical operations such as addition, subtraction, multiplication, or division cannot be applied on string data.

In C, there is no direct data type for handling strings; instead, we access them as normal arrays of characters. However, C provides some basic facilities to handle strings easily, such as representation of string constants with double quotes, initialization of strings, and several standard library functions to manipulate strings. Example strings are "India", "China", "C-Family", "123", "A123", etc.

Strings are stored like arrays of characters in the form of ASCII values, and these ASCII values are further represented in binary, which the computer understands. For example, the string "INDIA" is stored in 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 memory: 5 bytes for "INDIA" + 1 byte for the null character ('\0'). The value '\0' is called the null character, and its ascii value is zero. It is used to indicate the end of the string. The compiler automatically appends the null character at the end of string constants.

What is a string constant?

Like integer, float and 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. It is also called string literal.

For example: "India", "C-family", "A", "", "9440-030405", "A123", "1234"

Do not confuse 1234 with the "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, string lengths differ from one another, so there should be an indication to identify the end of a string. For this reason, the null character ('\0') is inserted at the end of every string. Almost all C library functions are designed with this null character as a string terminator. String library functions automatically add this null character at the end of a string. For example: scanf(), gets(), strcpy(), strcat(), etc.

Sometimes, programmers need to insert the null character explicitly while handling strings character by character. Some programmers use 0 in place of the null character because the value of the null character is zero ('\0' → 0). The ascii values of all chars lie in b/w 1-255, so the exceptional value 0 is used as the null-character-value to terminate a string. However, in programs, it is better to write '\0' instead of 0(zero) to represent in symbolically.

Note: Strings are nothing but character arrays. Therefore, any operation applicable to arrays can also be applied to strings. For example: printing, scanning, passing a string to function, accessing a string through pointer, etc.

String as a variable

we know that a variable is the name of a memory space used to store and access a value through the variable name. Similarly, to store and access the characters of a string, one requires a character array.

For example: char arr[20]; → this array 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 strings in different ways. This is shown in the examples below.

```
char a[9] = "C-Family"; // here null character is automatically appended at end string by the compiler
char a[] = "C-Family"; // the size of array is calculated 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' }; // if size is 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 ASCII values of "c-family"
```

The last declaration with ASCII values is somewhat confusing 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 the 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() provide a simple syntax and are specialized for string data. On the other hand, scanf() and printf() are generic I/O functions that support all data types such as int, float, long, and string.

For strings, the format specifier %s is used to read or write string data with scanf() and printf().

① scanf("%s", array-address) : This reads a string character by character from the keyboard until a newline or a blank space is encountered, whichever comes first. If input is: jack and jill↙, then scanf() accepts only "jack".

② scanf("%[^\\n]", array-address) : This reads a string **char by char** until a **newline** is encountered, that is, it accepts the **entire string including spaces**.

For example, if the input is: jack and jill↙ then this function accepts the entire string "jack and jill".

The gets() and this scanf("%[^\\n]", array-address) work in a similar way.

These input functions automatically append a null character ('\0') at the end of the string. For example

Scanning a string using: scanf()

```
char a[20];
printf("enter a string :");
scanf("%s", a); or scanf("%s", &a[0] );
printf("output = %s", a);

input: Jack and Jill // this scanf() treats as 3 strings
output: Jack
```

Scanning a string using: gets()

```
char a[20];
printf("enter a string :");
gets(a); or scanf("%[^\\n]", a );
printf("output = %s", a);

input: Jack and Jill
output: Jack and Jill ( scanned entire string )
```

The puts() and printf() works in similar way, both prints the string char by char until null character is found.

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
int main()
{   char a[100];
    printf("enter string :");
    scanf("%[^\\n]", a);           // a → &a[0]
    printf("\n the string you entered is : %s", a);
}
```

The scanf() function takes the array base address as an argument and fills the scanned string into this array. At the end of the string, scanf() automatically inserts the null character ('\0') . This is done by the function, not by the compiler.

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

int 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.

```
int main()
{
    char a[100], ch; int i, lower=0, upper=0, digit=0, space=0, other=0;
    printf("\n enter a string :"); gets(a);
    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 counts 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 IEGS

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.

```
int 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

```
int 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 both in lower and upper 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

```
int 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 the keyboard and stores into &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 the null character is found.

Finding Length of a string

input: INDIA input: All fruits are not apples

output: length=5. output: length=25

```
int main()
{
    char a[100], count=0;
    printf("\n enter any string :");
    gets(a);
    for(i=0; a[i]!='\0'; i++)      // counting all characters until the null is found
        count++;
    printf("\n length = %d", count);
}
```

Here, the loop repeats until the **null character ('\0')** is reached, and at the same time, the characters are counted using the variable count to find the string length. Generally, the **null character is not considered** as part of the length.

Finding sum of digits in a given string

This program accepts a string from the keyboard. The string can contain digits, and the program finds the sum of these digits. For example, if the input string is: "ABC567DEF9PQ12"

output is: 5+6+7+9+1+2 → 30

```
int 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, ... ; so here 48 is subtracted to convert ascii value to numeric value

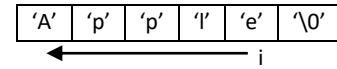
Printing a string in reverse order

ip: Apple

op: elppA

logic: first for-loop moves the 'i' to end of string, later prints char by char from last to first in reverse order

```
int 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 the number of words. It is assumed that there may be more than one space between two words, but there are no spaces around other symbols like dots, commas, etc.

input: A L L A P P L E S A R E R E D \0

```
int 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 is found
        {
            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, which contains more than one space between words. The inner while loop is used to skip extra spaces between words. When the index pointer i reaches the first space, the word-count (counter) is incremented, 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)

```
int 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 means, next word is started
            a[i]=toupper( a[i] );
        else
            a[i]=tolower( a[i] );   // if a[i] is not first letter of word then convert it 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.

```
int 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 two keys 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 are string constants stored in memory?

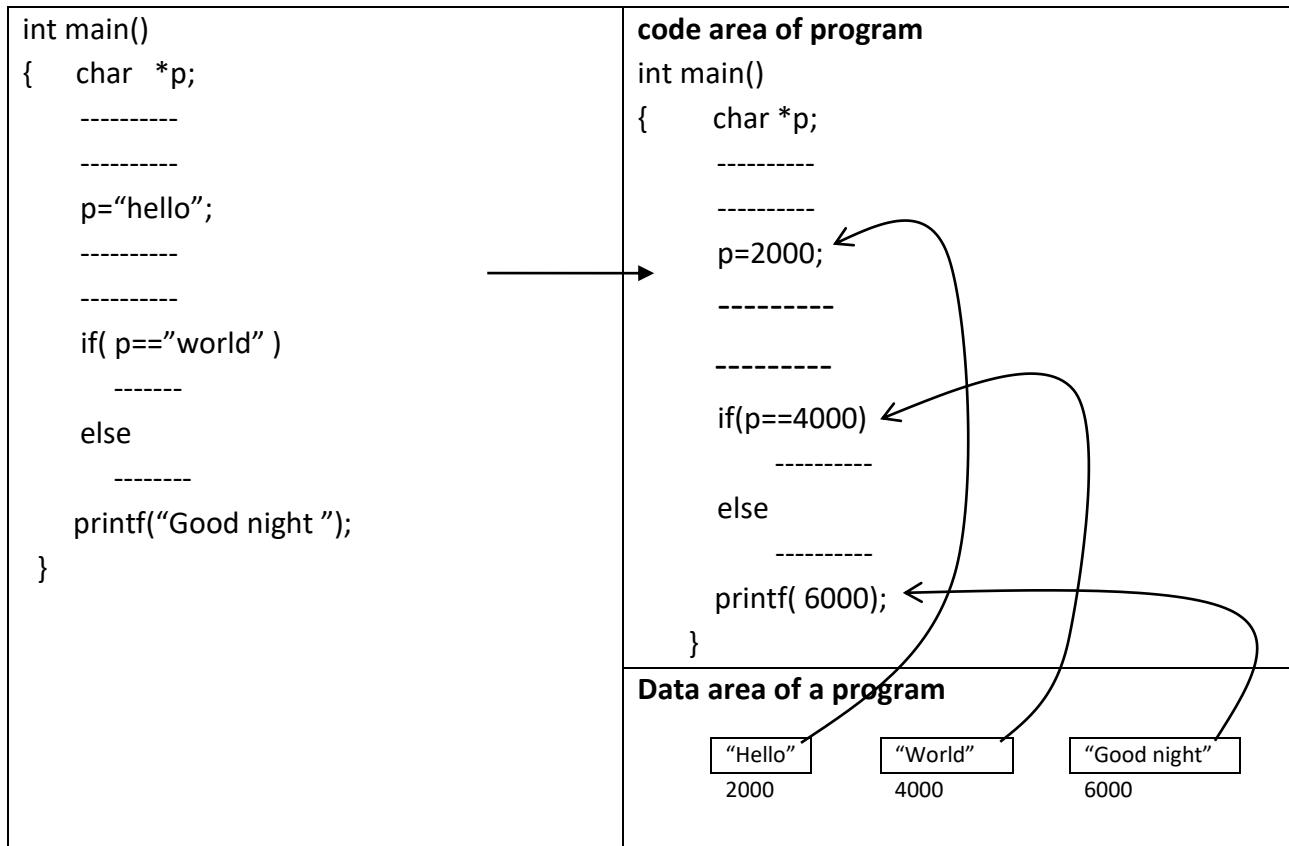
At compile time, the program's code and data are split into two separate entities and stored separately in the executable file. This file is then loaded as is into memory before execution. This can be represented as

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 placed in the data area, while in the code area their addresses are substituted (or linked). In this way, string constants (which are enclosed in quotes) are mapped to their memory addresses in the program.

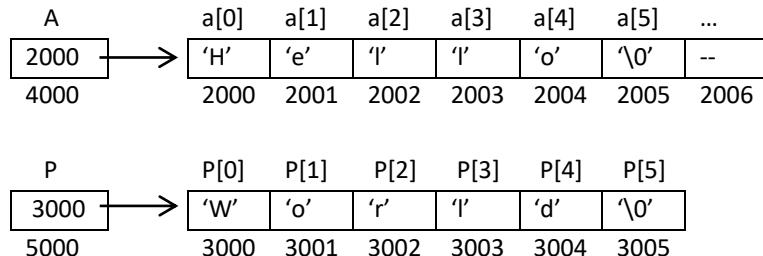
Observe the following picture to see how a string constant's address is mapped in the program.



Pointer to a string

As we know, a string can be in the form of a variable or a constant. In both cases, the strings are handled in a similar way.

```
char a[30] = "Hello";
char *p = "World";
```



As per the first declaration, we can say that the string is in the form of a variable, because it is loaded-initialized into an array, and an array can be treated as a variable. Later in the program, we can replace the string "Hello" with some other string, but the new string's length must be less than 30 characters (the array size).

As per the second declaration, we can say that the string is in the form of a constant. The space allocated for the string "World" is exactly length+1 (6bytes). Here, the address of the first character (&W) is assigned to the pointer 'p'.

Now, using the pointer 'p', we can read the string char by char. Later in the program, we can replace the string "World" with another string, but the new string's length must be the same or shorter than "World".

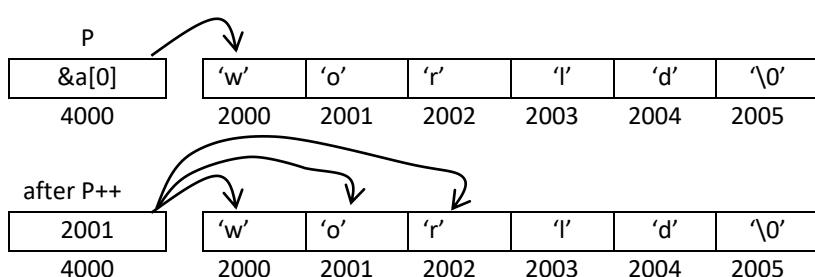
If the new string's length is longer, it will cross memory limits and may cause the program to crash.

This type of pointer initialization is widely used for string constants where no future changes are required, such as company-names, country-names, person-names, codes, labels...etc.

A pointer to a string is just like a pointer to an array, whether the string is a variable or a constant. In both cases, the characters are accessed in similar ways. Observe the above picture. If the pointer 'p' is pointing to the first character in a string, then the expression `*p` accesses the first character, `*(p+1)` accesses the second character, and so on. As we know, the expressions `*p`, `*(p+1)`, `*(p+2)`... can be written in array notation as `p[0]`, `p[1]`, `p[2]`, etc. For example

```
int main()
{
    char a[] = "world", *p; int i;
    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++); // the *p++ is nothing but *p and p++, this is equal to: printf("%c", *p); p++;
}
```

The above two loops print the same output. In the first loop, the pointer 'p' is not moved/incremented, whereas in the second loop, the pointer 'p' advances to the next character each time. After the loop finishes, the pointer 'p' moves to null-char. Let us see how pointer moves



Passing a string to a function

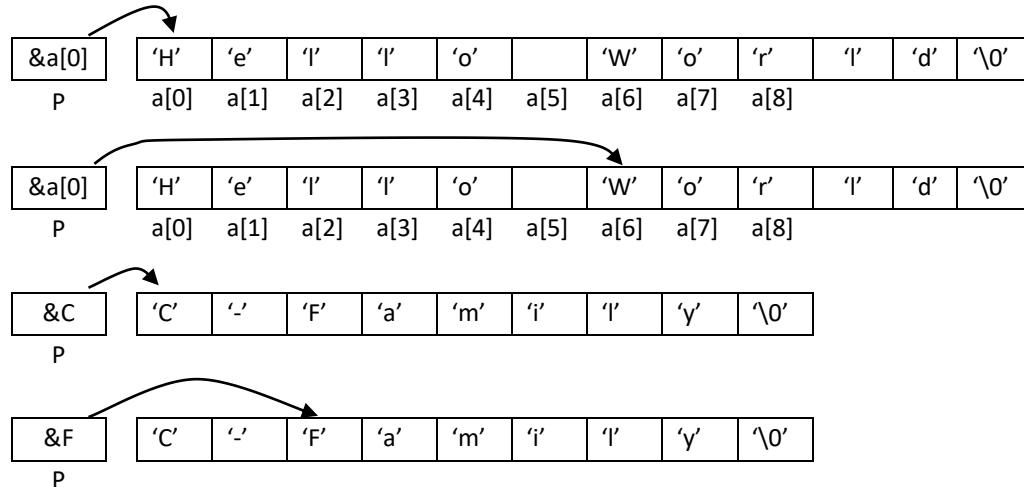
We already know how to pass an array to a function. Similarly, we can pass a string as an argument to a function. Passing a string to a function is the same as passing an array to a function: the address of the string is passed as an argument, and the receiving parameter must be “char*” type pointer. The following program demonstrates how to pass a string to a function

```
int 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 )
{
    for(int i=0; p[i] !='\0'; i++)
        printf ( %c", p[i] ); // printing char by char of string or use printf("%s", p);
}

output: Hello World
    World
    C-Family
    Family
```

The pointer parameter ‘p’ points to the string(s) in every function-call as given below



As we know, the parameter declaration `char *p` in pointer style is informal and sometimes confusing. Most programmers prefer the array-like pointer declaration `char p[]`. This is more formal and clearly indicates that `p` is receiving the address of a string, rather than the address of a single character.

Although the parameter declaration `char p[]` may look like an empty array, it is implicitly treated as a `char *p` pointer. This form of declaration is provided only for ease of understanding, and nothing more.

While manipulating strings, we often need to perform common operations such as copying a string, comparing, reversing, finding a substring, converting text to numeric format, etc.

Since these operations are frequently required in programming, it is preferable to abstract (hide) these tasks by writing separate functions for each operation. The C standard library already provides many string-handling functions such as `strlen()`, `strcpy()`, `strcmp()`, `strrev()`, etc., to help programmers avoid writing such logic from scratch. The following programs demonstrate how string library functions are developed and used. Here, a few sample user-defined functions are implemented to show how the standard library functions work internally.

Finding length of a 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, repeats until 'i' reaches to null
    }
    return(i);      // after the loop, the value of 'i' contains length, so returning 'i' value as length
}
int 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 necessary to copy a string from one location to another. For this, if one tries to copy a string using the assignment operator, the compiler shows an error.

```
for example: char X[20] = "Hello World";
            char Y[20];
            Y = X;           // Y = &X[0];
```

Here, we expect that the string "Hello World" is copied from X[] to Y[], but the compiler gives an error because we are unknowingly trying to assign the address of '&X[0]' to 'Y'. (If Y were a pointer, this would still be incorrect) The solution is to copy the char-by-char from the source array to the 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-char at end of string.
}
```

Main() fn to check the above function

```
int 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

some valid and invalid assignments on strings

// valid assignment char a[4]; a[0]='A'; a[1]='B'; a[2]='C'; a[3]='\0';	// valid assignment char a[4]; char *p=&a[0] p[0]='A'; p[1]='B'; p[2]='C'; p[3]='\0';	// invalid assignment char *p; p[0]='A'; p[1]='B'; p[2]='C'; p[3]='\0';	// valid assignments char *p; p=malloc(4); p[0]='A'; p[1]='B'; p[2]='C'; p[3]='\0';
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

```
int 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 the contents of two arrays are the same, the output will be "not equal" because, unknowingly, we are trying to compare the base addresses of the two arrays.

The expression if ($x == y$) is nothing but if ($\&x[0] == \&y[0]$). As we know, two arrays will not be stored at the same location in RAM, so the result is always "not equal".

In the second if statement also, the base addresses of two strings (if ($\&H == \&H$)) are compared, and again we get "not equal", because two string constants will not be stored at the same location.

The solution is to compare the strings character by character using a loop, as shown below:

```

void mystrcmp( char x[], char y[] )
{
    int i;
    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, the function returns 0. Otherwise, it returns the ASCII difference between the first unmatched characters of the two strings.

It returns a positive value if string1 > string2 in dictionary (alphabetical) order, or a negative value otherwise.

For example, if the two strings are:

```

x="Hello" and y="Hello" then returns (0);
x="World" and y="Words" then returns ('l'-'d'); // ascii difference is +ve , here World>Words
x="Words" and y="World" then returns ('d'-'l'); // ascii difference is -ve , here Words<World

```

The main() function to check the above function is

```

int 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");
}

```

Concatenation of two strings

Sometimes, it is necessary to attach one string to the end of another string. This operation is 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 the null character is found
    }
    for( j=0; y[j]!='\0'; j++) // attaching the second string at end of first string
        x[ i+j ] = y[ j ];
    x[ i+j ]='\'0';           // inserting null character at end of first string
}
int 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

Reversing a string

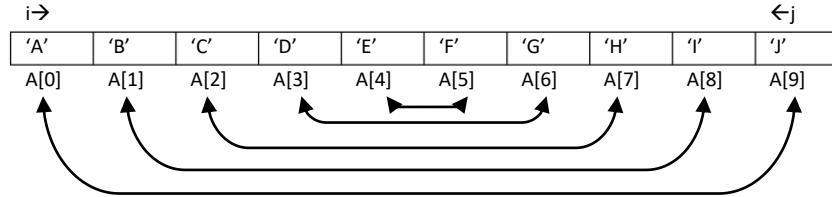
```

void myStrrev( char *a )
{
    int i;
    for(i=0; a[i]!='\0'; i++)
    {
        // loop ends when 'i' value is reached to null char, finally 'i' contains length
    }
    for( j=i-1, i=0; i<j; i++, j--) // before loop starts, 'i' points to first element, and 'j' points to last element in the string.
    {
        t=a[i];          // swapping
        a[i]=a[j];
        a[j]=t;
    }
}

int main()
{
    char a[100];
    myStrrev( a );
    printf("\n string after reversing %s", a );
}

```

ip: ABCDEFGHIJ
op: JIHGFEDCBA



String library functions

In the previous examples, we have seen how to write and use string functions. The C standard library provides a wide range of well-developed string functions, which help minimize coding time and allow us to build compact and reliable programs without rewriting the same code again and again. This library also simplifies the logic of complex programs, without feeling any difficulty while handling strings.

The prototypes of these functions are available in the string.h header file, so we can simply include it and use the functions directly in our programs

strlen() : This function takes the address of a string as an argument and returns its length. It counts and returns the number of characters in the given string without including the null character.

syntax is: intVariable=strlen(string); and proto-type is: int strlen(char *);

```
int main()
{
    char a[30] = "Hello World";
    int k;
    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.

```
int 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*);

```
int 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); and proto type: char* strrev(char*);
```

```
int main()
{
    char a[20] = "Hello"
    strrev(a);
    printf("output1 = %s", a);
    strrev(a);
    printf("output2 = %s", a);
}
```

output1: olleH
output2: Hello

strcmp(): Compares two strings to check whether they are equal or not. If both strings are equal, it returns 0.

Otherwise, it returns the ASCII difference of the first unmatched characters (positive or negative).

If the returned value is +ve, then string1 > string2 in dictionary order; otherwise, string1 < string2.

```
int main()
{
    char a[30] = "Hello World", 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

strcmpl() or stricmp(): It compares two strings, which is same as strcmp(), but it ignores the upper/lower case of alphabets. Syntax: int strcmpl(string1, string2);

```
int main()
{
    if(strcmpl("DELHI", "delhi") == 0)
        printf("Both are equal");
    else printf("Not equal");
}
```

Outputs: Both are equal

strncmpl() or strnicmp(): It compares two strings up to some specified number of characters, and also ignores the case difference. Syntax: int strncmpl(string1, string2, no-of-chars-to-compare);

```
int main()
{
    if(strncmpl("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);

```
int 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);

```
int main()
{
    char a[]="Jack n JILL";
    strlwr( a);
    printf("%s", a);
}
```

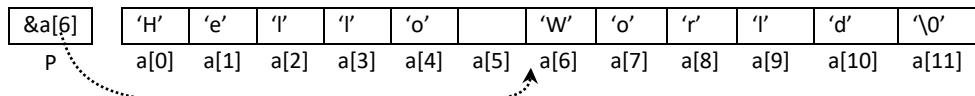
Output: jack n jill

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);

```
int main()
{
    char *p;
    p=strchr("Hello world", 'w');      // returns "&w" in "hello world" string.
    printf("%s", p);
}
```

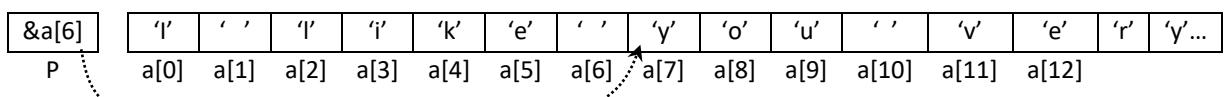
This printf() will prints the message "world" , because 'p' is pointing to rest of the string "world"



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.

```
int main()
{
    char *p;
    p=strstr("I like you very much", "you");
    printf("%s", p);
}
```

This printf() will prints 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 a 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 string 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
```

sprint(): It is like printf() function but with a little difference. The printf() puts the output data on the screen, whereas sprint() 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 KB.

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, it is better to revise the topic: pointer to 2D array.)

This chapter mainly concentrates on how to represent an array of strings and perform operations such as inputting, printing, sorting, searching, and finding substrings, etc. To store a group of strings, we need a data structure like a two-dimensional array of size $N \times M$, where N is the number of strings and M is the maximum length of a string.

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 method is to scan one string after another from the keyboard using a loop statement. When we use a two-dimensional array, we need to pass the base address of each row of the array to the `scanf()` function. The following program shows how to scan and print strings:

```
int 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 that the two-dimensional array expression `a[i][j]` is converted into `*(a + i * column-size + j)`.

Whereas the expression `a[i]` is converted into `(a + i * column-size)`, which gives the address of the i -th row.

(refer pointer to 2D array topic, page 199)

Finding biggest string from N strings.

(Here biggest means alphabetically comes to last in dictionary order)

```
#include<stdio.h>
#include<string.h>
int main()
{
    char a[20][30], *temp;
    int 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 the largest; therefore, its base address is stored in the temp pointer. Later, all strings are compared with temp. If a larger string is found in the rest of the array, its address is copied to temp. In this way, the address of the largest string will finally be stored in temp.

Sorting of 'N' Strings

We can sort strings just like integers, but strings cannot be compared using the `>` operator, because in that case their base addresses are compared instead of their characters. Therefore, the library function `strcmp()` is used to compare two strings. The following program gives a demonstration of sorting N strings:

```
#include<string.h>
int 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]);
}

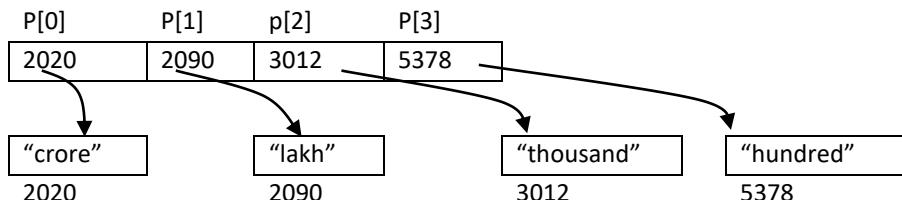
```

Initialization of strings to a pointer array

If string constants are fixed and do not require any modification during program execution, then it is better to initialize them to a pointer array rather than a normal 2D array. This approach saves a lot of memory space and is also easier to manage—for example, swapping strings, sorting strings, inserting a new string between existing strings, or deleting a string. In this method, addresses are manipulated instead of the actual 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



Printing a single digit in English words

input: 7

output: seven

```

int main()
{
    char *a[] = {"zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine"};
    printf("enter a digit :");
    scanf("%d", &n);
    if( n<0 || n>9 )
    {
        printf("invalid digit");
        return;
    }
    printf("output is %s", a[n] );
}

```

Printing a number in English words

Input: 2345

Output: two thousand three hundred and four five

```
#include<stdio.h>
int main()
{    char *a[] = {"", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine"};
    char *b[] = {"", "", "twenty", "thirty", "fourty", "fifty", "sixty", "seventy", "eighty", "ninty"};
    char *c[]={ "crore", "lakh", "thousand", "hundred and", ""};
    long int d[]={10000000L, 100000L, 1000, 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 **recursion**. It is the process of defining something in terms of itself, and is sometimes referred to as a **circular definition**.

The syntax, usage, and execution of a recursive function are the same as those of a normal function. There is no technical difference between a recursive function and a non-recursive one. However, for beginners, the logic of recursion is somewhat difficult to understand, and writing programs using it can be challenging. The major source of confusion lies in understanding how self-calling, self-returning, arguments, parameters, and local variables are managed during recursive calls.

If you are already familiar with functions, you can more easily follow the logic of recursion by observing some simple examples. In the following sections, we will explain recursion with pictures to help you understand the basic concept. Before going into examples, let us look at some mathematical analysis of **recurrence relations**.

1) $f(n) = n + n/2 + n/4 + n/8 + \dots + 1$

the mathematical recurrence relation is:

$$f(n) = n + f(n/2) \text{ if } n > 1$$

$$f(n) = 1, \text{ if } n == 1$$

2) $\text{fact}(n) = n * n-1 * n-2 * \dots * 3 * 2 * 1$

the recurrence relation is:

$$\text{fact}(n) = n * \text{fact}(n-1), \text{ if } n > 1$$

$$\text{fact}(n) = 1, \text{ if } n == 1$$

3) $\text{power}(x,y) = x * x * x * x \dots y \text{ times}$

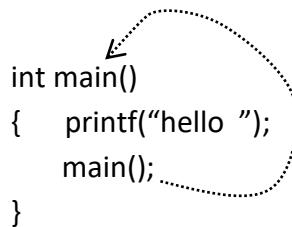
the recurrence relation is:

$$\text{power}(x,y) = x * \text{power}(x,y-1), \text{ if } y > 1$$

$$\text{power}(x,y) = 1, \text{ if } y == 1$$

First demo program

```
int main()
{
    printf("hello ");
    main(); // self-calling (recursive-call)
}
```



output: hello hello hello hello ... until stack overflow (memory full)

Here, the `main()` function is called recursively inside its own body. Therefore, the control re-enters the body of `main()` again and again, and it repeatedly prints "hello" many times. However, in every recursive call, the operating system (os) allocates memory to store the function's return address. After several calls, this memory becomes full, and the program is forcefully stopped by the OS. This forced stop is also known as program crash. This is just a demo program to illustrate how recursive calls are made.

Let us see the following example to understand how local variables behave inside a recursive function.

```

int 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, a new copy of 'k' is created and initialized with 1. So the output is 1, 1, 1, 1.....

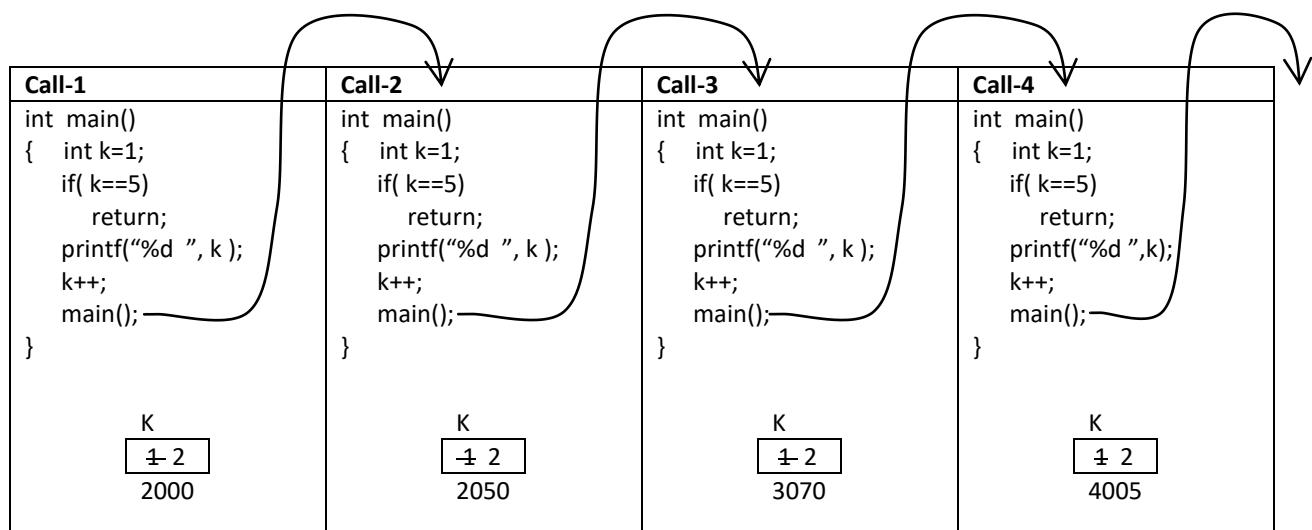
Here, the value of K never reaches 5, because the incremented value (k++) in one call does not affect the next call of k, so it always prints 1,1,1,1..... until memory is full.

Note: Since a new copy of k is created in every call, memory becomes full after some calls and leads to a program crash.

Three points to understand Recursion

- + When a recursive function calls itself, the programmer should imagine that it is like calling another function of same code as shown in the figure below. Of course, in memory, the same function body will be executed repeatedly, similar to a loop.
- + For every recursive call, one set of local variables (in this example, k) is created and remains available until that call is terminated. That is, when the call is terminated, all its local variables are freed (deleted).
- + A recursive function executes like a loop statement. As we know, every loop has a terminating condition. In the same way, to stop recursive calls, there must be a termination condition, which is often called the base condition. Generally, it appears at the beginning of the function (we will see this in the next examples).

This figure explains how the recursive calls should be imagined for above example



Here, in every call, one fresh copy of k is created and initialized with 1. Observe that the memory address of k in the first call is 2000, in the second call it is 2050, and so on. So a separate memory is created for k in every call. Therefore, the incremented value (k++) in one call will not affect the next call. So, the output is 1 1 1 1 ... (until memory is full).

Let us take k as a parameter instead of a local variable, and see the following example.

Expect the output of following program

```

int main()
{
    display(1); // passing value '1' as argument
}

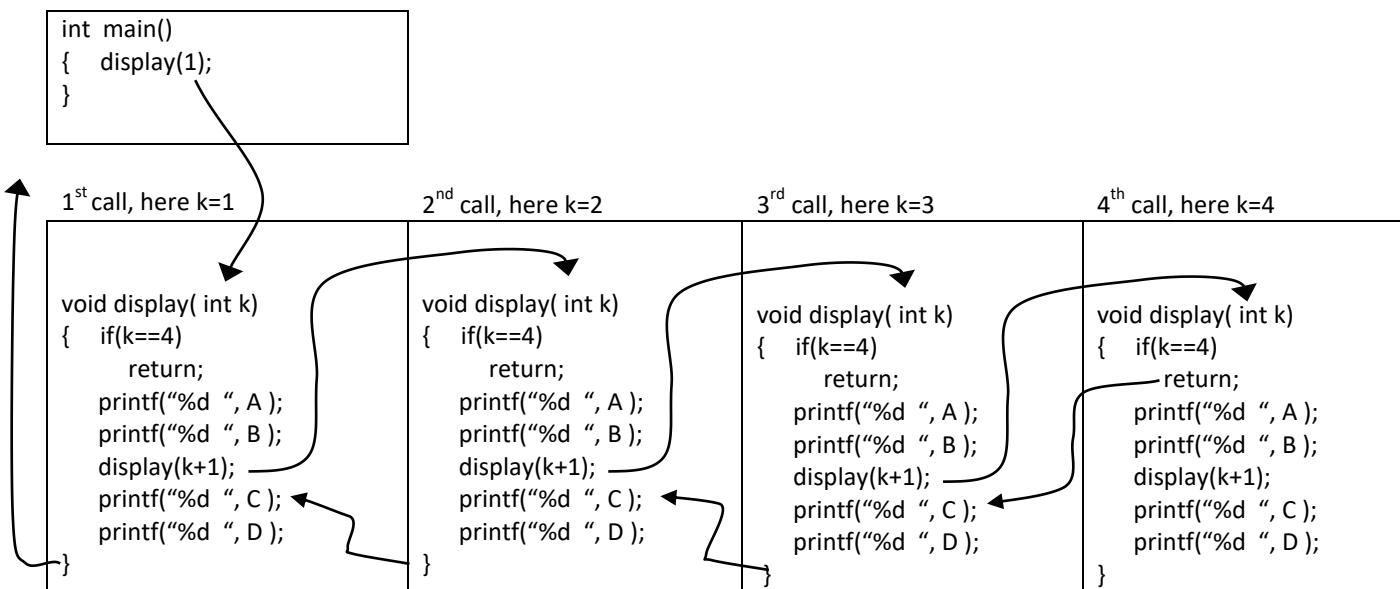
void display( int k )
{
    if(k==4)
        return; // termination of recursion
    printf(" A ");
    printf(" B ");
    display(k+1); // recursive call
    printf(" C ");
    printf(" D ");
}

```

Output: AB AB AB CD CD CD

For every next call, we are passing $k+1$ as the argument. Therefore, in the first call $k = 1$, in the second call $k = 2$, and in the fourth call $k = 4$. Here, the recursion terminates and returns the control.

When the above return statement gets executed in the 4th call, many people think that the control immediately transfers back to `main()`. This is a common misunderstanding. Actually, the control returns to the previous call in this series of calls. The following picture explains this clearly.



* The last closing brace '}' of the function works as a return statement. Observe the arrow in the above code

* Here, the 4th call returns to the previous 3rd call, and the 3rd call returns to the previous 2nd call. In this way, recursive calls are returned and terminated. The above picture gives the best view of how recursive calls can be imagined.

* In a recursive function, all instructions above the recursive call are executed just before going to the next call. So here, the output is AB AB AB. All instructions below the recursive call are executed while returning to previous calls. So here, the output is CD CD CD.

The main difference between a loop and recursion is that a loop works on a single set of variables, whereas recursion works on multiple sets of variables.

*Printing 1 2 3 3 2 1 recursively

```

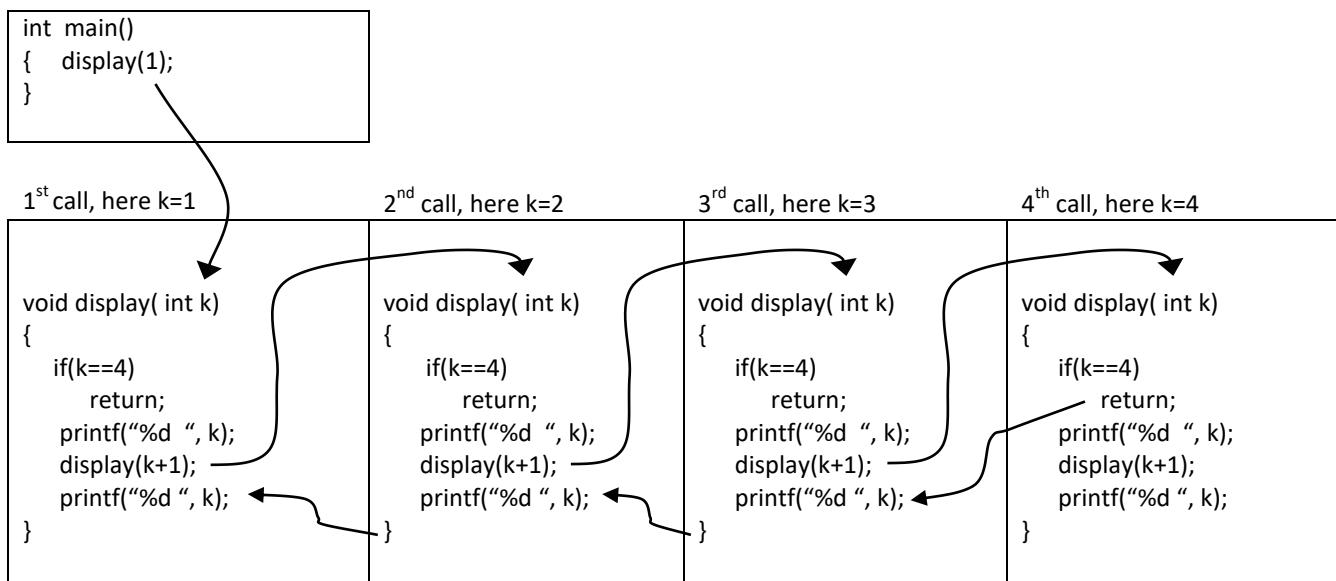
int main()
{
    display(1); // passing argument 1
}

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” is printed by the first printf(“%d”, k), and the output “3 2 1” is printed by the second printf(“%d”, k) statement. The first printf() executes just before the next call is made, whereas the second printf() executes while returning to the previous calls.

The following figure shows how the output 1 2 3 3 2 1 prints on the screen.



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

```

int main()
{
    scanf("%d", &N);
    show(N);
}

void show(int N)
{
    if( N==0) return;
    printf("%d ", N);
    show(N/2); // sending half value of N to the next call
}

```

Most of the recursive programs covered in this chapter are simple, and they can also be done using loop control statements instead of recursion. For these problems, a loop is simpler than recursion. However, to make the concept of recursion easier to understand, these problems are explained using recursion. But the last two problems in this chapter cannot be solved using loops.

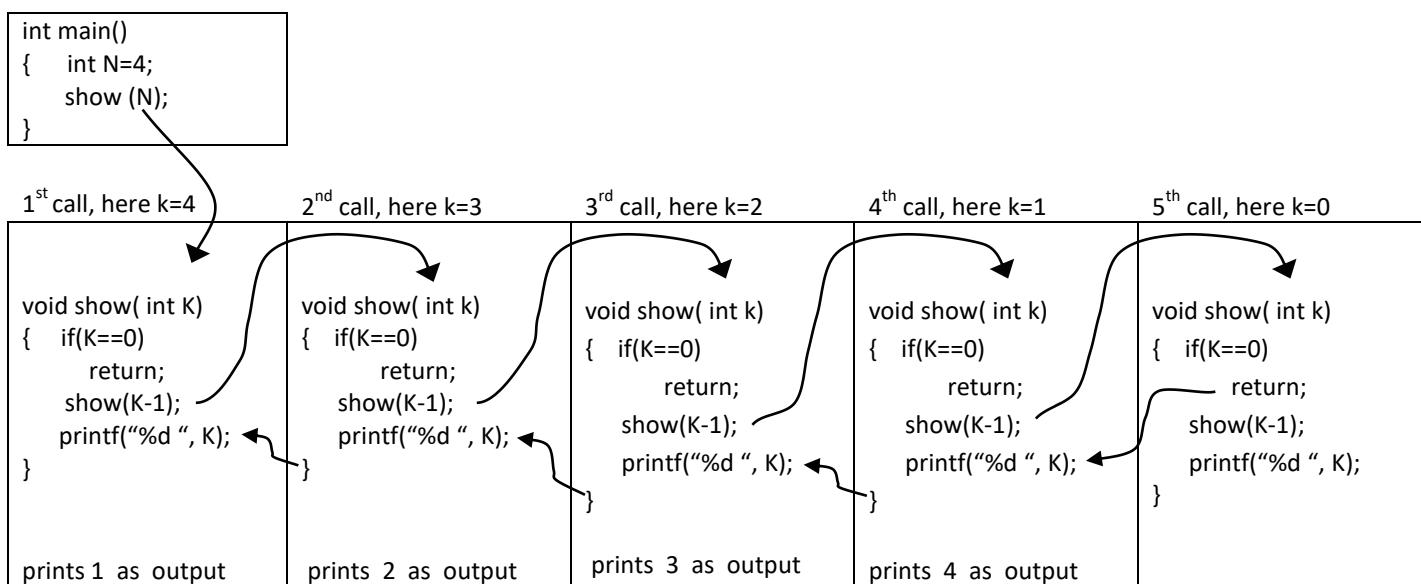
Printing 1 to N numbers using recursive function

```

ip: N=4
op: 1, 2, 3, 4.
int main()
{
    scanf( "%d", &N );
    show(N);
}
void show( int N )
{
    if(N==0) return;
    show( N-1 );
    printf("%d ", N );    // the value of N is printed while returning control to the previous call
}

```

When N reaches 0, the recursive calls stop, and the value of N is printed from the last call to the first call while returning control to the previous calls. This is shown in the picture below.



Guess the output

```

int 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

int 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

```

int main()
{
    printTable( 8 , 20 );      // here value '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

```

int main()
{
    show(1);                  // value '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

```

int main()
{
    show(1, 7);              // value '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... , whereas count=7,6,5,4,3,2,1
    show( P*2 , count-1 );
}

```

Generating and printing list of numbers from N to 1

Here, N is input from the keyboard, and the program prints the list of numbers until the value of N becomes 1.

If N is even, then the next number of N is $\rightarrow N/2$ (half).

If N is odd, then the next number of N is $\rightarrow 3N + 1$.

For example, if the input N is 13, then the output will be: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. V

```
int 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

```
int main()
{
    show( 345 );
}

void show( int N )
{
    if( N==0 )
        return;
    show(N/10);
    printlnEnglish( N%10 );
}

void printlnEnglish( 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

```
int 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
int 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() function recursively for 5 times
}
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 by 2 and collect the remainders. The collection of remainders forms a binary number.

```

int main()
{
    show( 13 );
}

void show( int N )
{
    if( N==0 )
        return;
    show( N/2 );          // N=N/2→ 13, 6, 3, 1
    printf("%d", N%2 );   // output is → 1%2, 3%2, 6%2, 13%2
}

```

Printing all factors of 15

ip: 15 , op: 1, 3, 5, 15

```

int 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 (following code doesn't work, check out)

```

int 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 1 st call int sum=0 sum=sum+7 = 0+7	At 2 nd call int sum=0 sum=0+7 =0+7	At 3 rd call int sum=0 sum=sum+7 =0+7	At 4 th call int sum=0 sum=sum+7 =0+7	At 5 th call int sum=0 return sum // returns '0'
--	---	---	---	--

Output: 0 (wrong output)

The above function produces the wrong output. This is because, for every recursive call, a fresh copy of the local variable 'sum' is created and initialized to 0. So, the added value 7 does not carry over to the next call's sum. Finally, this function returns 0 as the output.

To solve this problem, take sum as a parameter and pass its value to the next call. See the next problem.

Above program with 'sum' as parameter (this code works fine)

```

int 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 1 st call sum=0 sum=sum+7 = 0+7	At 2 nd call sum=7 sum=sum+7 = 7+7	At 3 rd call sum=14 sum=sum+7 =14+7	At 4 th call sum=21 sum=sum+7 =21+7	At 5 th call sum=28 return sum; // returns 28
--	--	---	---	---

Here, sum is a parameter (not a local variable). The added value 7 is passed to the next calls. At the 1st call, sum=0; at the 2nd call, sum=0+7; at the 3rd call, sum=0+7+7, and so on. Finally, returns total sum as 28

Above program without 'sum' as parameter (simplified version)

```

int main()
{
    k=find( 4 );
    printf( " %d ", k ); //output is: 28
}

int find ( int n )
{
    if( n==0 )
        return 0;
    return 7 + find( n-1 );
}

```

Let us see the following picture to understand how function calls are made and how the return value is passed back.

First call (n=4)	Second call (n=3)	Third call (n=2)	Fourth call (n=1)	Fifth call (n=0)
<pre>int find(int n) { if(n==0) return 0; return 7+find(n-1); } // return 7+7+7+7+0</pre>	<pre>int find(int n) { if(n==0) return 0; return 7+find(n-1); } // return 7+7+7+0</pre>	<pre>int find(int n) { if(n==0) return 0; return 7+find(n-1); } // return 7+7+0</pre>	<pre>int find(int n) { if(n==0) return 0; return 7+find(n-1); } // return 7+0</pre>	<pre>int find(int n) { if(n==0) return 0; return 7+find(n-1); } // return 0</pre>

Note: I have solved many problems using recursion, but most result-calculating functions come under the two logics explained above. The logic of taking sum as a parameter is suitable for all problems, but the last logic is simpler. However, the last logic may not be suitable for all problems.

Do not use global or static variables. These variables will be discussed at the end of this topic.

Finding factorial of a given number using recursive function

The recursive definition of a factorial can be written in mathematical form as

$$\begin{aligned}
 f(n) &= n * f(n-1) && \text{if } n > 1 \\
 f(n) &= 1 && \text{if } n = 0 \text{ or } 1 \\
 f(5) &= 4 * f(3) \\
 &\quad 4 * 3 * f(2) \\
 &\quad 4 * 3 * 2 * f(1) \\
 &\quad 4 * 3 * 2 * 1 = 24
 \end{aligned}$$

```

int fact(int n)
{
    if(n==1)
        return 1;           // base condition
    return n*fact(n-1);   // fact(n-1) is a recursive call
}

int 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)
<pre>int fact(int n) { if(n==1) return 1; return n*fact(n-1); } // return 4*3*2*1</pre>	<pre>int fact(int n) { if(n==1) return 1; return n*fact(n-1); } // return 3*2*1</pre>	<pre>int fact(int n) { if(n==1) return 1; return n*fact(n-1); } // return 2*1</pre>	<pre>int fact(int n) { if(n==1) return 1; return n*fact(n-1); } // return 1</pre>

Finding factorial using an iterative process (loop) is simpler than recursion. This is just a demo program to explain how recursive functions are executed.

The last call returns 1, and it is substituted into its previous-call statement. The previous-call then returns $2*1$, which is again substituted into its previous-call. In this way, the final result 24 is returned to the main() function.

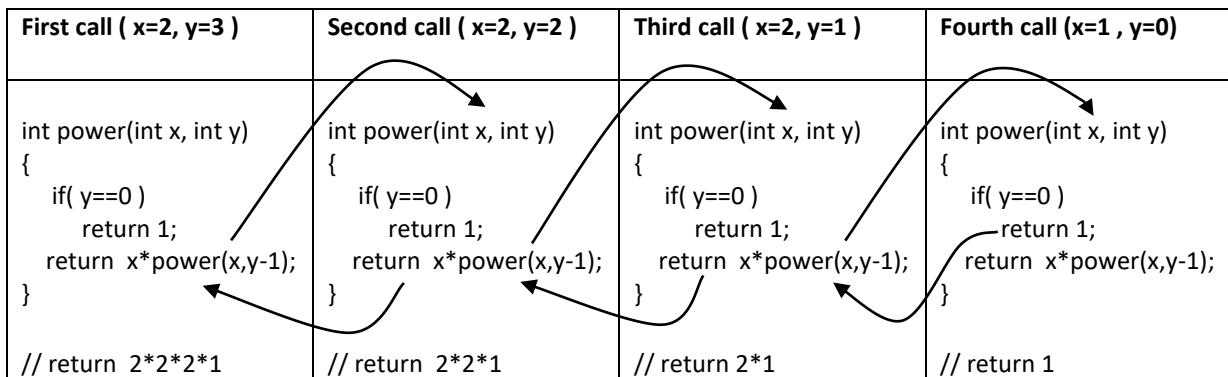
Finding x^y , where 'x' is base and 'y' is exponent

The recurrence relation is

$$\begin{aligned} \text{fun}(x,y) &\rightarrow x * \text{fun}(x, y-1) && \text{if } y>1 \\ \text{fun}(x,y) &\rightarrow 1 && \text{if } y==0 \end{aligned}$$

```
int power(int x, int y)
{
    if(y==0)
        return 1;
    return x*power(x, y-1);
}

int main()
{
    int a=2,b=3,c;
    c=power(a,b);
    printf("\n %d^%d=%d",a,b,c);
}
```



Guess the output

```
int main()
{
    k=find( 345 );
    printf( "%d ", k );
}

int find ( int N )
{
    if( N==0 ) return 0;
    return N%10 + find( N/10 );
}
```

Guess the output

```
int main()
{
    show( 1 , 2 );
}

void show ( int X , int Y )
{
    if( X>20 ) return;
    printf("%d ", X );
    show( X+Y, Y );
}
```

Recursive function to find GCD. (Greatest Common Divisor)

This program finds the greatest common divisor of two numbers

Let the input values be x and y , where $x < y$. (The logic given below also works when $x > y$)

The recursive definition can be written in mathematical form as

$$\begin{aligned} f(x,y) &= f(y \% x, x) \quad \text{if } y \% x \neq 0 \\ f(x,y) &= x \quad \text{if } y \% x == 0 \end{aligned}$$

The logic is to continuously divide the y by x until the remainder becomes zero.

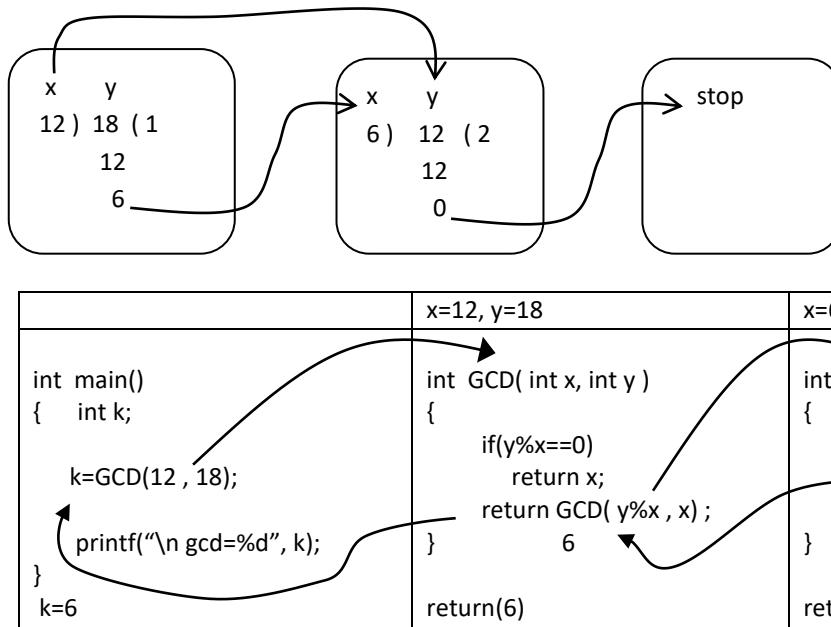
Here, take y as the dividend and x as the divisor.

If $y \% x$ is zero then x will be the **GCD**, and the process stops.

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



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, z; // here x, y, z are local variables to the function.
    if(count==0) return; // stop condition
    printf("%d ", x);
    z=x+y; // generating next 'z' term
    x=y; y=z; // moving x,y to next terms
    fibo(count-1); // repeat until 'count' down to 0.
}
int 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..... (Wrong output)

The above function produces the wrong output because, for every recursive call, a fresh copy of the local variables x and y is created and initialized with 0 and 1. Thus, the incremented/changed values of x and y do not affect the next successive calls (we have already discussed this many times).

To solve this problem, take x and y as parameters and pass the updated values from one call to the 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 );
}

int 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 and y is enough for all recursive calls, rather than creating a fresh new copy in every call. To avoid creating these copies of x and y, we can declare them as static or global variables. But this creates another problem! Once a static/global variable is created, it will not be destroyed even after the function call ends; moreover, it cannot be reinitialized for the next upcoming calls for same. Let us see following example

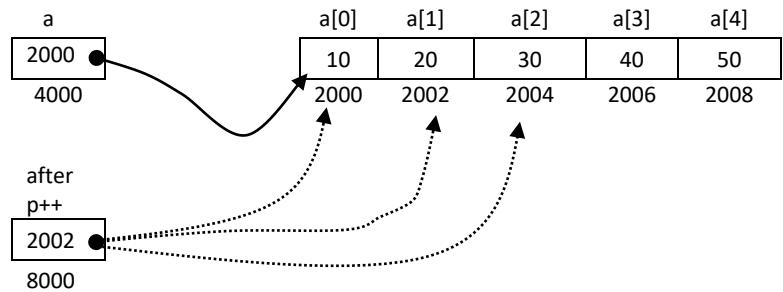
```
int main()
{
    show();      // at this call, the output is 1 to 5
    show();      // in this second call also we expect 1 to 5, but no output will be displayed
}
int k=1;
void show( )
{
    if(k>5) return;
    printf("%d ", k );
    k++;
    show( );
}
```

We expect the output as 1 to 5, 1 to 5 (since the show() function is called twice from main()). But as k is a global variable, only one copy of k is created and shared among all calls. After printing 1 to 5 in the first call, k becomes 6 and is not re-initialized to 1 after the recursive calls end. Therefore, no output will be displayed for the second call. Because of this side effect, we do not recommend using static or global variables in recursion.

Guess the output of following program

```
Int 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

```
int main()
{
    show( "Hello" );
}

void show ( char *p )
{
    if( *p=='\0' ) return;
    show( p+1 );
    printf("%c", *p );
}
```

Guess the output of following program

```
int 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.

$$\text{fibo}(n) = \text{fibo}(n-1)+\text{fibo}(n-2). \text{ if } n>2$$

$$\text{fibo}(n) = 0. \quad \text{If } n==1$$

$$\text{fibo}(n) = 1. \quad \text{If } n==2$$

0	1	1	2	3	5	8	13	21	34	55	89	...
1	2	3	4	5	6	7	8	9	10	11	12	..

$$\text{Fibo}(5) \rightarrow \text{fibo}(4)+\text{fibo}(3)$$

$$\rightarrow \text{fibo}(4-1)+\text{fibo}(4-2)+\text{fibo}(3-1)+\text{fibo}(3-2)$$

$$\rightarrow \text{fibo}(3-1)+\text{fibo}(3-2) + \text{fibo}(2)+\text{fibo}(2)+\text{fibo}(1)$$

$$\rightarrow \text{fibo}(2)+\text{fibo}(1)+\text{fibo}(2)+\text{fibo}(2)+\text{fibo}(1)$$

$$\rightarrow 1+0+1+1+0 \rightarrow 3$$

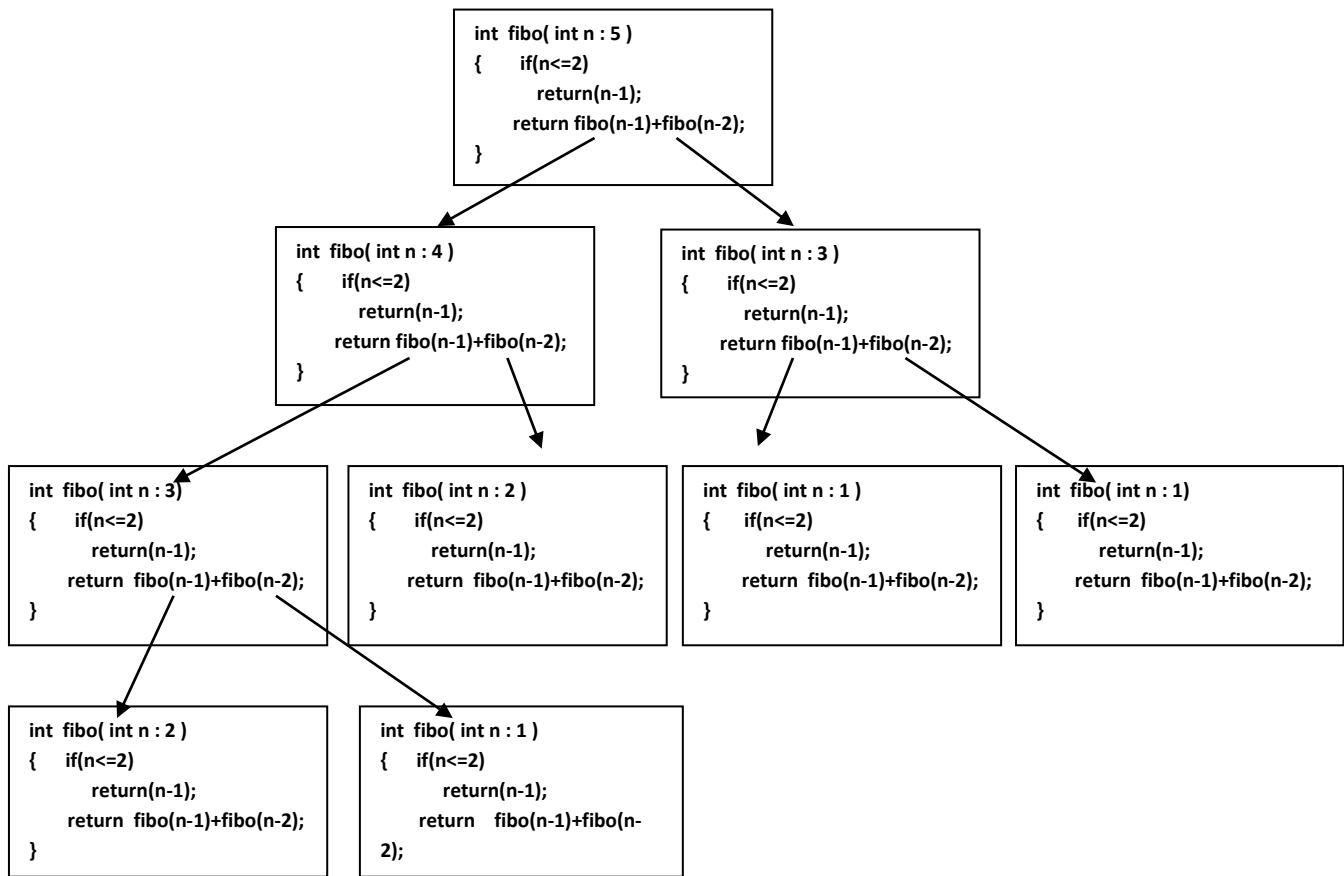
```

int main()
{
    int n=5;
    x=find(n);
    printf(" the Nth term = %d", x);
}

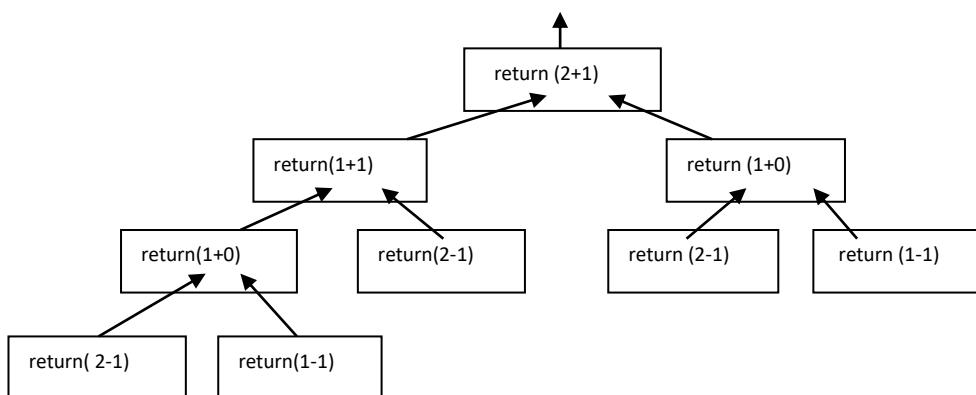
int fibo( int n )
{
    if(n<=2) return(n-1);
    return fibo(n-1) + fibo(n-2);
}

```

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 a knight. Here, the knight visits every cell only once and follows the rules of its movement, i.e., it moves only in an L-shape. This function takes the x, y coordinates of the first step on the 8×8 board and displays the order of movements in terms of the step number of every cell. The 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
int 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++)
    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 a non-recursive one with the help of loop control and sometimes with a stack. However, by doing so, the complexity of the code may increase, it becomes more prone to logical errors, and even debugging the iterative code can be difficult. So, for problems with a recursive nature, recursion is simpler than the iterative process.

A recursive function slows down the execution of a program because of the creation and destruction of variables. Additional storage space is also required to store the function return address. However, writing code using recursion is easy, clear, and easier to modify.

We know that passing arguments from one function to another is common in modular programming. If any set of arguments needs to be passed to a recursive function, then the same set of arguments has to be passed for the next recursive calls, even if they are unnecessary. In some cases, this is useful, but in other cases, it seems unnecessary (i.e., one copy is enough for all calls). There is no alternative to overcome this problem. One might think of using static or global variables, but once a static variable is created, it will exist until the end of the program(until main() fn is terminated). The second time the same function is called, it will not be reinitialized automatically. So, using such variables is not recommended.

Moreover, server-based applications run continuously for months or even years. If any local variable is declared as static or global variable, it will remain in memory unnecessarily for that entire duration. Furthermore, the excessive use of global variables often leads to misuse and makes the application unreliable.

Conclusion

If your recursive application needs to be used frequently by many users, which results in creating unnecessary copies of local variables, and if you are capable of writing non-recursive code, then it is most advisable to implement it in a non-recursive way, as it would be faster and take less space.

If an application is very complex, has rare usage, and requires little space for local variables, then it is better to implement it recursively rather than non-recursively. Recursion is simple and easy to code.

It is the programmer's choice which method should be used and when. If execution time and space are not constraints, then it is better to implement the application recursively.

The implementation of recursion-based problems in a non-recursive way using loops and stacks is an advanced concept. That topic is beyond the scope of this book.

Memory Allocation Systems

In many real time systems, the size of data is 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.

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.

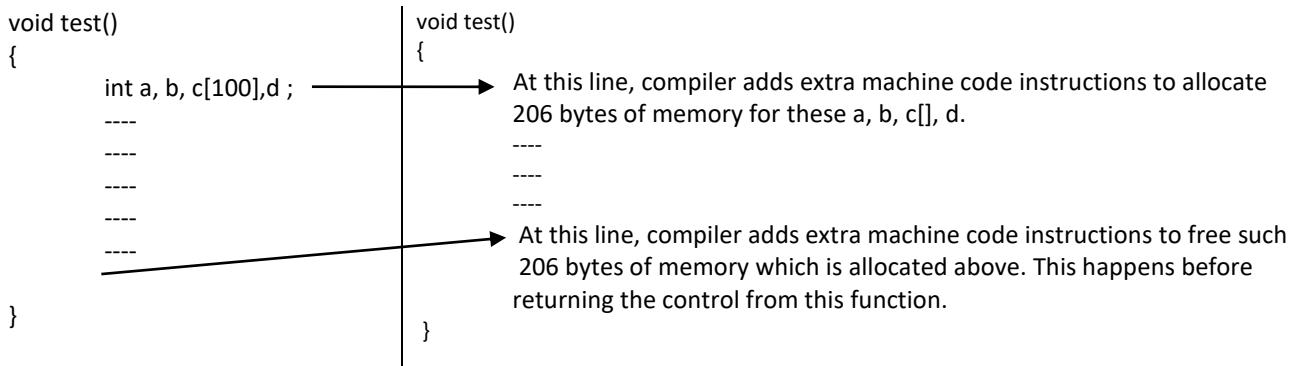
Of course, from the beginning of this book, we have been using this static allocation system only.

Allocating memory at runtime based on input size 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 to our program for allocation/de-allocation 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



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(int=4 bytes), phone-number(int=4 bytes),...

Dynamic Memory Allocation system

In many real-time systems, the size of data is not known or cannot be predicted until runtime. In such cases, dynamic allocation is the preferred choice. Dynamic allocation refers to allocating memory during the execution of a program. Here, the programmer must write explicit instructions to allocate and de-allocate memory with the help of pointers.

This dynamic memory allocation (DMA) system allows us to create, expand, shrink, or even release memory at runtime. This feature is extremely useful for efficient memory management, especially when working with large collections of data through data structures such as arrays, lists, stacks, queues, trees, graphs, and so on.

To support this feature, C provides four built-in functions:

- `malloc()` – to allocate memory
- `calloc()` – to allocate memory and initialize it to zero
- `realloc()` – to resize previously allocated memory
- `free()` – to release allocated memory

malloc()

The malloc function allocates a block of memory of the user-wanted size in bytes and returns the starting address of the allocated block. This memory is raw, allowing us to store values of any type, such as int, float, char, etc. Since malloc() does not know in advance what type of data the programmer intends to store, it returns the address as a void* type. Before using this memory, it must be type-cast to the appropriate pointer type.

prototype: void* malloc(unsigned int);

syntax: **pointer= malloc(number of bytes to allocate);**

example: p=malloc(10); // creates 10 bytes of memory and returns starting address to 'p'

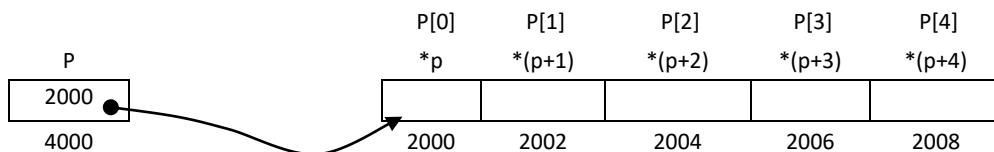
In the example above, the malloc() function allocates 10 bytes of memory in RAM and returns the starting byte address, which is then assigned to p. One question arises: what type should the pointer be? This fairly depends on the type of data we are going to store in this memory. For instance, if you are going to store an array of float values, then the pointer should be of type float*. Remember that malloc() returns just an address with no specific type; that is, it returns a void* type address. Therefore, it should be type-cast according to the pointer type. This is shown below:

Example: Allocating memory for 'n' integers

```
int n, *p;
scanf("%d", &n);
p=(int*) malloc( n*sizeof(int) );
```

Converting void* to int*

let us see, how above allocated memory is mapped to the pointer 'p'



How to access dynamic memory?

Accessing dynamic memory is the same as accessing array elements through a pointer.

Observe the following expressions:

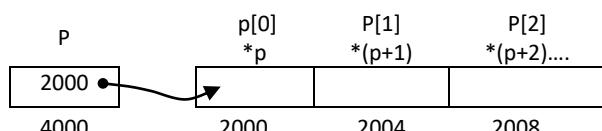
- *(p+0) → p[0] accesses the first location (first 2bytes)
- *(p+1) → p[1] accesses the second location (second 2bytes)
- *(p+2) → p[2] accesses the third location
- *(p+ i) → p[i] accesses the i+1th location.

The link between arrays and pointers was already discussed in the pointers chapter, please refer to it.

If you are creating memory for floating-point values, then a **float*** type pointer is required.

For example, creating memory for N floats, where N is the input value.

```
float *p;
scanf("%d", &N);
p=(float*)malloc( N*sizeof(float) );
```



Demo program for accepting 'N' integers and printing

```
#include<alloc.h>
Int 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 code, malloc() creates memory for n integers, which can later be accessed through the pointer p. Here, the input size can be anything, making memory usage more efficient.

calloc()

It is similar to malloc(), but after allocating memory, it clears any garbage values by filling all bytes with zeros. It is also like malloc(), but after allocation of memory, it clears the garbage values by filling with zeros in all bytes of memory.

Syntax is: **pointer = calloc (number of items , size of item);**

calloc() takes two arguments: the first is the number of items to be allocated, and the second is the size of each item. For example, allocating memory for 5 floats

```
float *p;
p = (float *) calloc(5, sizeof(float));
```

realloc()

The realloc() function is used to resize (reduce or expand) memory that has already been allocated using malloc(), calloc(), or realloc(). It takes a previously allocated memory address and adjusts it to the new size.

```
syntax: pointer = realloc( oldAddress, newSize);
int *p,
p=(int*)malloc(10*sizeof(int));
---
p=(int*)realloc(p , 20*sizeof(int)); // increasing size 10 to 20 integers ( extra 10 integer's space is added)
---
p=(int*)realloc(p , 5*sizeof(int)); // reducing size 20 integers to 5
---
p=(int*)realloc(NULL, 10*sizeof(int)) // here old address is null, so it creates fresh memory like malloc() function.
```

While expanding the size of old memory, if the required amount of memory is not available in the adjacent locations, realloc() allocates a new block of memory at a different location, copies the old contents into it, and then releases the old memory. If sufficient memory is not available, it returns NULL.

free()

It is used to release memory that is no longer required by the program. This memory should not belong to statically allocated storage such as **int a[10]**; it must be dynamically allocated using **malloc()**, **calloc()**, or **realloc()**. The **free()** function takes the base address of the allocated 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);
```

Pros and Cons of DMA

A program's data area is divided into stack and heap areas. In the stack, temporary local variables are stored (auto storage class), whereas in the heap, dynamic variables and others are stored. For the Dynamic Memory Allocation (DMA) system, a special block of memory is created in RAM and managed as the heap.

In the static allocation system, the size of an array must be given at coding time, like **int a[100]**, and it cannot be changed at runtime. The static allocation system is faster than the dynamic system because memory allocation on the stack is faster than on the heap.

In the static system, the relative memory address is given to all variables at compile time, which makes it faster. In contrast, with the DMA system, memory is allocated at runtime by searching for a suitable location and maintaining a separate table to register which locations are allocated and which are not. This makes the program slower.

If the size is known, then static allocation is the better choice; otherwise, dynamic allocation is preferable.

Program accepts N strings (country names) and prints them

This program accepts strings one by one until the user enters the termination string "end". It then displays all the input strings on the screen.

```
#include<stdio.h>  
#include<string.h>  
#include<alloc.h>  
int main()  
{     char **p=NULL, a[50];  
      int count=0,i;  
      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

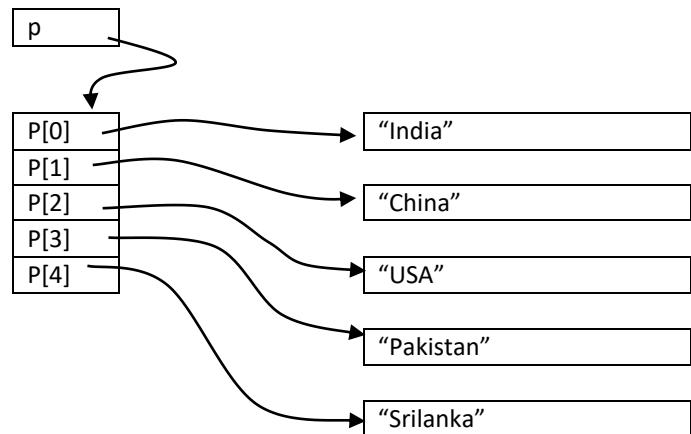
India↙

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 just by 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")

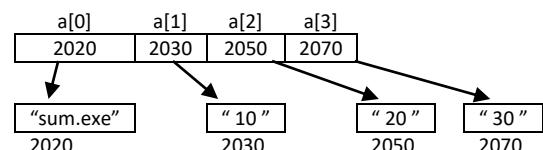
When a program is run, the operating system loads .exe file into memory and invokes the startup function main(). Here, we can assume the OS is the caller of main() function. Therefore, the arguments can be passed to the main() by specifying a list of values at the command line as shown below.

C:\tc\bin>sum 10 20 30

Unlike other functions, the arguments to main() passes in different manner. They pass as array of string constants. That means, even if you give an integer values, they pass as string. In the above case also, the arguments will be passed as "sum", "10", "20", "30". (Passes as array of string addresses as shown in below).

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

```
int 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` (this is main() function call)

Output: `sum of numbers = 100`

```
int 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. (file name of our c program is "show.c")

Input: `C:\tc\bin>show Apple`

Output: `length of Apple = 5`

```
int main( int count, char *a[ ] )
{
    if( count==1)
    {
        printf("insufficient arguments");
        return;
    }
    printf("the length of %s = %d", a[1], strlen(a[1]) );
}
```

User Defined Data Types

Structure is a user-defined, collection-type data type where several items are packed together as a single unit.

All collections can be of the same type or different types, but most of the time, they are of different types.

Therefore, we often say that an array is a collection of homogeneous items, whereas a structure is a collection of heterogeneous items. Before going in depth, let us briefly understand this with an example:

```
struct MyDate           // this is a structure declaration
{
    int day;           // here 'struct' is a keyword and 'MyDate' is a structure-name
    int month;          // here day, month, year are structure members
    int year;
};

int main()
{
    struct MyDate K, S; // declaring K,S as variables of structure type, creates space as shown below pic
    K.day = 10;          // filling values to all members (day, month, year)
    K.month = 5;
    K.year = 2000;

    S.day = 20;
    S.month = 2;
    S.year = 2021;
    ----
    printf("\n %d %d %d ", K.day, K.month, K.year); // output: 10 5 2000
    printf("\n %d %d %d ", S.day, S.month, S.year); // output: 20 2 2021
}
```

K			S		
day	month	year	day	month	year
10	5	2000	20	2	2021

In the above program, the declaration of **struct MyDate{...};** defines a new data type of user-type and does not occupy any space in the machine code file. Instead, it serves as a blueprint. Based on this definition, the variables **K** and **S** are declared in the program, and we can store and retrieve values through these variables.

Let us see each parameter at the structure declaration

```
struct MyDate ..... > keyword
{
    int day;           > identifier ( structure-name )
    int month;          > member names ( day, month, year)
    int year;
}; ..... > semicolon(;) is required for termination

int main() ..... > data-type ( 'struct MyDate' serves as the data-type-name )
{
    struct MyDate K, S; ..... > variables ( here K , S are variables of structure )
    -----
    -----
}
```

The declaration of structure variables is the same as that of normal variables.

for example, **struct MyDate a, b;** (this is same as: **int a, b;**)

here 'struct MyDate' serves as data-type-name whereas K,S are variables. Let us see one by one

Data types can be classified into two categories: 1) predefined types 2) user-defined 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 are already designed and implemented at the compiler level during the development of C. Therefore, we can directly use them in our programs.

User-Defined Data Types: These are custom types created by the programmer. C provides the freedom to define types according to program requirements; hence, they are called user-defined types.

There are three kinds of user-defined types: 1. Structure 2. Union 3. Enumerator

1) Structures

Generally, it is difficult to handle large collections of data using only basic types. With arrays, we can manage collections of homogeneous items of a known size. However, when we need to handle collections of heterogeneous items, we use structures.

For example, consider employee data containing an ID, name, address, salary, etc. Handling such heterogeneous items individually is difficult and can lead to complex code. If we group all these items into a single unit (as one record), they can be managed more easily. Fortunately, C provides the concept of structures for this purpose.

A structure is a user-defined, collection-type data type in which several items are packed together as a single unit. These items may be of the same type or different types, but most of the time, they are of different types. Therefore, we often say that an array is a collection of homogeneous items, whereas a structure is a collection of heterogeneous items.

Syntax to declare a structure:

```
struct structure-name
{
    data_member1;
    data_member2;
    ...
    data_memberN;
};
```

for example:

```
struct student
{
    int idno;
    char name[30];
    int marks1, marks2;
};
```

The declaration of the above structure defines or creates a new data type called '**struct student**'.

Using this type, we can create variables and manage their data.

Syntax to declare structure variables: **struct structure-name variable1, variable2, ;**

for example: **struct MyDate K, S ;**

The memory allocation for the K and S will be as:

K				S			
Member's name →	Idno	name	marks1	marks2	idno	name	marks1
Bytes occupied →	2	30	2	2	2	30	2

Accessing members in a structure:

we have two operators to access members of structures

1. selection operator, dot (.) // used to access members of a structure variable directly.
2. pointer-selection operator, arrow (\rightarrow) // used to access members of a structure through a pointer indirectly.

a) accessing members using selection operator

syntax: structure_variable . member_name

for eg: the expression **K.idno** accesses the 'idno' of K

the expression **K.name** accesses the 'name' of K

the expression **K.marks1** accesses the 'marks1' of K

note: the expression 'K' accesses the total memory of structure. (all member at once)

```
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

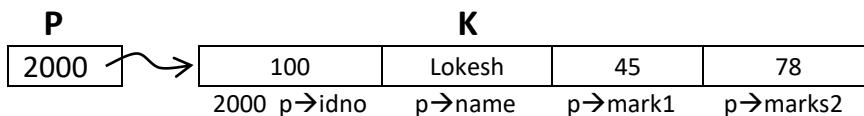
b) accessing members using pointer-selection operator

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 the 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** // “**p->idno**” is a simplified form of **(*p).idno**, this is called as: **p arrow idno**

(*p).name or **p->name** is equal to 'K.name'

The arrow operator (\rightarrow) introduced in second version of C language

arrow is a short-cut operator, used to access the members in a structure through a pointer.

Remember: **(*p).name** is equal to **p->name** , that is, **(*p)** can be taken as **p->**

Note: The expression **(*p).name** cannot be written as ***p.name** because the dot operator (.) has higher precedence than the pointer operator (*). Here, the compiler first tries to evaluate **p.name**, which results in an error because **p** does not have members of a structure; it actually contains an address. The expression **(*p).name** is valid because it is equivalent to **K.name**. (because ***p** is **K**)

Initialization vs assignment of structure variable

Assignment of structure variable

```
struct student K;
K.idno = 100;
strcpy(K.name, "Lokesh");
K.marks1 = 45;
K.marks2 = 78;
```

Assignment of structure members is the same as assigning values to basic data-type variables.

Initialization of structure variable

```
struct student K={ 100 , "Lokesh" , 45 , 78 },
```

Initialization of structure is same as initialization of array, here all values must be put in pair of braces { }

Demo program filling and displaying structure member values

Let student contain values like idno, name, and two subject marks, as in the example above, and print all the details. (Here, each member is assigned a value individually instead of initializing the whole structure at once.)

```
struct student // this is global declaration of a structure
{
    int idno;
    char name[30];
    int m1, m2;
};

int 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.idno, s1.name, s1.m1, s1.m2 );
}
```

The following program accepts student details from keyboard and printing on the screen.

```
int main()
{
    struct student k; //declaration of student variable
    printf("\nEnter student details:");
    // scanning details
    printf "\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, structure members are scanned individually according to their built-in type. The entire structure cannot be read as a single value unless a special function is written to handle it.

Filling and displaying student details through functions

The following two functions, fill() and show(), manipulate the student data.

The fill() fn fills the student details with the data: 100, "Lokesh", 45, 78, while the show() fn displays them.

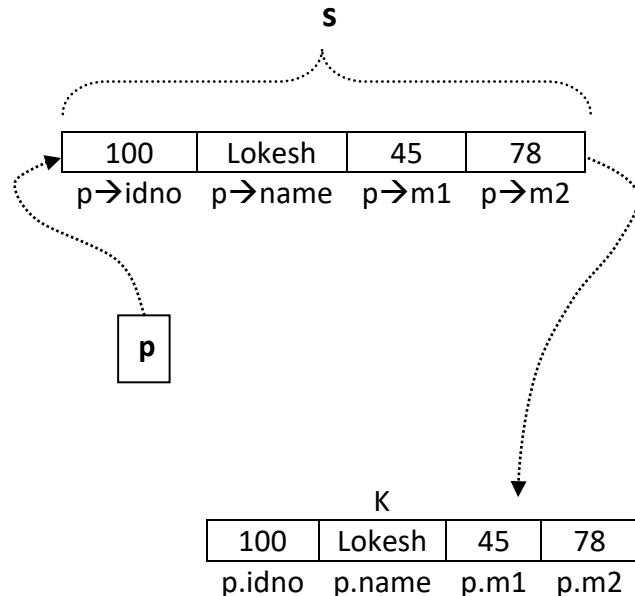
The fill() fn uses call-by-reference, whereas the show() function uses call-by-value.

```
struct student
{
    int idno;
    char name[30];
    int m1, m2;
};

int 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 fill() function uses the call-by-reference method. Here, the pointer p points to the structure s (as shown in the figure). The expressions p->id, p->name, and p->m1 access the memory of the structure 's'.

In other words, the fill() function indirectly assigns values to the members of 's'.

On the other hand, the show() function uses the call-by-value method. When it is called, all the member data of 's' are copied into a new structure 'k', which is then displayed one by one using 'k'.

Scanning two dates from the KB and finding whether they are equal or not?

* trying with & without using functions

ip: 12 9 2010

12 9 2010

op: equal

ip: 12 9 2010

13 9 2011

op: not equal

```
struct Date
{
    int d, m, y;
};

int main()
{
    struct Date a , b;  int k;
    scan( &a );           // scan date1 (day, month, year) using scan() function.
    scan( &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 scan( 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 `scan()` function uses call-by-reference because the values scanned in the `scan()` function need to be returned to the `main()` function. Therefore, they are stored indirectly through pointers.

On the other hand, the `compare()` function uses call-by-value, as it only compares the two given dates without modifying them.

We know that if data needs to be updated or modified, call-by-reference is used; otherwise, call-by-value is used.

Scanning two times from the 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 ;
};

int main()
{
    struct Time a, b, c;
    a=scan();           // this scan() fn returns the scanned time ( it is opposite to above scan() fn )
    b=scan();
    c=add(a,b);        // add two times like c=a+b
    print(c);          // printing 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 print( 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

```
int main()
{
    struct Time a, b, c;
    scan(&a);
    scan(&b);
    add(&c, &a, &b);
    print(&c);
}

void scan( 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 print( struct Time *p)
{
    printf("output time is : %d - %d - %d", p->h, p->m, p->s);
}
```

The `scan()` function should be written using call-by-reference, but `add()` and `print()` can use either method. If the structure size is less than 20 bytes, call-by-value is preferred, as the members can be easily accessed. If the structure size is greater than 20 bytes, call-by-reference is preferred, because creating a second copy of a large structure in the called function is unnecessary. In this case, for the `add()` and `print()` functions, call-by-value is the best choice because the size of the structure is less than 20 bytes.

Finding big and small of three input values.

Complete the code without modifying existing code

```
struct BigSmall
{
    int a, b, c;
    int big, small;
};

int main()
{
    struct BigSmall k;
    accept ( &k );
    find ( &k );
    show ( k );
}

void accept ( struct BigSmall *p) { ... } // here scan three values into a,b,c
void find ( struct BigSmall *p) { ... } // here find and small of a,b,c
void show ( struct BigSmall s ) { ... } // print big and small values.
```

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
};

int 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
}

```

Let us see some advantage of structures

- 1) taking two students data variables with and without structures

```
int 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

```
int main()
{
    // declaration of two student's variables with structure.
    struct student s1, s2;
    ---
}
```

This seems to be so easier than the above.
Thus structures provide a great convenience.

- 2) When we swap two structure variables, all their members are automatically swapped, so there is no need to swap each member individually.

- 3) When we pass a structure variable to a function, all its members are passed together.

- 4) Structures reduce the complexity of handling variables, making it easier to manage large collections of data.

- 5) Structure vs Its Variable

Many people say that the declaration of a structure works like a building plan on paper, whereas the declaration of its variable works like the constructed building of that plan.

Thus, the building plan is considered logical, while the constructed building is considered physical.

(Logical vs Physical → Idea vs Implementation)

```
struct student
{
    int idno;
    char name[30];
    int marks1, marks2;
};

int main()
{
    struct student K, S; // here K,S works as two buildings of above structure-plan.
    ----- // here space creates for idno, name, mark1, marks2 in K and S.
    -----
}
```

// this declaration works as building-plan

// here space will not be created for idno, name, marks1, marks2

the structure declaration works as a blueprint or template of our type and does not occupy any space in the executable file. It only provides the compiler with the layout information—i.e., the structure name, the data type of each member, and the space required for each member.

Using this structure template, we can create variables whose data can then be managed. The structure name is the identifier given to the structure and must follow the rules of variable naming.

A structure can be declared in the local or global scope, depending on the requirements of the program.
Local scope means: declaring a structure inside a function as a local variable.

Array of structures

An array of structures is useful when several instances of details need to be handled in a program.
We can create an array of structures just like we create arrays of 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, structure arrays are also accessed element by element.
Here, each element `s[0]`, `s[1]`, `s[2]`, etc., is a complete structure item.

Syntax to access members of a structure array: `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 using above structure 'person'

```
int main()
{
    struct Person s[3];
    int i;

    // scanning three persons details
    for( i=0; i<3; i++ )
    {
        printf("enter name & age of a person %d:", i+1 );
        fflush(stdin);
        gets( &s[i].name );
        scanf("%d", &s[i].age );
    }

    // printing three persons details
    for( i=0; i<3; i++ )
    {
        printf("\n %s %d ", s[i].name, s[i].age );
    }
}
```

Above program using functions,

```

int 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;
};

```

```

int 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", 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								

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++]; // copying remaining coefficients if any left in p1[]
    while(p2[j].coeff!=0) p3[k++]= p2[j++]; // copying remaining coefficients if any left in p2[]
    p3[k].coeff = 0;
}

int 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 at the global scope, then all the variables declared with it also become global.

```
struct Account
{
    int accNo;
    char name[30];
    float balance;
} a1, a2;      → Here a1, a2 are variable of type "struct Account";
```

Above example with Initialized values

```
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 a structure

```
struct          // structure name omitted
{
    int accNo;
    char name[30];
    float balance;
} a1, a2;
```

In this method, we cannot declare additional variables of the same structure later in the program, because the "structure name" is not available to refer to it in the future. This style of declaration used in nested structures.

```
struct Account
{
    int accNo;
    char name[30];
    float balance;
    struct          // called nested structure
    {
        int day,month,year;
    } openedDate;
};
```

Nested structures

A structure within another structure is called a nested structure. We can nest structures as many times as needed, and the complexity of using them does not necessarily increase even if they are deeply nested.

```
struct Student
{
    int idno;
    struct Date
    {
        int day,month,year;
    } joinedDate;
};

int main()
{
    struct Student p;
    p.idno=10;
    p.joinedDate.d=10;
    p.joinedDate.m=12;
    p.joinedDate.y=2020;
    ----
}
```

P			
Idno	joinedDate		
	day	month	year

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;**

Example: **typedef float RS;**

```
typedef float DOLLOR;
```

```
typedef int* INTP;
```

Let us see, how variables can be declared in the program

```
int main()
{
    RS amount1;          // equal to: float amount1;
    DOLLOR amount2;     // equal to: float amount2;
    -----
    -----
}
```

Here amount1 & amount2 are nothing but **float** type variables, this is equal to: **float amount1 , amount2;**

Similarly, structure types can be declared in two ways.

```
struct Date
{
    int d,m,y
};

typedef struct Date MyDATE; //now MyDATE 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;
} MyDATE; // this MyDATE not a variable, it is alias-name of struct Date
```

The above structure can also be declared in simple way as

```
typedef struct
{
    int d,m,y;
} MyDATE;
```

Advantages of a structure

1. Easy access to members

Using a structure variable, it is easier to access each member compared to handling them as individual variables. This provides more clarity and makes the code easier to manage.

2. Assignment between structures is possible

The compiler allows copying one structure to another of the same type, just like assigning integers.

All members are copied bit by bit.

```
struct Date X={10, 4, 2000};  
struct Date Y;  
Y=X; // all members of X are copied to Y (this 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 at once in single call  
swap (&e1, &e2); // swapping members
```

4. 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.

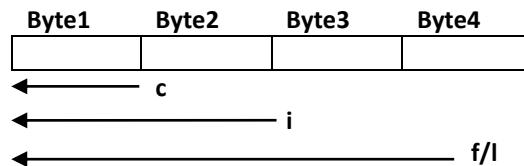
6. We can create array of structures

Unions

- 1) A **union** is also a user-defined, collection-type data type like a structure.
- 2) In a **structure**, a separate memory space is created for each member.
- 3) In a **union**, a single memory space is created, which is shared by all members.
- 4) Imagine four people staying in a house:
 - 1) In the structure case, each person has their own laptop and can use it independently at the same time.
 - 2) In the union case, there is only one laptop, and it is shared by all people at different times.
 - 3) This second situation represents a union, where the same resource (memory space) is shared by all members, but only one can be used at a time.
- 5) A union defines a **generic data type** instead of a specific one. It allows storing **different types of values** in the same memory location, but only one at a time. Remember here, all members are of **different types**.
- 6) The size of a union is equal to the size of its **largest member**.
- 7) The main purpose of a union is to save memory space when members are required to be handled one at a time (instead of all at the same time).

```
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.

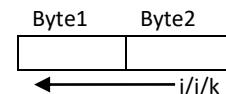
Here only one item can be stored & retrieved at one moment.

Demo program showing how the data is shared among all members of a union.

```
union myType
{
    int i, j, k;
};

int 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. Let us have one more example

```
int 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);
}
```

Output: 30 30 30. // As sharing a common space, the last value replaces all previous values.

Let us see a practical example:

the following print function can print all types of values based on passed argument values.

```
#define CHAR 1
#define INT 2
#define FLOAT 1
typedef struct
{
    union
    {
        char cc;
        int ii;
        float ff;
    }value;
    char valueType;
}MyType;

Int main()
{
    MyType var;
    var.value.cc='A';
    var.valueType=CHAR;
    print(var);

    var.value.ii=100;
    var.valueType=INT;
    print(var);

    var.value.ff=300.45;
    var.valueType=FLOAT;
    print(var);
}

void print(MyType var)
{
    switch(var.valueType)
    {
        case CHAR: printf("\n%c", var.value.cc);
                     break;

        case INT:   printf("\n%d", var.value.ii);
                     break;

        case FLOAT: 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, remember, and modify. Enumerator maps the values to names there by we can handle the values with names.

```
enum enumerator-name { name1 = value1 , name2 = value2, ... nameN = valueN } ;
```

Let us see one program: this program displays number of days in a given month.

```
enum months { jan=1, feb=2, mar=3, apr=4, ....};

int main()
{
    int month;
    printf(" enter month number(1-12): ");
    scanf("%d", &month);
    switch(month)
    {
        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 as

```
int 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

If values are not explicitly assigned at coding time, the compiler automatically assigns default integer values starting from 0. For example,

```
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.

Enumerators with explicit user-defined values

At the time of declaring an enumeration, we can assign explicit integer values to the names, and these values can be any integers. It is also possible for two or more names to have the same value (duplicate values).

```
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" is 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=-5, G, H }; // Here B=A+1, D=C+1, E=22, G=F+1, H=G+1
```

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

int 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 };

int main()
{
    enum Color wallColor, floorColor;
    wallColor=White;
    floorColor=Grey;
    ...
    ...
}
```

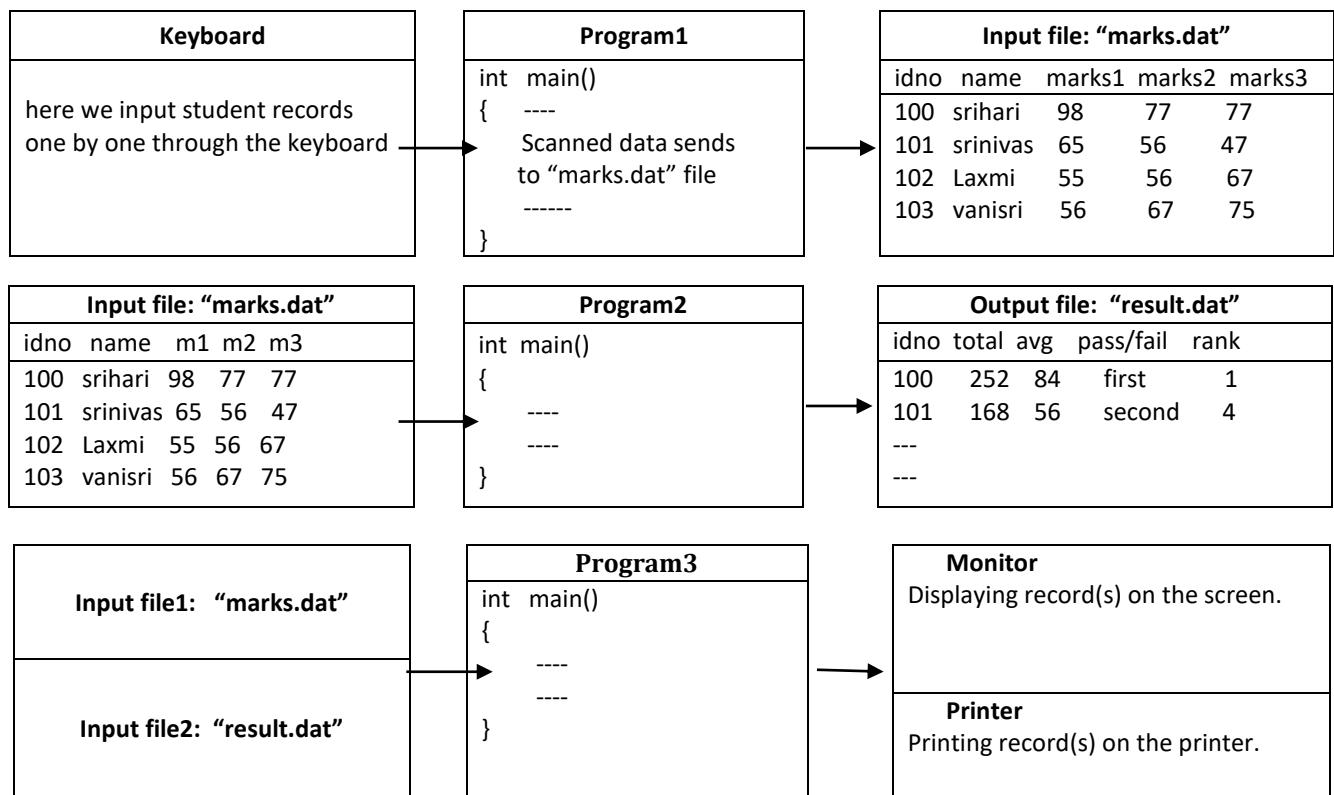
File handling in C

In real-life applications, certain data needs to be accessed many times in the present and future, such as bank transactions or business records. Therefore, such data is stored permanently in the computer so that it can be retrieved whenever required. For this purpose, secondary storage devices are used to store data permanently in the form of files. These are often called data files.

A data file can be defined as a collection of data stored permanently in a secondary storage device. Examples of secondary storage devices include hard disks, CDs, DVDs, pen drives, and magnetic tapes.

Each file is identified by a valid name, called a file name. The interaction with files is treated as interaction with I/O devices in the computer. Therefore, all secondary storage devices are considered as I/O 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 the results are stored in a separate file called "result.dat" or displayed on the screen. This depends on the specific requirement of the problem. Following picture explains how our programs interact with files.



In computer science, files are classified into two types, **1. Data files, 2. Executable program files**.

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.

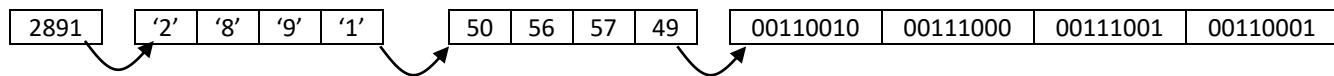
2. Executable files: for example, .exe, .com, .dll, etc; contains machine code instructions. Executable files are created using compilers, whereas data files are created using executable files. In computer, any file is stored either in **text & binary** format. All executable files are stored in binary format, whereas data files are stored in text or binary formats depending on our requirement.

Text Format Files: Data is stored in the form of ASCII values (human-readable string format).

Binary Format Files: Data is stored in the same way as it is in program variables (raw binary format, not human-readable).

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 a file. For example, the 2byte int-value 2891 is stored in text file as



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. The library functions converts this numeric to text and text to numeric while storing and reading back from files. These text 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 in the computer, by writing a special software program to manage them, or by using ready-made text editor softwares. Using text editors, we can read/write/modify the text data. We have several text editors like notepad, word pad, ms-word, etc.

In text files, the \n character is stored as two characters \r\n (Windows format), but while reading back, it is interpreted as a single character (\n). The library functions handle this automatically while writing and reading \n to and from files, so we don't need to worry about how this character is managed internally.

In text files, the value 26 is inserted as the end-of-file (EOF) mark (similar to the null character \0 for strings). Some people may wonder: if the file data itself contains the value 26 (for example, an employee's age), how is it differentiated from the EOF mark? In such cases, the number 26 is stored as the characters '2' and '6' with their ASCII values 50 and 54, so there is no conflict between the actual data and the EOF marker.

Binary files

In this kind of file organization, the data is stored exactly as it exists in program variables. Unlike text files, no conversion is performed while storing or reading the data back. Generally, binary files are extensively used for maintaining similar types of data, such as employee records, bank accounts, insurance accounts, and so on.

In binary I/O, storing and reading take place directly between RAM and the hard disk; therefore, binary files are faster than text files. At the end of text files, the value 26 is inserted as an end-of-file marker, whereas in binary files no such value is inserted. Instead, the end-of-file is determined by the file size.

In the case of binary files, we cannot manipulate the data using text editors.

For example, consider a student structure: idno | name | marks → int | char(30) | int.

If this data is dumped as it is into a file, then it must be read back in the same way (2 bytes at a time for idno, 30 bytes at a time for name, and so on). Since text editors are unaware of this user-defined structure format, they cannot read/write this data. (Text editors are designed to read/write only ASCII values, one byte at a time.)

To handle binary files, special software programs are required. For example, a '.pdf' file is a binary format file and cannot be opened in a text editor. Instead, it requires software like Adobe Reader. Similarly, '.jpg' images, sound files, and movie files are binary data, and each requires specialized software for handling.

In C, files can be handled in two ways: **High-Level File Handling and Low-Level File Handling**

In high-level file handling, C provides plenty of library functions with file streaming support. These functions not only allow reading and writing data from files but also handle tasks like converting numbers to text, text to numbers, interacting with printers, and organizing files randomly. Thus, high-level file handling is easier to use.

In contrast, low-level file handling has no such library support. Here, the programmer must write code for every task. This approach is mainly used in hardware industries for chip-level programming to develop customized software.

File Streaming or File Buffering:

The total hard disk space is divided into several blocks, where each block size may be 1024/2048 bytes. This is not a fixed size and it may vary. The contents of our files are stored in these blocks of the hard disk; if the file size is 3000 bytes then it takes two or more blocks. We can't perform insert/delete/modify operations directly on file data when it is in the hard disk, because the hard disk supports only block-by-block reading and writing. So first we load (dump) the file data from the hard disk into RAM and then perform the respective operations; later we store back the updated data to disk. RAM supports byte-by-byte read-write operations, therefore we can perform any operations when the data is in RAM.

When we open an existing file, the total file contents are not dumped into RAM; only the first block is loaded into RAM and then the file pointer is set to it. The remaining blocks are loaded one by one as the file pointer advances. The file pointer is a pointer used to read/write file contents. When the file pointer reaches the end of the first block, the second block is loaded into the same memory (the 1st block is replaced by the 2nd block). In this way, file data is loaded into RAM and operated on. This process is called file-streaming and is managed with the help of the operating system. For random file organization, the blocks are loaded & replaced randomly according to the movement of the file pointer. The OS provides this file-streaming facility to move the file data from disk to RAM and from RAM to disk.

Stream I/O = array of bytes memory + read() function + write() function + open() function + close() function + other functions. This array of bytes memory is also called a buffer or stream memory with all its functionality.

High-level file IO

File operations are mainly three

1. Opening a file
2. Applying read/write/modify operations on file data
3. Closing a file

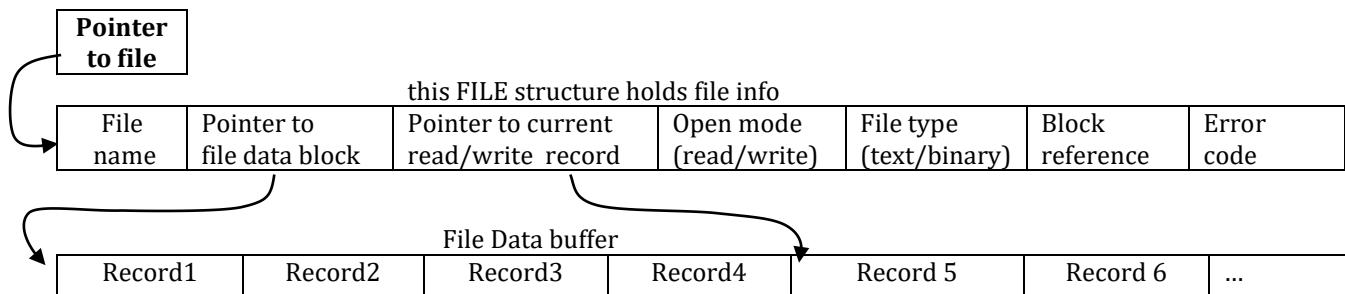
Opening a file: The function fopen() is used to open a file. When a file is opened, the file contents are dumped from the hard disk to RAM, and then the file pointer is set to it. The function prototype is:

```
FILE* fopen(char *fileName, char *openMode);
```

The first argument is the name of the file that we want to open, and the second argument is the 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 structure as given below picture



The **FILE** structure is a predefined structure defined in the stdio.h file. This structure holds extra information about the file's name, file-open-mode, file-type (binary/text), buffer size (block size), a pointer to the file data, and a pointer to the current read/write byte. This **FILE** structure is used for file streaming as above said.

The fopen() function first creates memory for the **FILE** structure and a file data buffer, then loads the file's contents from the disk into this memory. Finally, it returns the address of this **FILE** structure. If it is unable to open a file, it returns **NULL**. Let us know file open modes

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 will be 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.

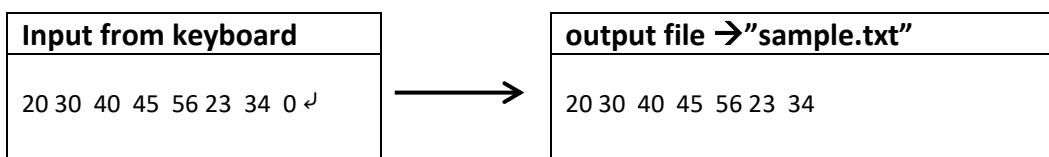
These two functions are specialized to handle text files. Using these functions, we can also perform formatted input/output on text files. Formatted means, writing data using io flags such as %d %5d %-5d %.2f %s %30s etc;

fread(), **fwrite()**: These are specialized for binary files reading/writing data. Of course, we can also use these functions 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 the keyboard until the last input is zero, and writes each scanned value to the disk (file).



```
#include<stdio.h>
int main()
{
    FILE *fp;                                // pointer to a 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' value to file
    }
    fclose( fp );   // saving file contents on disk
}
```

Reading integers from a file

This is opposite to above program, reading integer from a file and showing on the screen.

```
#include<stdio.h>
int 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; // stops when file pointer is reached end-of-file (EOF value is -1)
        printf("%d ", n);          // displaying scanned on the screen
    }
    fclose(fp); // the file content will not be saved on disk, because, file is opened in read-only mode( no updatioins),
}                                         // so this function only clears from RAM.
```

Above program in binary i/o format

```
#include<stdio.h>
int 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() functions
    }
    fclose(fp); // saves the 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 the screen which has read from the file
    }
    fclose(fp); // will not be saved on disk, because opened in read-only mode, only clears from RAM.
}
```

fwrite() function syntax:

fwrite(&variable, sizeof(variable), number of items, file-pointer)

&variable → this gives the address of the variable where the data is exist

sizeof(variable) → this gives the size of data to be written to file from the given variable address .

no.of variable items → in case array of items exist.

file-pointer → pointer to file where the data to be written

fread() function syntax:

fread(&variable, sizeof(variable), number of items, file-pointer)

&variable → this gives the address of a variable where the data to be inserted

sizeof(variable) → this gives the size of data to be read from a given file.

no.of variable items → in case array of items exist.

file-pointer → pointer to file where the data to be read.

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 write only **odd** numbers into another file called “**output.txt**”.

file name: “input.txt”	→	File name: “output.txt”
12 17 20 13 29 30 32 35 40 43 27 20 21 29 31 39 11 12 10 8 2 3		17 13 29 35 43 27 21 29 31 39 11 3

```
#include<stdio.h>
int main()
{
    FILE *fs, *ft; int n, k;
    char sName[30], dName[30];
    printf("enter source file name :");
    gets(sName);
    printf("enter destination file name :");
    fflush(stdin); 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 \n", n);
    }
    fclose(fs); fclose(ft);
}
```

Note: The file input.txt must be present in the current working directory. If the file is located elsewhere, you must provide the full path while opening it. For example, if the file is on the Desktop in Windows OS, you should open it as: fp = fopen("C:\\Users\\<username>\\Desktop\\input.txt" , "rt");
 (Here replace <username> with your actual Windows username)

More file IO functions

fputc(): It is used to write a 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 a single character from a file and returns ASCII value of it. **ch=fgetc(file_pointer);**

fputs(): Writes a string to file, syntax is: **fputs(string , file_pointer);**

fgets(): Reads a string from file, syntax is: **fgets(array_address_to_hold, number of bytes read, file_pointer);**

fputw(): Writes an integer to file (here ‘w’ means word, word means integer in machine-code)

fgetw(): Reads an integer from a file

fseek(): 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 beginning-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 the keyboard and writes into a file called "sample.txt".

The written contents of file can be seen using any text editor like notepad.

Input from keyboard:

```
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
```

```
#include<stdio.h> // to include FILE structure and its IO functions.
int 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
}
```

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>
int 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);
}
}
```

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.

```
#include<stdio.h>
int main()
{
    FILE *fs, *ft; char ch;
    char sName[30], dName[30];
    printf("enter source file name :");
    gets(sName);
```

```

printf("enter destination file name:");
fflush(stdin);
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; // fread() returns 0 or 1 when file is reached to end-of-file
      fwrite( &ch, sizeof(ch), 1, ft );
}
fclose(fs); fclose(ft);
}

```

Handling student details (in binary file format)

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 a 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↵

 enter idno (0 to stop): 0↵

output: idno name mark1 mark2 result

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];
};
int 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
int 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 sucessfully 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);
}
```