

COMPUTER PROGRAMMING

Computer Fundamentals

Topics covered

Computers Fundamentals: Binary Number System, Computer memory, Computer Software.

Algorithms and Programming Languages: Algorithm, Flowcharts, Generation of Programming Languages.

Course Information

Tentative Grading scheme :- 25(MST);45(EST);30(Sessional)

Attendance policy :- 75% including all emergencies

Punctuality :- No entry if more than 5 mins late. You should attend the class/lab in proper dress and proper place as you can be asked to switch on the video anytime.

What is programming?

Computer programming language expresses a set of detailed instructions for a computer.

Editor and Console

The screenshot shows a development environment with two main panes. The top pane is a code editor titled "HelloWorld.c" containing the following C code:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello, World!\n");
6
7     return 0;
8 }
```

The bottom pane is a terminal window showing the execution of the program:

```
C:\Users\GHANENDRA\Desktop\Tutorials Bookmarks Logos\C Language\HelloWorld.exe
Hello, World!
-----
Process exited after 0.1954 seconds with return value 0
Press any key to continue . . .
```

C Program to Print "Hello World"

Compiler to run the program

https://www.onlinegdb.com/online_c_compiler

DevC++ (Find some other compiler for MAC)

CppDroid app for Android phones

E-box for the labs (e-box.co.in) is also for cell phones

First program

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

```
int main() {  
    printf("Hello World");  
    return 0;  
}
```

// No need to understand now, just run it

Simple integers program

```
#include <stdio.h>

int main(){
    int a = 10, b=11,c=a+b;
    printf("a = %d, b = %d, c = %d",a,b,c);
    return 0;
}
```

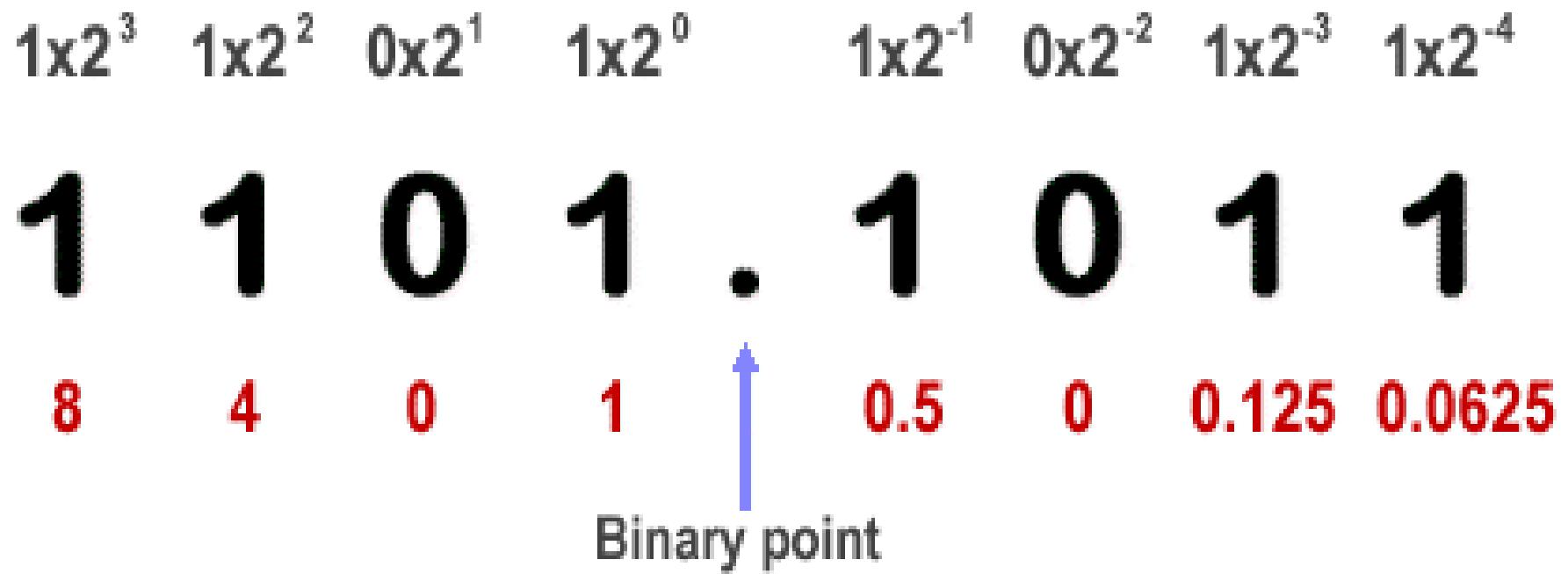
Computer only knows binary

Computer only knows 0 and 1

Any information can be encoded in binary strings

0 or 1 is called a bit, set of 8 bits is called a byte

Binary number system



$$8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 + 0.0625 = 13.6875 \text{ (Base 10)}$$

<u>Decimal</u>	<u>Binary</u>
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Negative binary numbers: 2's complement

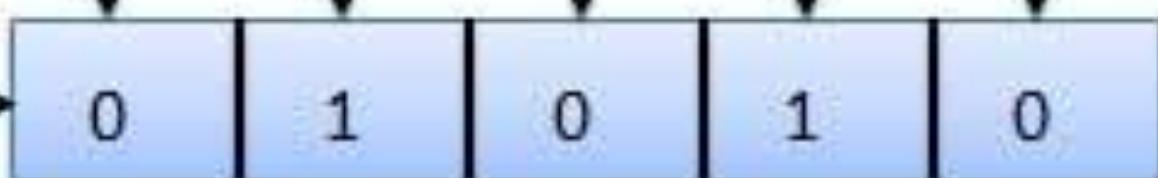
Flip zeros and ones and add 1

decade number	binary
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Given number →

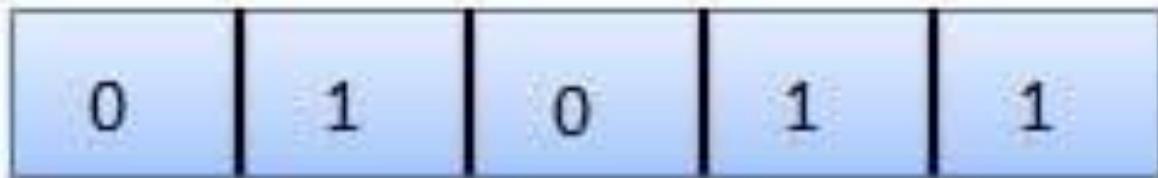


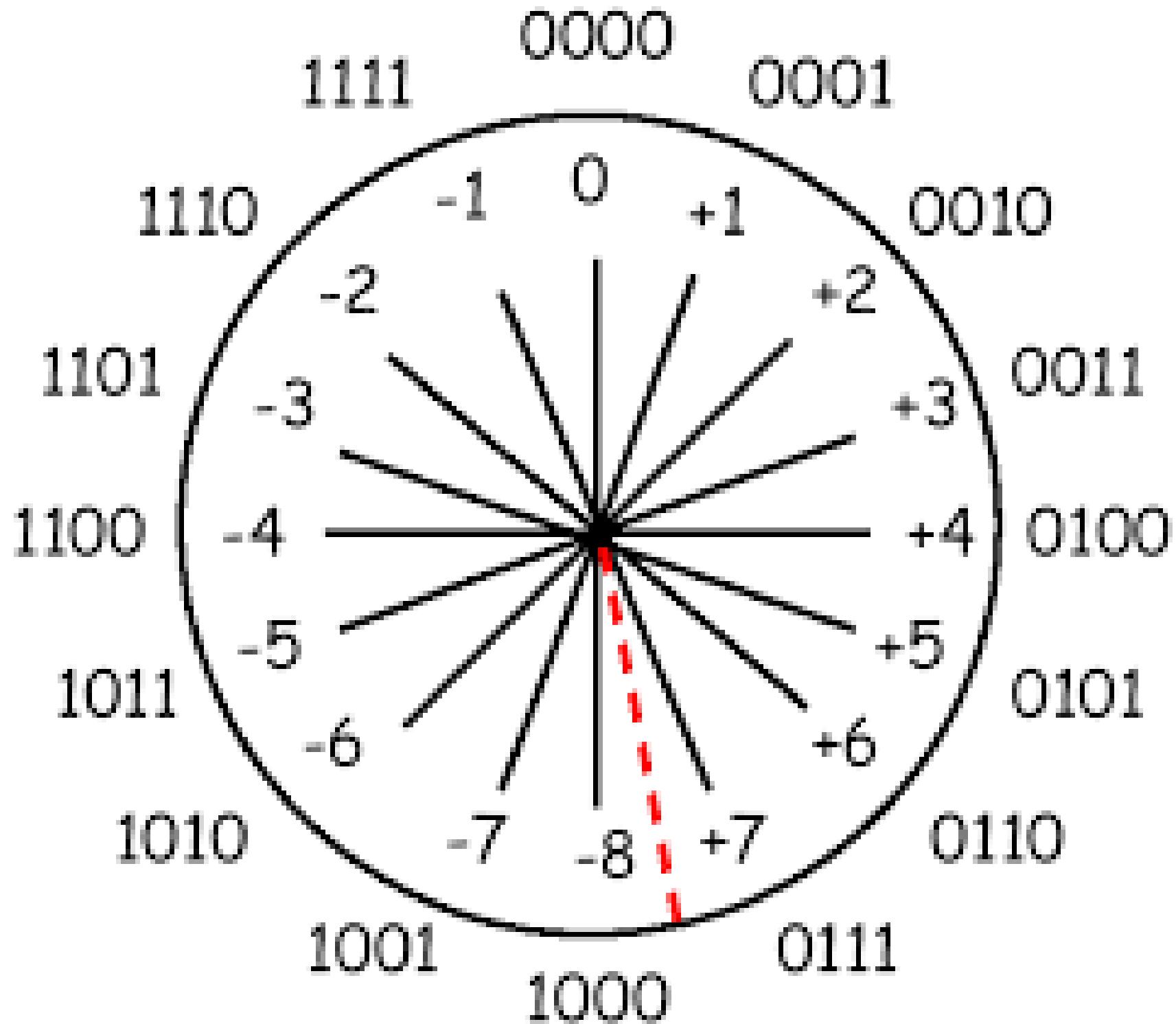
1's complement →



Add 1 +

1





1101 into unsigned and signed

Unsigned (+): So $1101 = 13$ in decimal

Signed (+/-): If the most significant bit (MSB) is 1 then number is negative. So **1101** means a negative number. Then find its 2's complement to find its value which is 0011. So **1101 = -3**.

Find *signed* decimal values for 10100101 and 01111111.

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Example: Decimal to Octal

$$670_{10} = 1236_8$$

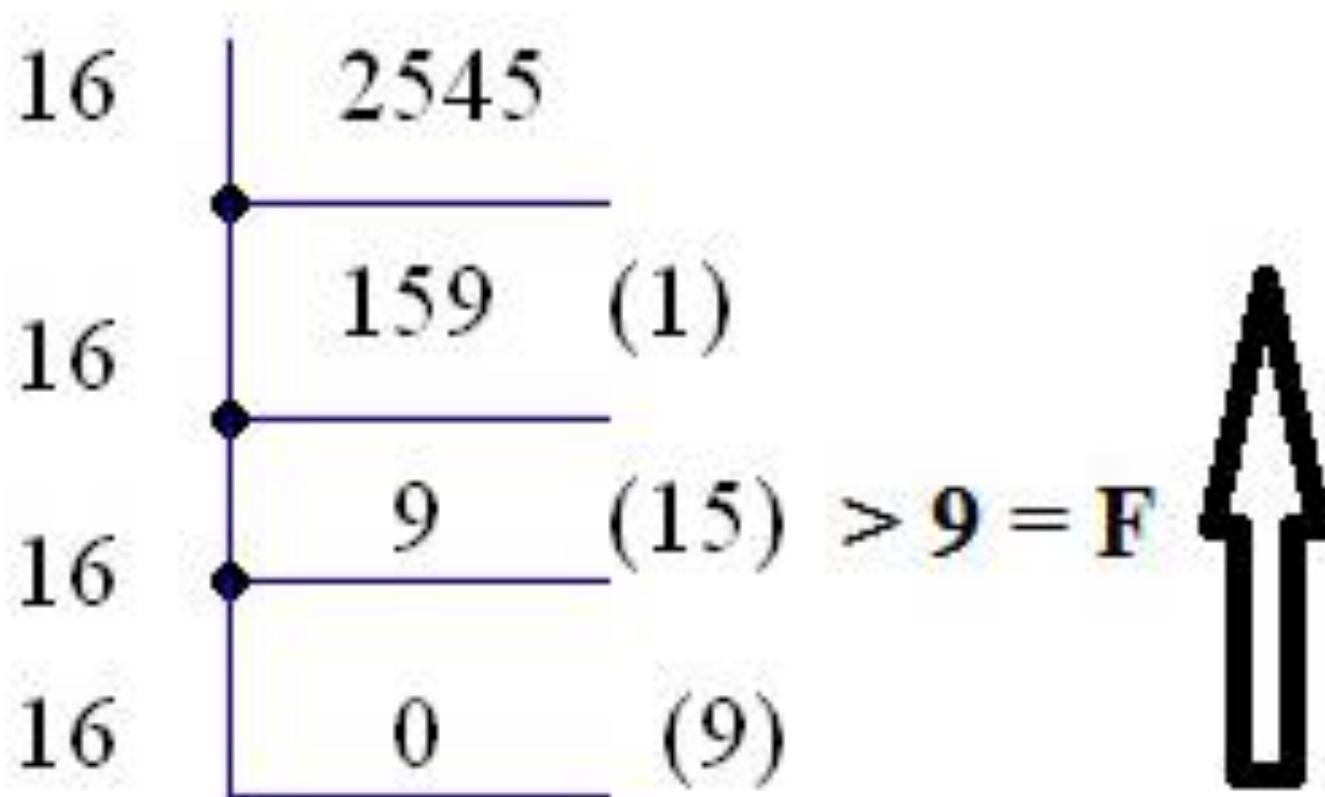
$$670 \div 8 = 83 \text{ r.}6$$

$$83 \div 8 = 10 \text{ r.}3$$

$$10 \div 8 = 1 \text{ r.}2$$

$$1 \div 8 = 0 \text{ r.}1$$

Example: Decimal to Hexadecimal



= 9F1
(Hexadecimal)

Find the Hex Equivalent for Binary 1011010

101

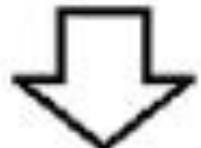
group 2

1010

group 1

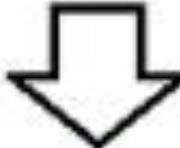
*Group 2 containing only 3 bits,
so add 0 to the left*

0101



5

1010



A

So the answer is 5A in hexadecimal

Octal to Hex Conversion

Convert the octal 752_8 to its octal equivalent.

1. Separate the digits of the given octal number, if it contains more than 1 digit.

7 5 2

2. Find the equivalent binary number for each digit of octal number. Add 0's to the left if any of the binary equivalent is shorter than 3 bits.

7 5 2

111 101 010

3. Write the all groups' binary numbers together, maintaining the same group order.

111101010

4. Separate the binary digits into groups, each containing 4 bits or digits from right to left. Add 0s to the left, if the last group contains less than 4 bits.

0001 1110 1010

5. Find the hex equivalent for each group.

0001 1110 1010
1 E A

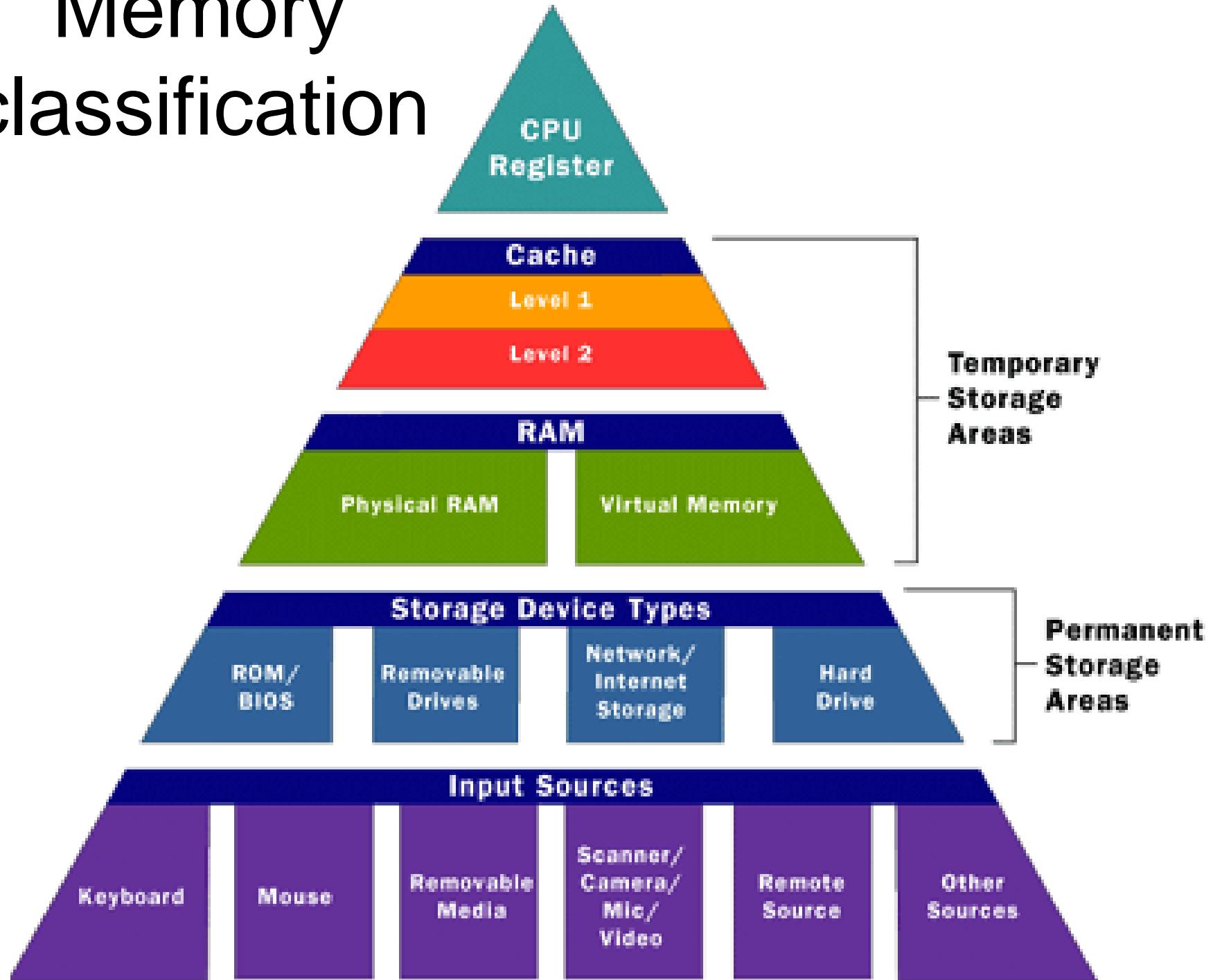
6. Write all hex equivalent of each group together where keeping the same order provides the hex equivalent for the given octal number.

1EA

Result

$$752_8 = 1EA_{16}$$

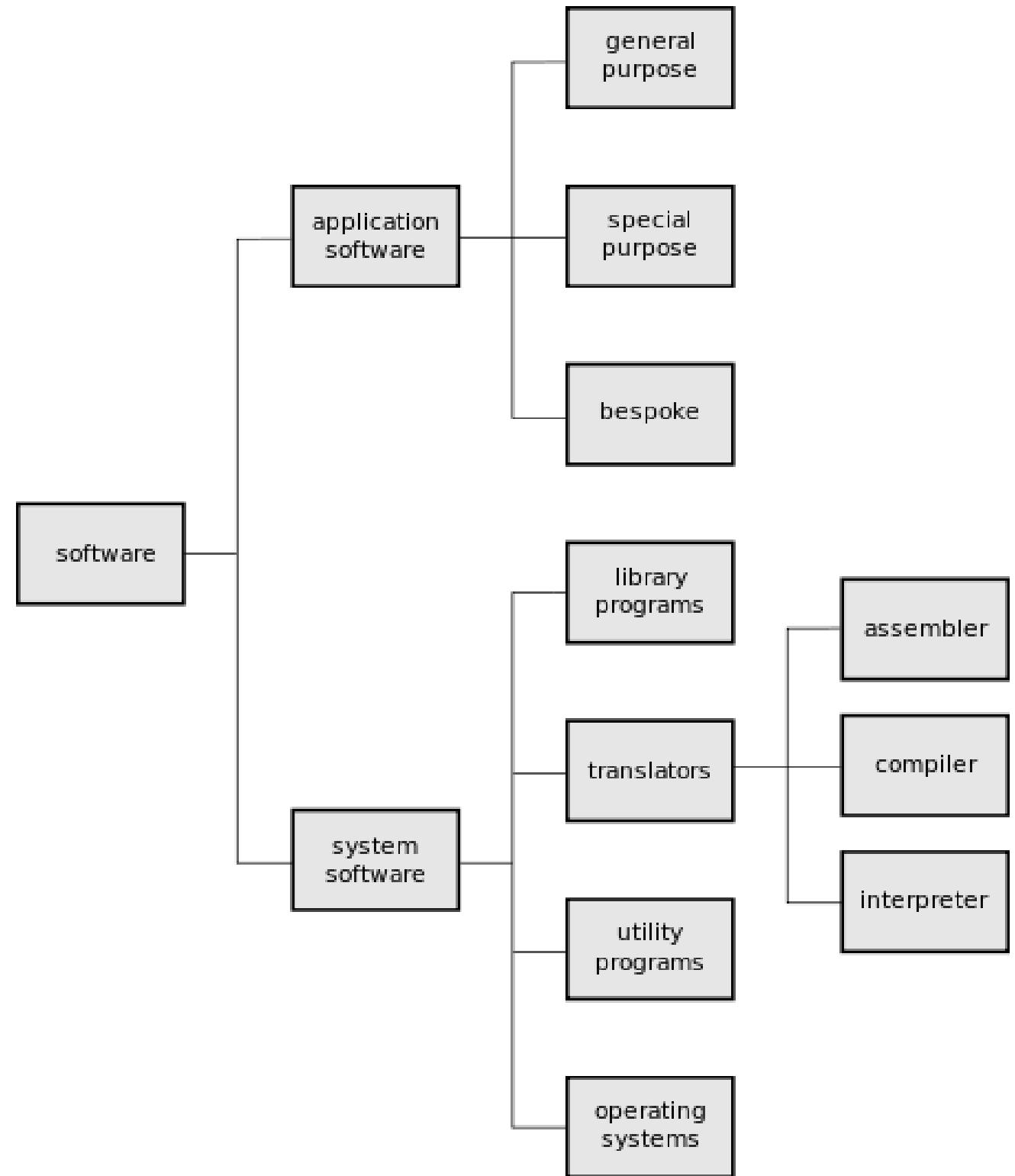
Memory classification



Memory basics

- **Registers** is small memory spaces inside processor.
- **Cache** is a high speed random access memory, integrated with the CPU.
- **Level 1** is very small (usually between 2 and 64 KB).
- **Level 2** resides on a memory card near CPU about 256 KB to 2 MB. Slower than L1 but faster than main memory.
- **Main memory** - RAM where programs and data are kept for processor
- **Virtual memory** – Part of Hard disk which is used by RAM to give an impression of larger capacity.

Software classification



Algorithms and Programming Languages:

Algorithm
Flowcharts
Generation of Programming Languages

Algorithm

Finite sequence of explicit and unambiguous instructions, which when provided with a set of input values produces an output and then terminates.

Examples Of Algorithms

Write an algorithm to add two numbers entered by user.

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum<=num1+num2

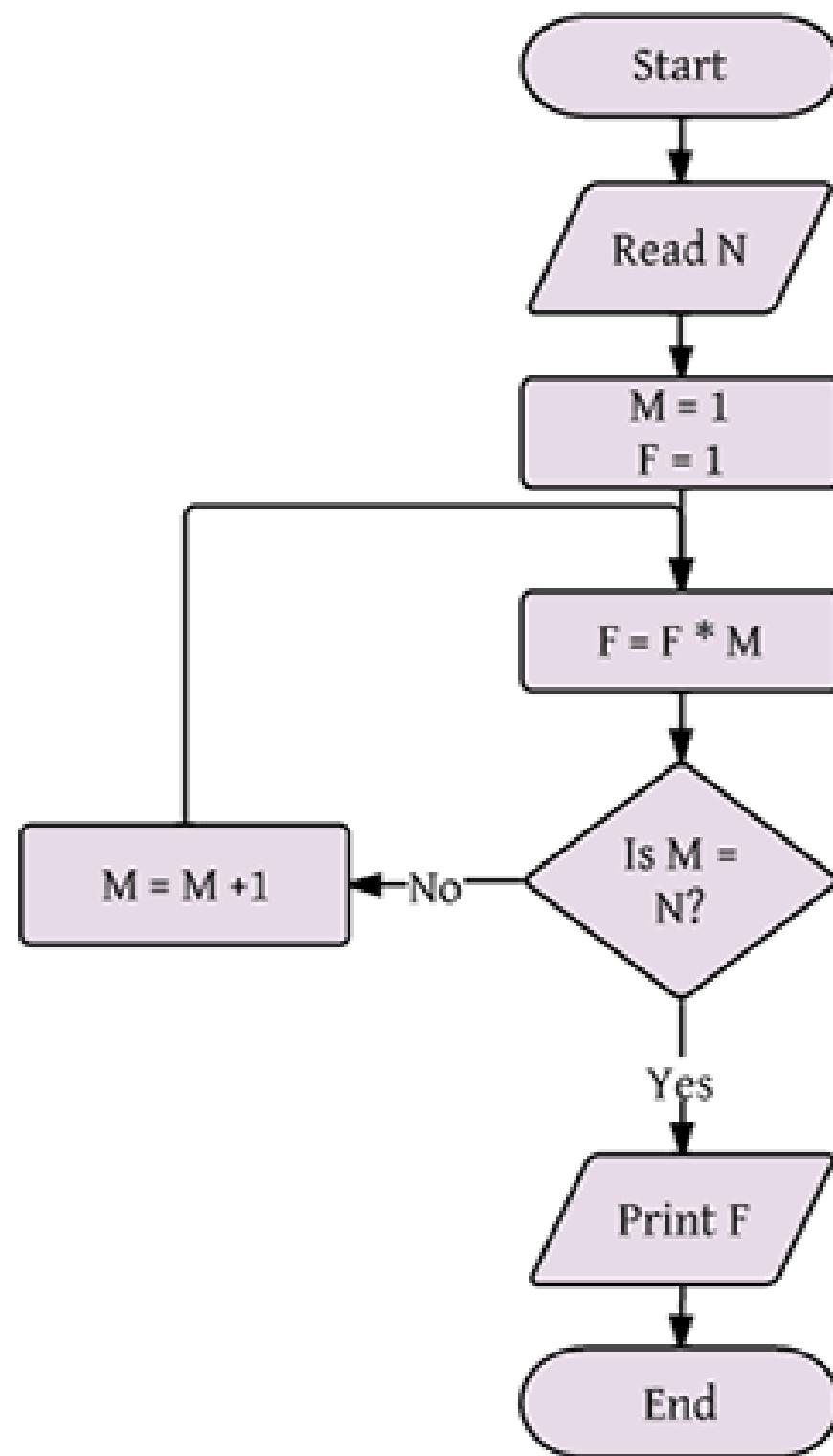
Step 5: Display sum

Step 6: Stop

Flowchart

It is a pictorial form of an algorithm.

Boxes represent operations and arrows represent sequence in which the operations are executed



Pseudo code

- (1) Pseudo code is a generic way of describing an algorithm without using any specific programming language-related notations.

- (2) It is an outline of a program, written in a form, which can easily be converted into real programming statements.

```
If within boundaries of search
    Calculate midpoint
    If value is at midpoint
        Return midpoint
    Else if value at midpoint is too large
        Look to the left
    Else if value at midpoint is too small
        Look to the right
    EndIf
EndIf
Return -1 if value wasn't found|
```

5 generations of languages

First - machine language

Second - assembly language

Third - high-level programming languages, such as C, C++, and Java.

Forth – more close to human language, e.g. SQL

FIND ALL RECORDS WHERE NAME IS "SMITH"

Fifth - languages used for artificial intelligence and neural networks.

1-4 generation examples

Generation	First	Second	Third	Fourth
Code example	<code>10101010011000101 10011010100000010 11111111101000101</code>	<code>LDA 34 ADD #1 STO 34</code>	<code>x = x + 1</code>	<code>body.top { color : red; font-style : italic }</code>
Language	(LOW) Machine Code	(LOW) Assembly Code	(HIGH) Visual Basic, C, python etc.	(HIGH) SQL, CSS, Haskell etc.
Relation to Object Code (generally)	--	one to one	one to many	one to many

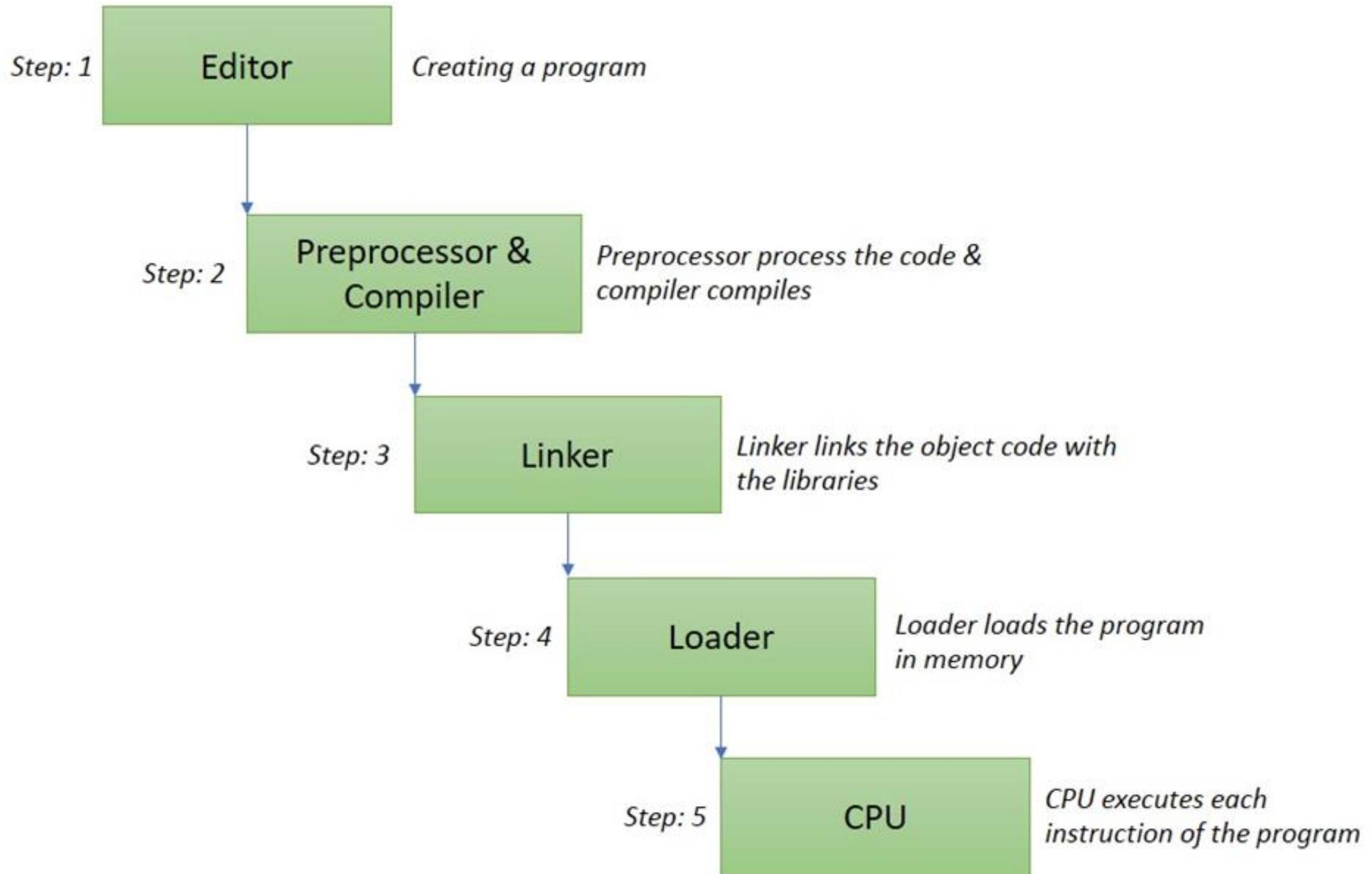
Compiler, assembler, interpreter

Compilers are used to convert high level languages (like C, C++) into machine code. **Example - GCC**

Assembler are used to convert assembly language code into machine code.

Examples - X86 assemblers

An interpreter is a computer program which executes a statement directly at runtime **Examples: Python**



Program life cycle

Basic definitions of computer malware (malicious software)

Adware (short for advertising-supported software)

Bots automatically perform specific operations such as video gamings, online contests

Bugs are human errors in programming

Ransomware - displaying messages to demand money while locking the computer programs

Rootkit – malicious remote access to computer

Spyware – peeping Tom

Trojan horse – password stealer

Worm – exploits host computer resources

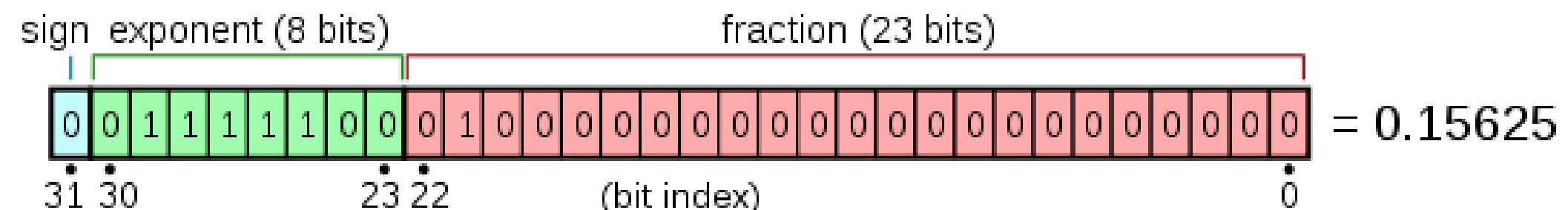
Spam – Mass of unsolicited (unwanted) emails

IEEE-754 standard

32-bit floating-point

Decimal numbers in computer memory

Basic idea – sign bit, exponent, fraction



How decimal numbers are stored in binary (4 bytes)?

Consider 5.2

5->101 and 0.2 -> .00110011... (record carry for
0.2x2=0.4, 0.4x2=0.8, 0.8x2=1.6, 0.6x2=1.2)

So 5.2 = 101.00110011... = 1.010011...E+2

Conventional trick $127+2=129=10000001$

0 10000001 0100110011001100110

SignBit 8-bit Expo 23-bits mantisa

Example 1: Suppose that IEEE-754 32-bit floating-point representation pattern is 0
10000000 110 0000 0000 0000 0000

Sign bit $S = 0 \Rightarrow$ positive number
 $E = 1000\ 0000B = 128D$ (in
normalized form)

Fraction is $1.11B$ (with an
implicit leading 1) =

$$1 + 1 \times 2^{-1} + 1 \times 2^{-2} = 1.75D$$

The number is $+1.75 \times 2^{(128-127)}$
= $+3.5D$

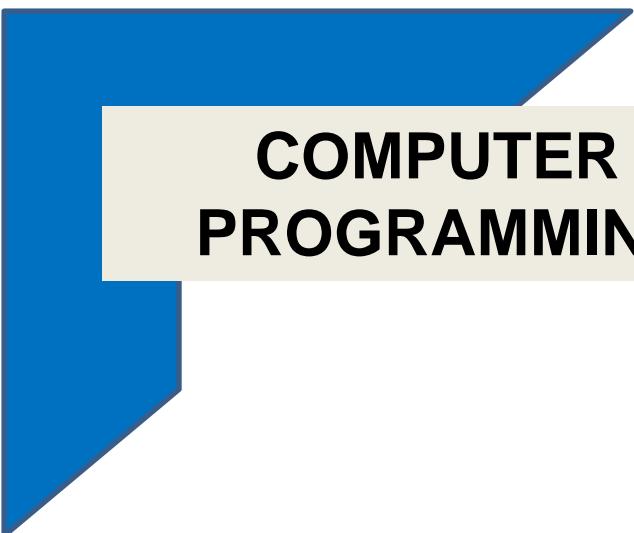
Exercise for floating point representation

- (1) Convert -7.5D into 754 IEEE format.
- (2) Convert IEEE-754 32-bit floating-point representation into floating point decimal number:
10101010 10101010 10101010 10101010

Check list – Do you know these?

Binary Number System, Computer
memory, Computer Software.

Algorithm, Flowcharts, Generation of
Programming Languages.



COMPUTER PROGRAMMING



Basics about C program

Topics covered

- Structure of C Program
- Life Cycle of Program from Source code to Executable
- Compiling and Executing C Code
- Keywords
- Identifiers
- Primitive Data types in C
- Variables
- Constants
- Input/Output statements in C
- Operators
- Type conversion and type casting

Structure of a C program

A C program is divided into different sections. There are six main sections to a basic c program :

1. Documentation/ Comments Section
2. Preprocessor Directives/ Link Section
3. Definition Section
4. Global Declaration Section
5. Main Function
6. Sub-program Section

BASIC STRUCTURE OF A 'C' PROGRAM:

Documentation section [Used for Comments]
Link Section - Header files, preprocessing definition
Definition Section
Global declaration section [Variable used in more than one function]
main() { Declaration part Executable part }
Subprogram section [User-defined Function] Function1 Function 2 : Function n

Example:

```
//Sample Prog Created by:Bsource  
#include<stdio.h>  
#include<conio.h>  
#define PI 3.14  
int a=10;  
void main()  
{  
printf("a value inside main(): %d",a);  
fun();  
}  
void fun()  
{  
printf("\n a value inside fun(): %d",a);  
}
```

1. Document/ Comments Section

- The documentation section is the part of the program where the programmer gives the details associated with the program.
- He usually gives- Name of the program,
 - The Author And Other Details,
 - which the programmer would like to use later.
- Comment means explanations or annotations that are included in a program for documentation and clarification purpose.
- Comments are completely ignored by the compiler during compilation and have no effect on program execution.
- This section is optional

1. Documentation/ Comments Section (cont..)

- Multiline Comments starts with ‘/*’ and ends with ‘*/’
- Single line comments starts with ‘//’

Example:

```
/* Comments Section
Structure of C Program
Author: XXXX
Date : 28/07/20
*/
// My first C Program
```

2. Link Section

- This part of the code is used to declare all the header files that will be used in the program.
- This leads to the compiler being told to link the header files to the system libraries.
- **Example:**
- **#include<stdio.h>**
- These are also called preprocessor directives.
- Preprocessor Directives are always preceded with ‘#’ sign .
- These are the first statement to be checked by the compiler.
- There are many preprocessor directives but most important is
`#include <stdio.h>`

2. Link Section (cont...)

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

- Tells the compiler to include the file stdio.h during compilation
- ‘stdio.h’ stands for standard input output header file and contains function prototype for input output operations e.g. printf() for output and scanf() for input, etc.

2. Link Section (cont...)

Other popular header files:

#include<string. h> (String header)

#include<stdlib. h> (Standard library header)

#include<math. h> (Math header)

3. Definition Section

- In this section, we define different constants. The keyword `define` is used in this part.
- `#define PI 3.14`

4. Global Declaration Section

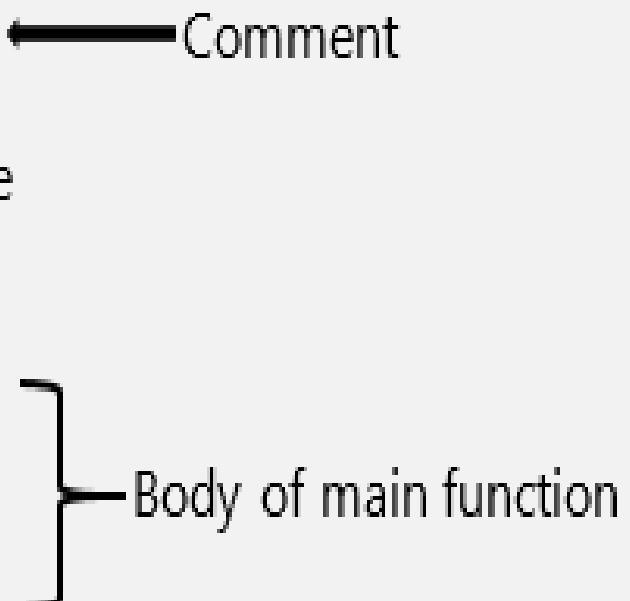
- The global variables that can be used anywhere in the program are declared in global declaration section.
- This section also declares the user defined functions.
- float area(float r);
- int a=7;

5. Main Function

- It is necessary to have one main() function section in every C program.
- This section contains two parts, declaration and executable part.
- The declaration part declares all the variables that are used in executable part.
- These two parts must be written in between the opening and closing braces.
- Each statement in the declaration and executable part must end with a semicolon (;).
- The execution of program starts at opening braces and ends at closing braces.

Example of a C Program

```
/* My first C program to print Hello, World! */ ← Comment  
#include <stdio.h> ← Pre-processor directive  
  
int main() ← Main function  
{  
    printf("Hello, World!"); ← Function  
    return 0;  
}
```



6. Sub-program Section

- All the user-defined functions are defined in this section of the program.
- User can define their own functions in this section which performs particular task as per the user requirement. So user creates this according to their needs.

Example:

```
int add(int a, int b)
{
    return a+b;
}
```

Example of a C Program

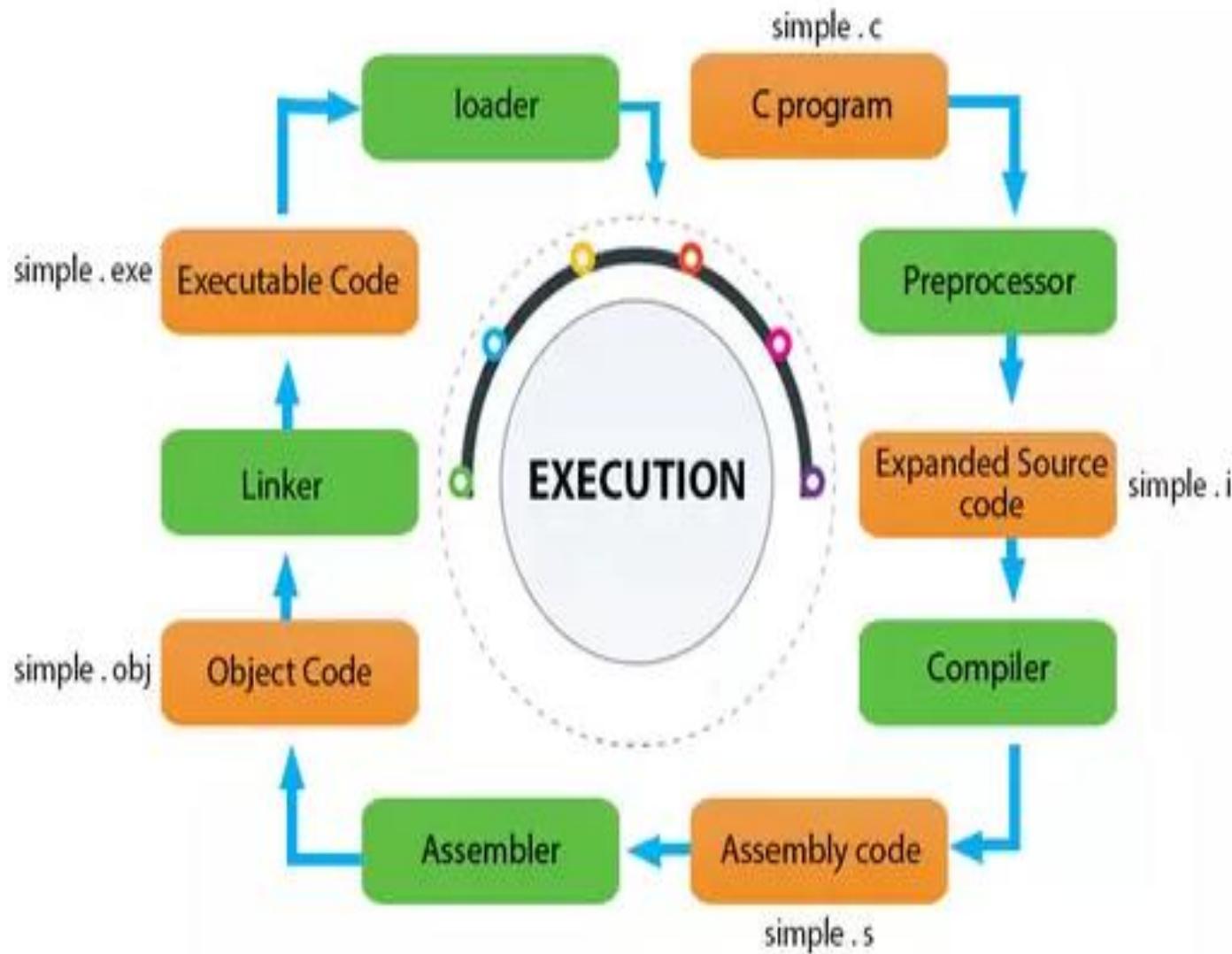
```
//File Name: areaofcircle.c
// Author:XXX
// date: 01/10/2020
//description: a program to calculate area of circle
#include<stdio.h>//link section
#define pi 3.14;//definition section
float area(float r);//global declaration
int main()//main function
{
    float r;
    printf(" Enter the radius:n");
    scanf("%f",&r);
    printf("the area is: %f",area(r));
    return 0;
}
float area(float r)
{
    return pi * r * r;//sub program
}
```

Life Cycle of a C Program

```
/* My first C program to print Hello, World! */ ← Comment  
  
#include <stdio.h> ← Pre-processor directive  
  
int main() ← Main function  
{  
    printf("Hello, World!"); ← Function  
    return 0;  
}  
} ← Body of main function
```

Let us suppose, we write above program in Notepad and save the file as simple.c

Life Cycle of a C Program



Preprocessor

Compiler

Assembler

Linker

loader

```
/* this is demo */  
#include<stdio.h>  
void main()  
{  
    printf("hello");  
}
```

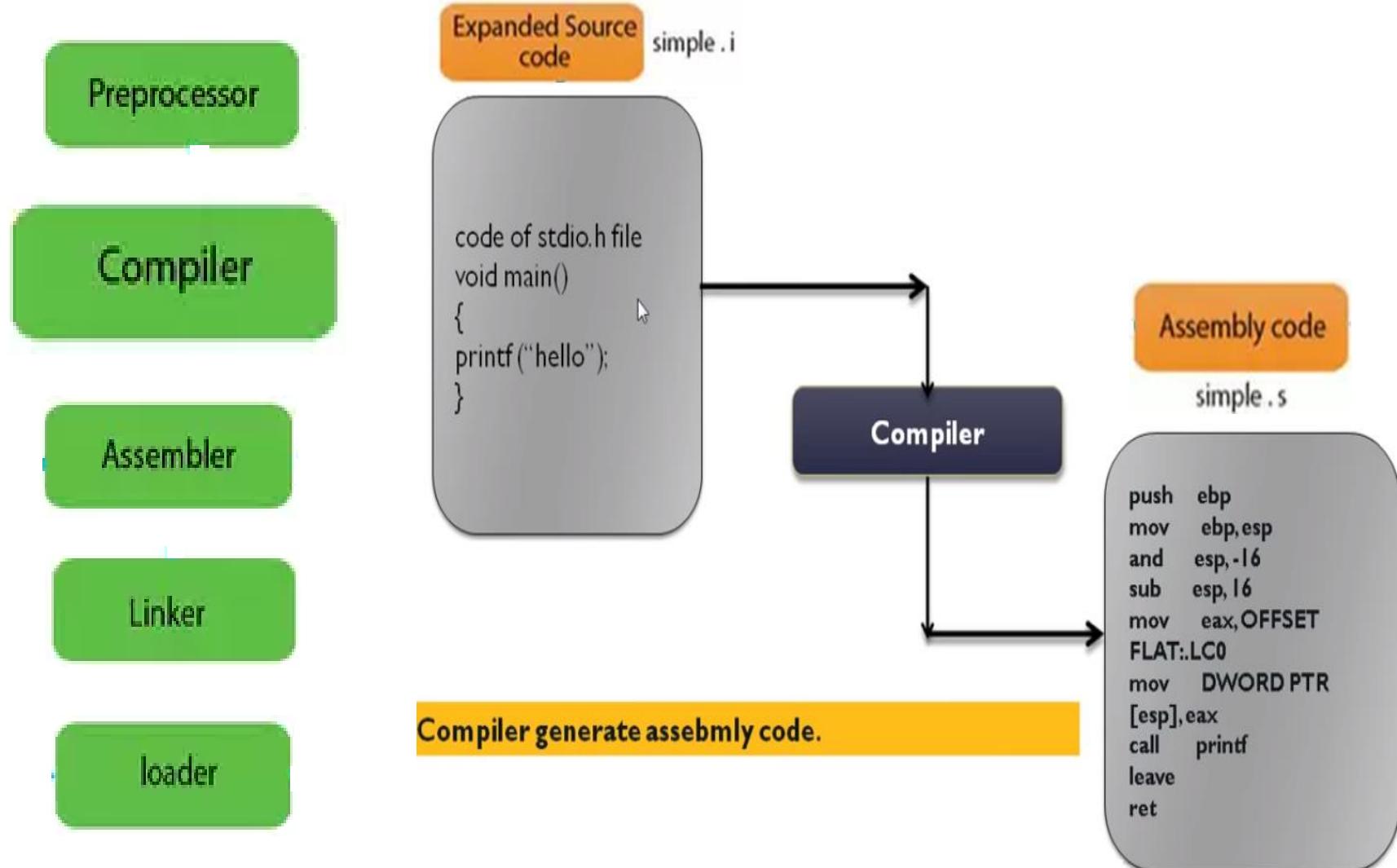
Preprocessor

Expanded Source code

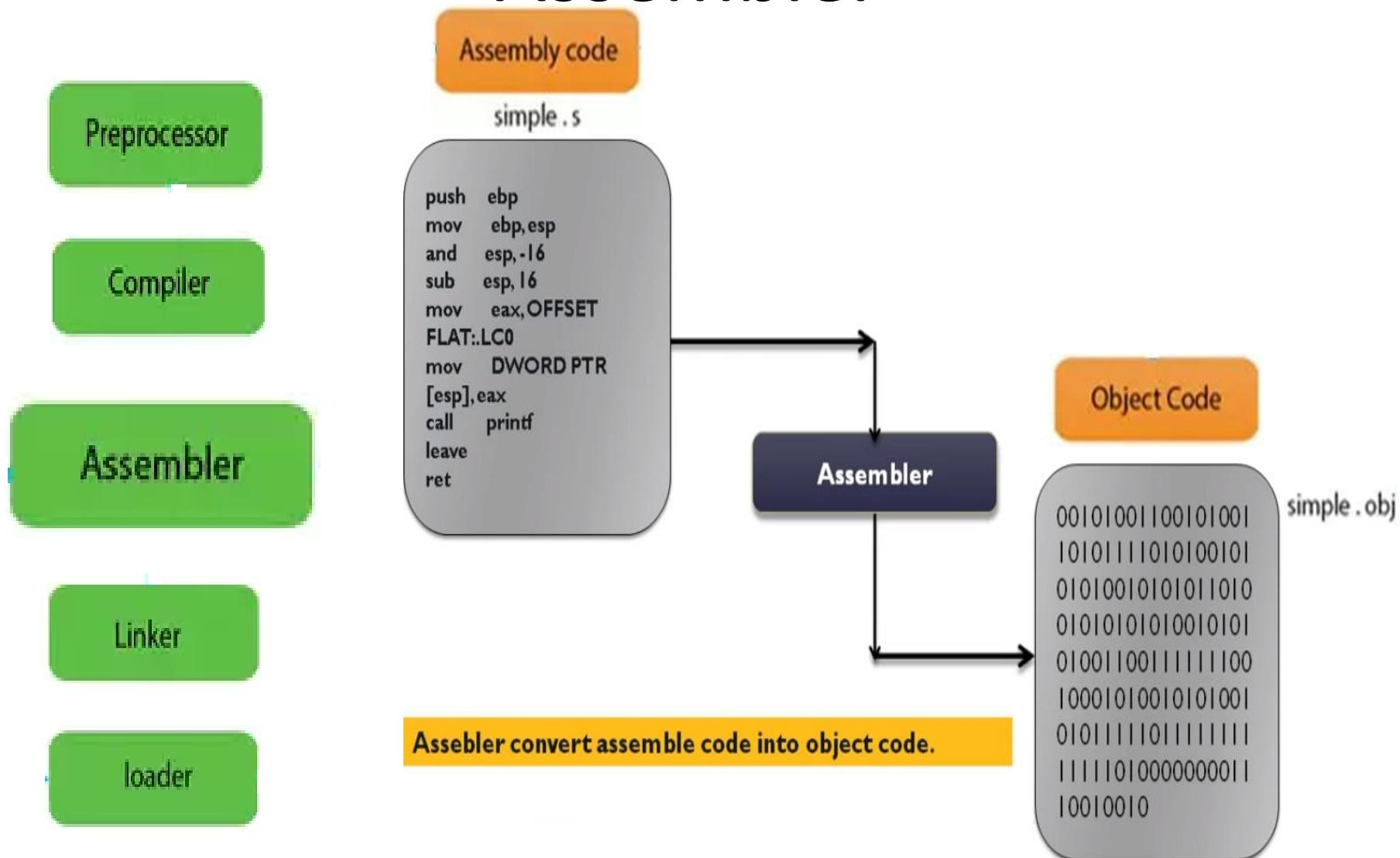
simple .i

```
code of stdio.h file  
void main()  
{  
    printf("hello");  
}
```

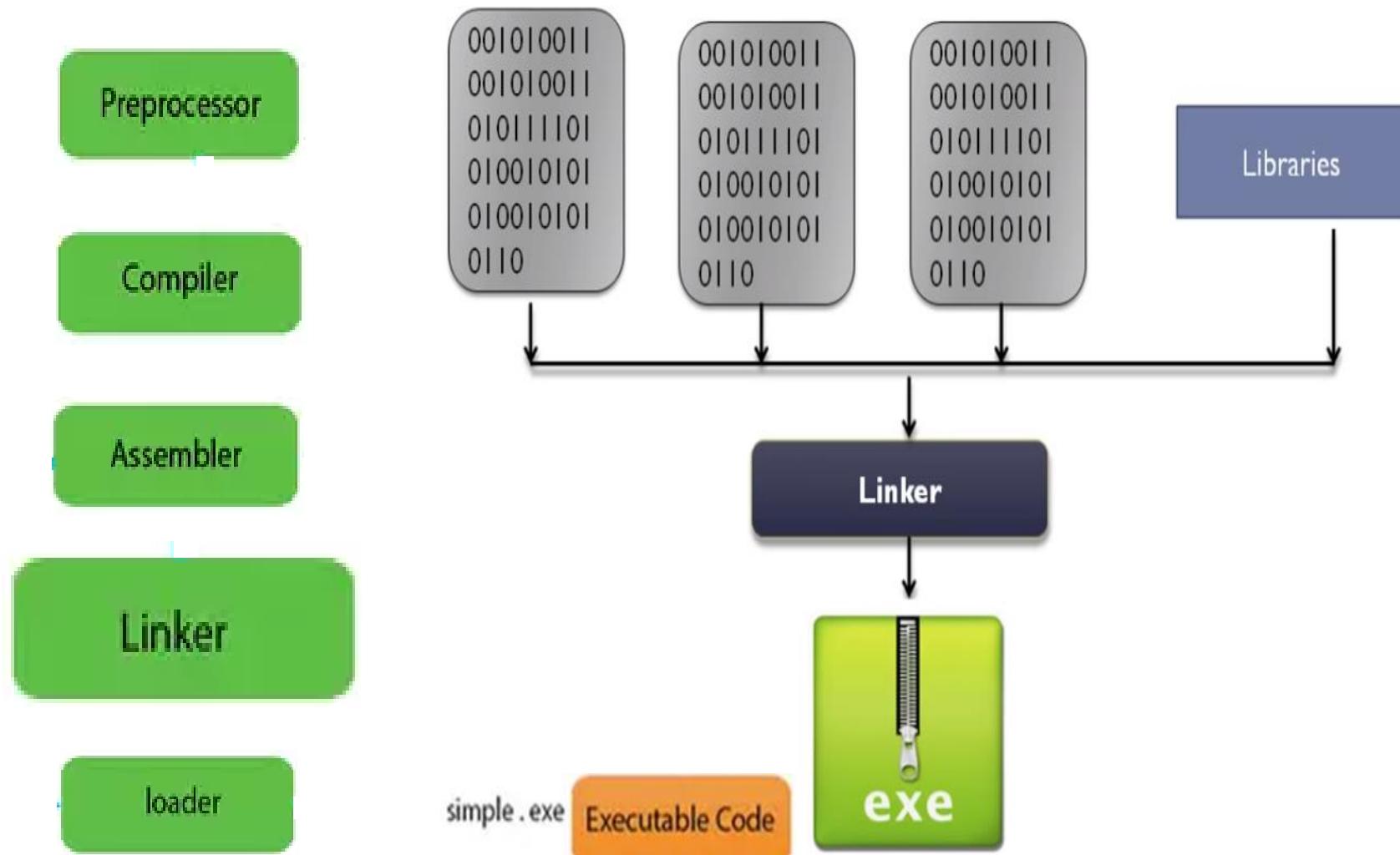
Preprocessor remove comments and include header files in source code, replace macro name with code.



Assembler



Linker



Loader

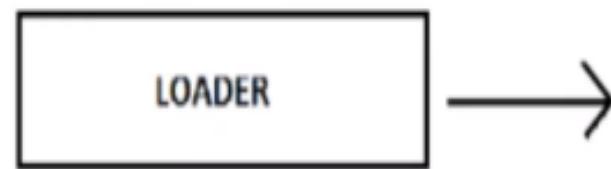
Preprocessor

Compiler

Assembler

Linker

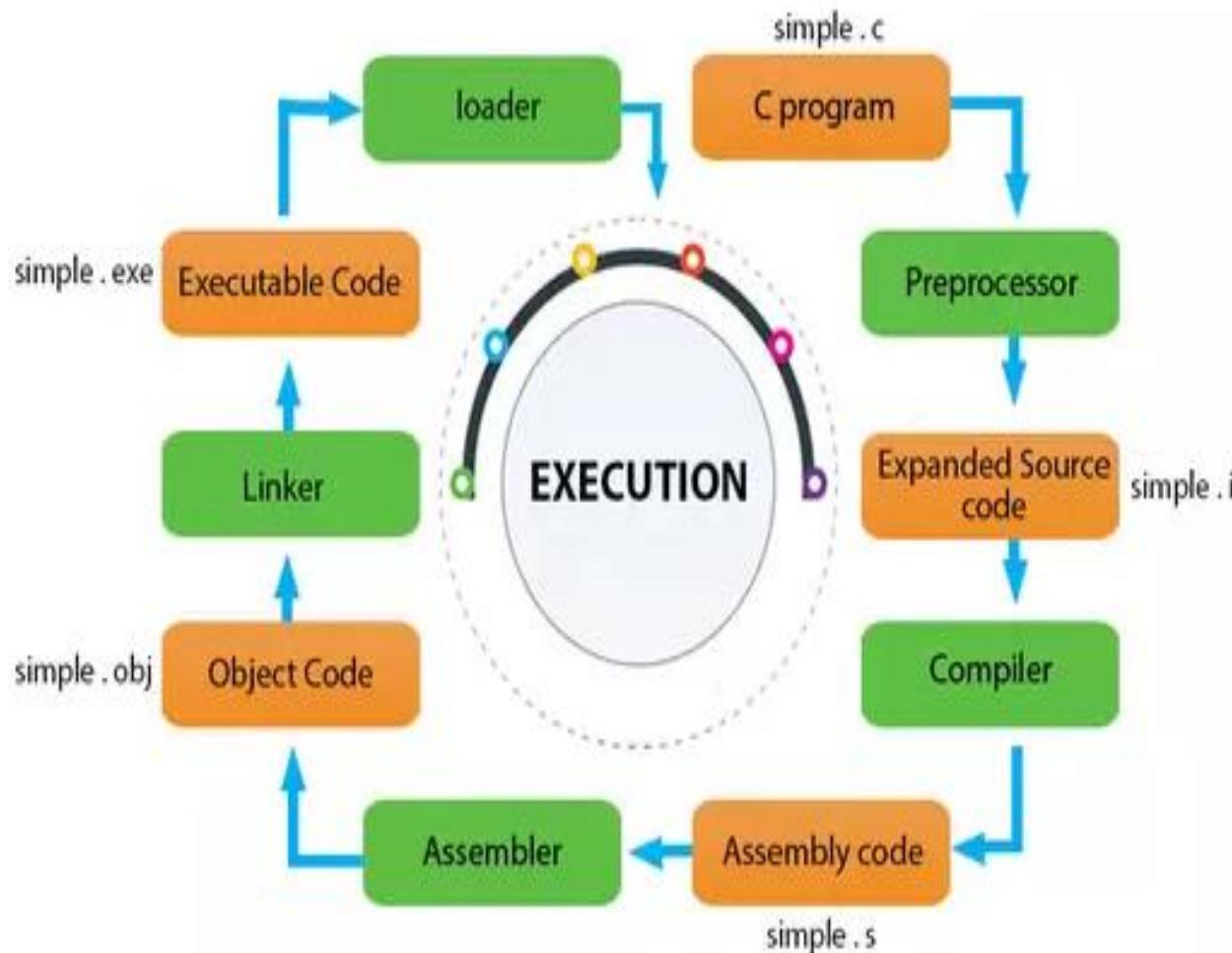
loader



MEMORY

Places these Machine
Code Object Decks in
Memory Ready for
Execution

Life Cycle of a C Program



C Character Set

Letters	Digits	Escape Sequences
Capital A to Z	All decimal digits 0 to 9	Blank space
Small a to z		Horizontal tab
		Vertical tab
		New line
		Form feed

C Character Set (Special Characters)

,	Comma	&	Ampersand
.	dot	^	Caret
;	Semicolon	*	Asterisk
:	Colon	-	Minus
'	Apostrophe	+	Plus
"	Quotation mark	<	Less than
!	Exclamation mark	>	Greater than
	Vertical bar	()	Parenthesis left/right
/	Slash	[]	Bracket left/right
\	Back slash	{ }	Braces left/right
~	Tilde	%	Percent
_	Underscore	#	Number sign or Hash
\$	Dollar	=	Equal to
?	Question mark	@	At the rate

C Character Set (Special Characters)

- Each character is assigned a unique numeric value.
- These values put together are called character codes.

Delimiters

Delimiters	Use
:	Colon Useful for label
;	Semicolon Terminates the statement
()	Parenthesis Used in expression and function
[]	Square Bracket Used for array declaration
{ }	Curly Brace Scope of the statement
#	hash Preprocessor directive
,	Comma Variable separator

Keywords or reserved words

The following list shows the reserved words in C. You cannot use them for variable names

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

Identifiers in C

C identifier is a name used to identify a variable, function, or any other user-defined item.

An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).

C is a **case-sensitive** programming language. Thus, *Manpower* and *manpower* are two different identifiers in C.

Rules for forming Identifier Names

- It cannot include any special characters or punctuation marks (like #, \$, ^, ?, . , etc.) except underscore ‘_’
- There cannot be two successive underscores
- Keywords cannot be used as identifiers
- Case of alphabetic characters is significant

Rules for forming Identifier Names

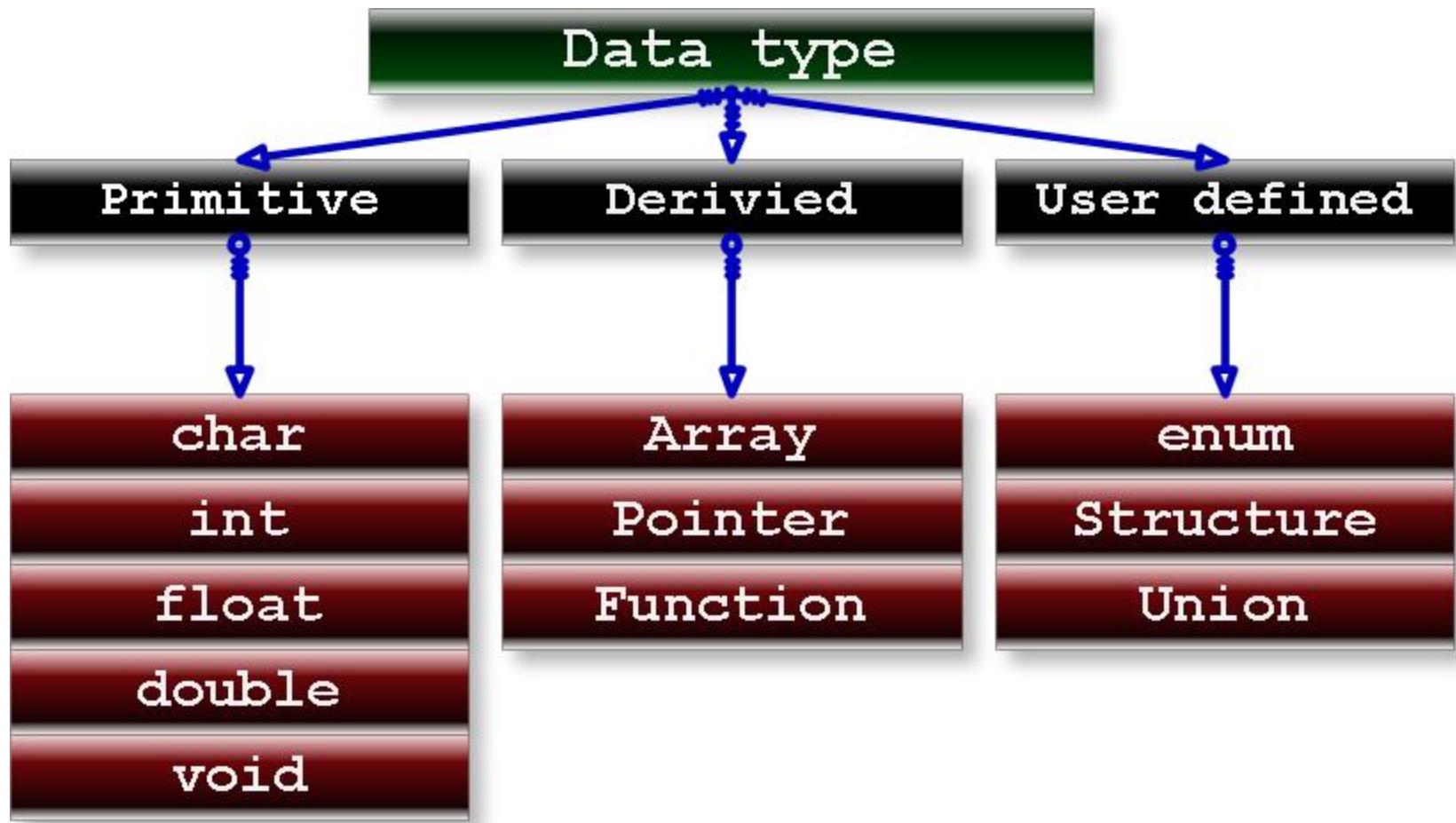
- It can be of any reasonable length
- It can be mainly upto 63 characters long
- Underscore can be used to separate parts of the name in an identifier

Data Types

Data types in C refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

C supports a number of qualifiers that can be applied to the basic data types e.g. short, long, unsigned and signed etc.

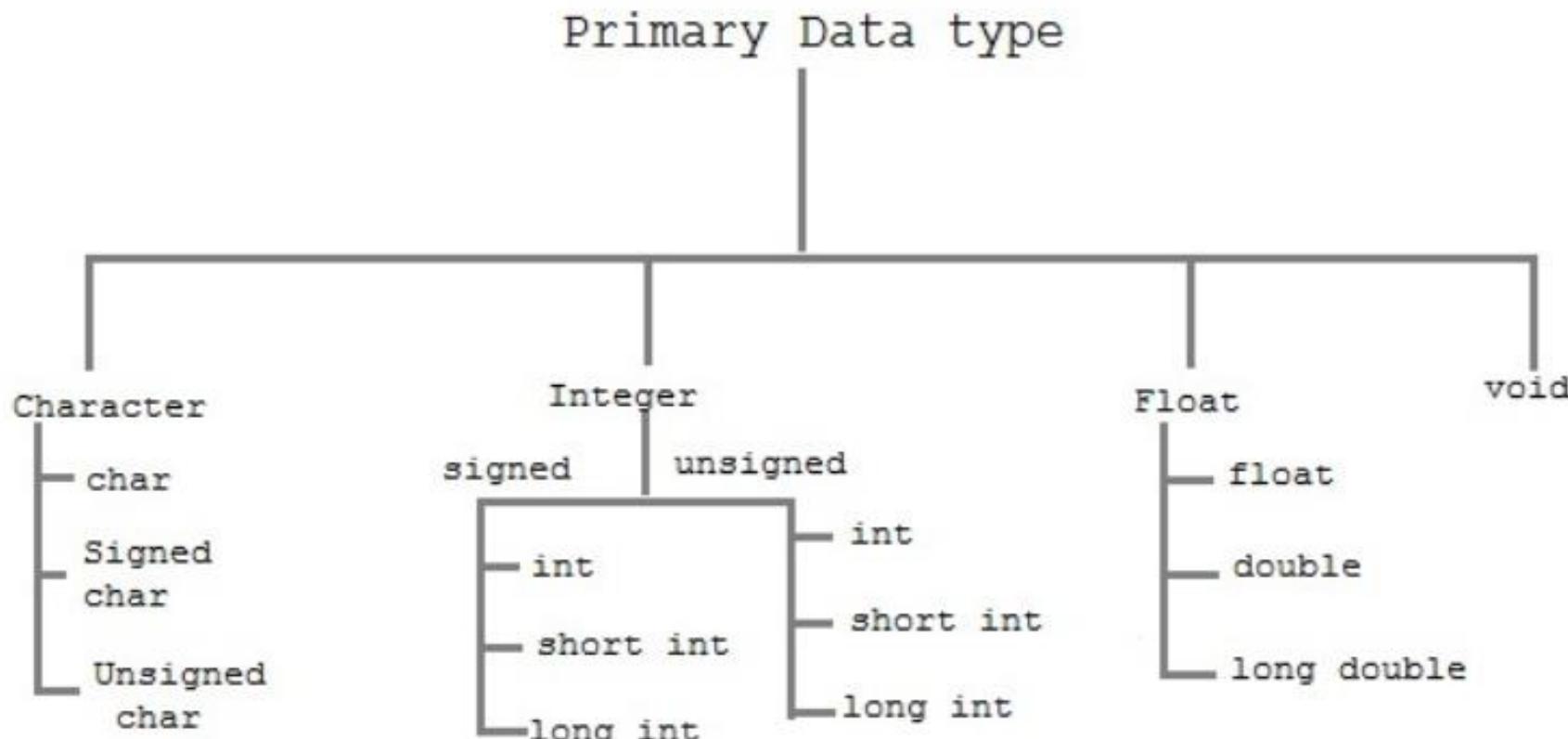
Data Types



Primitive Data types in C

Keyword	Identifier (Format Specifier)	Size (Bytes)	Data Range
char	%c	1	-128 to +127
int	%d	4	-2 ³¹ to +2 ³¹
float	%f	4	-3.4e38 to +3.4e38
double	%lf	8	-1.7e308 to +1.7e308
long int	%ld	8	-2 ⁶³ to +2 ⁶³
unsigned int	%u	4	0 to 2 ³²
long double	%Lf	16	
unsigned char	%c	1	0 to 255

Primitive Data Types



Difference between short and long integers

Short Integer

- 2 bytes in memory
- -32768 to +32767
- Program runs faster
- %d or %i
- int or short int

Long Integer

- 4 bytes in memory
- -2147483648 to +2147483647
- Program runs slower
- %ld
- long or long int

When a variable is declared without short or long keyword, the default is short-signed int

Difference between signed and unsigned integers

Short Signed Integer

- 2 bytes in memory
- -32768 to +32767
- %d or %i
- long signed integers
(-2147483648 to
+2147483647)

Short Unsigned Integer

- 2 bytes in memory
- 0 to 65535
- %u
- long unsigned int
(0 to 4294967295)

float and double

float

- 4 bytes in memory
- $3.4e-38$ to $+3.4e+38$
- `%f`
- `float`

double

- 8 bytes in memory
- $1.7e-308$ to $+1.7e+308$
- `%lf`
- `double`

long double $3.4e-4932$ to $1.1e+4932$, takes 10 bytes in memory

Difference between signed and unsigned char

Signed Character

- 1 byte in memory
- -128 to +127
- %c
- char

Unsigned Character

- 1 byte in memory
- 0 to 255
- %c
- unsigned char

When printed using %d format specifier, it prints corresponding ASCII character

Variables

- Variable is a meaningful name given to the data storage location in computer memory.
- When using variable, we actually refer to address of the memory where the data is stored.
- It can store one value at a time.
- Its value may be changed during the program execution.

Variables

```
#include<stdio.h>

int main()
{
    int a =10;
    printf("%d",a);
    return 0;
}
```

Rules for defining Variables

- It must begin with character or an underscore without spaces.
- Length of variables varies from compiler to compiler. However, ANSI standard recognizes max length of variable up to 31 characters.
- Its name should not be a C keyword
- It can be a combination of upper and lower case.
- It should not start with a digit
- Blanks and commas are not permitted

Examples of Variables

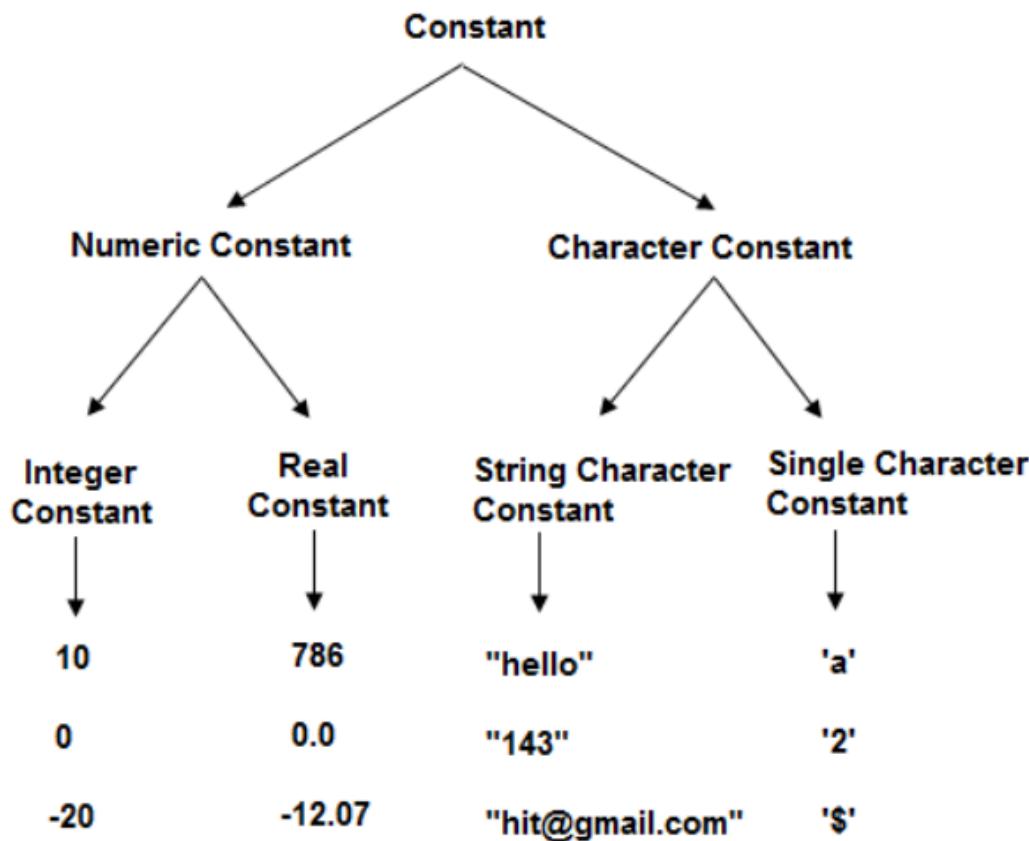
- int emp_num;
- float salary;
- char grade;
- double balance_amount;
- unsigned short int velocity;

Initializing Variables

- int emp_num = 7;
- float salary = 1100.50;
- char grade = 'A';
- double balance_amount = 5000;

Constants

- A constant is an entity whose value does not change during the execution of the program



Constants

- Declaration

```
const int var = 100;
```

Constants

```
#include<stdio.h>

int main()
{ const int a=10;
  printf("%d",a);
  a=20; // gives error you can't modify const
  return 0;
}
```

Variables and constants



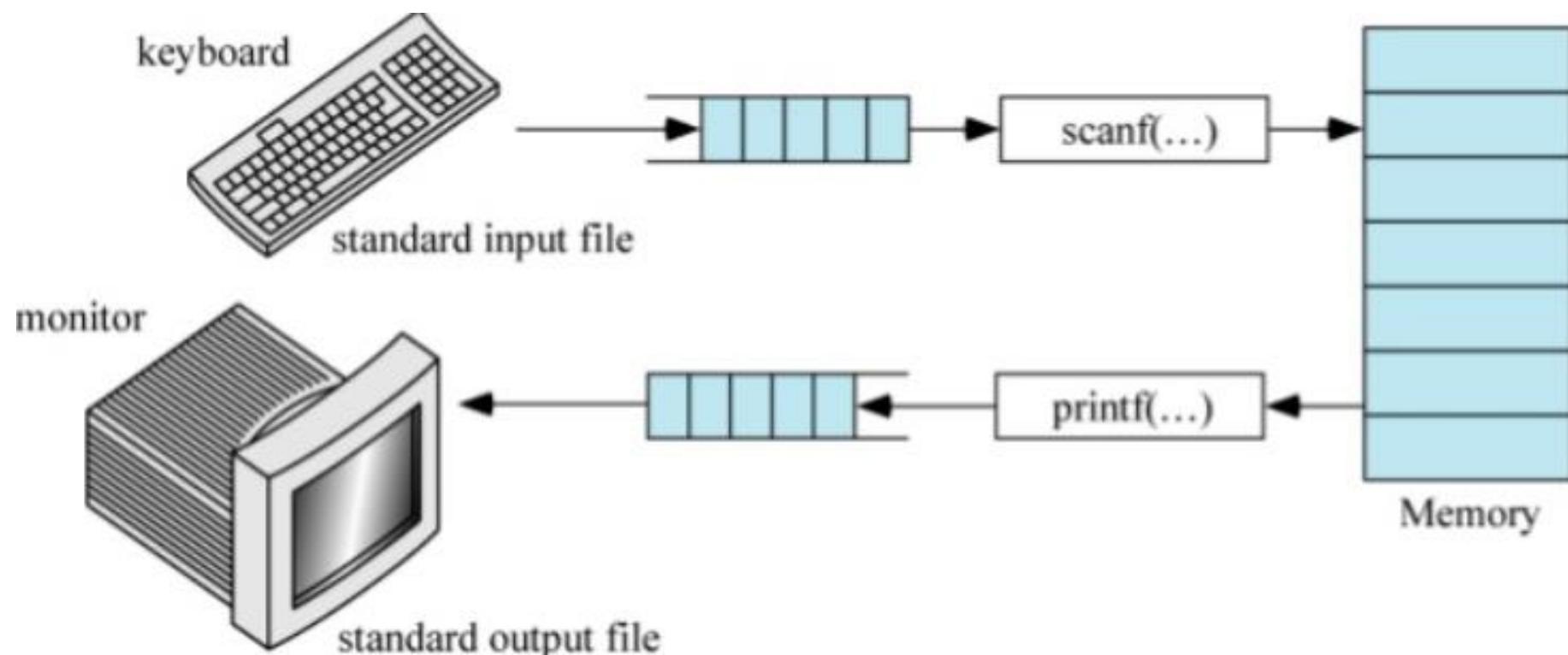
Variables



Constants

Input/ Output Statements

- C language supports two input/output functions printf and scanf.
- printf is used to convert data stored in a program into a text stream for output to the monitor.
- scanf is used to convert the text stream coming from the keyboard to data values and stores them in program variables.



printf()

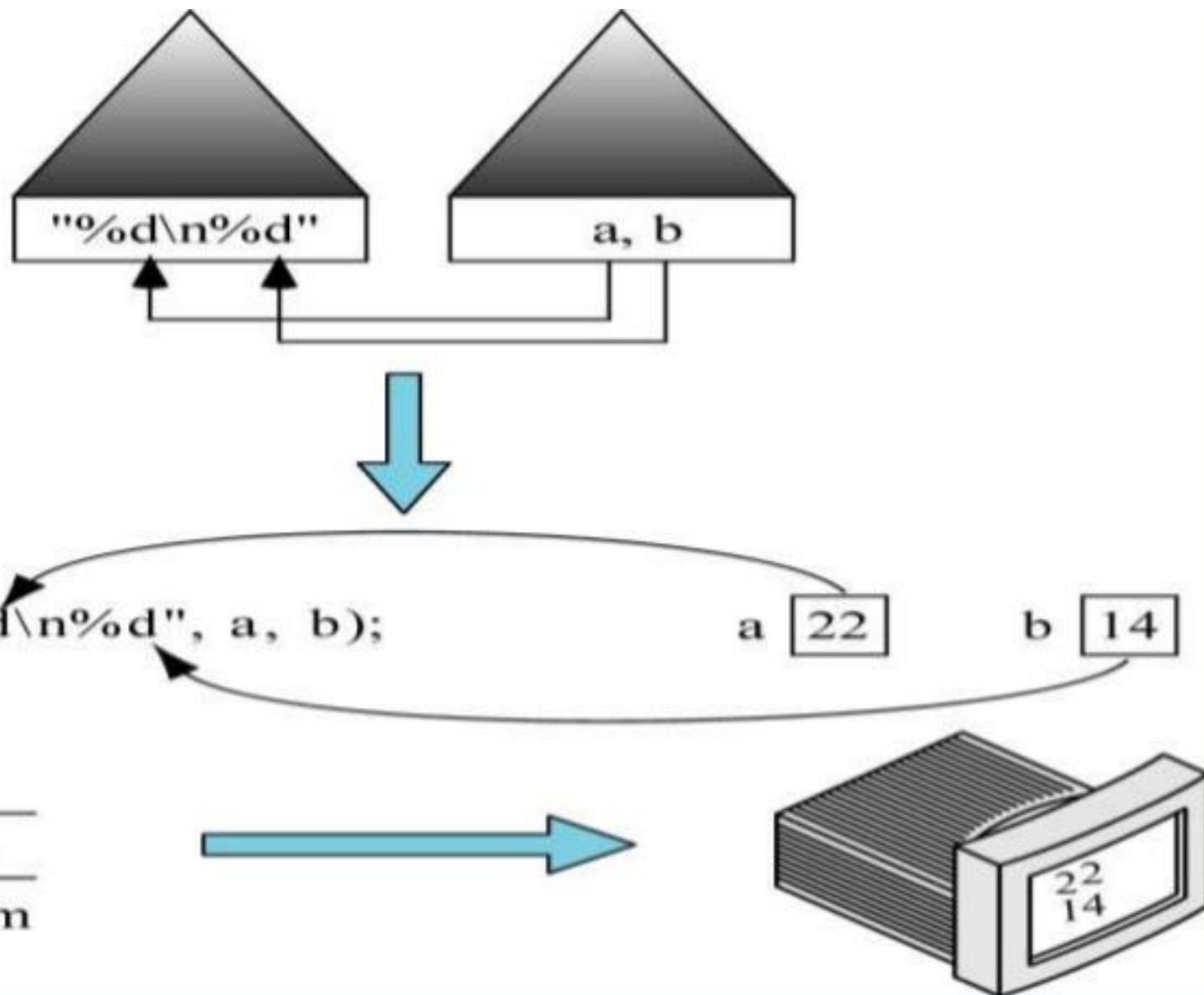
- Print Formatting.
- Displays information required by the user.
- Prints values of the variables.
- Takes data values, converts them to text stream using formatting functions specified in a control string and passes resulting text stream to standard output.

printf()

- The control string may contain zero or more conversion specifications, textual data, and control characters to be displayed.
- `printf(“control string” , arg1, arg2....., argn);`
- Control characters can also be included in printf statement.
 - `\n , \t , \r , \a` etc.

printf()

- `printf("This is a message \n");`
- How do we print the value of a variable?
`int a = 22, b = 14;`
- Answer: Use special format specifiers depending on the type of the variable
- Format specifiers begin with % sign



Format Specifier	Description
%d	Integer Format Specifier
%f	Float Format Specifier
%c	Character Format Specifier
%s	String Format Specifier
%u	Unsigned Integer Format Specifier
%ld	Long Int Format Specifier

scanf()

- Scan Formatting.
- Read formatted data from the keyboard.
- Takes text stream from the keyboard, extracts and formats data from the stream according to a format control string and then stores the data in specified program variables.

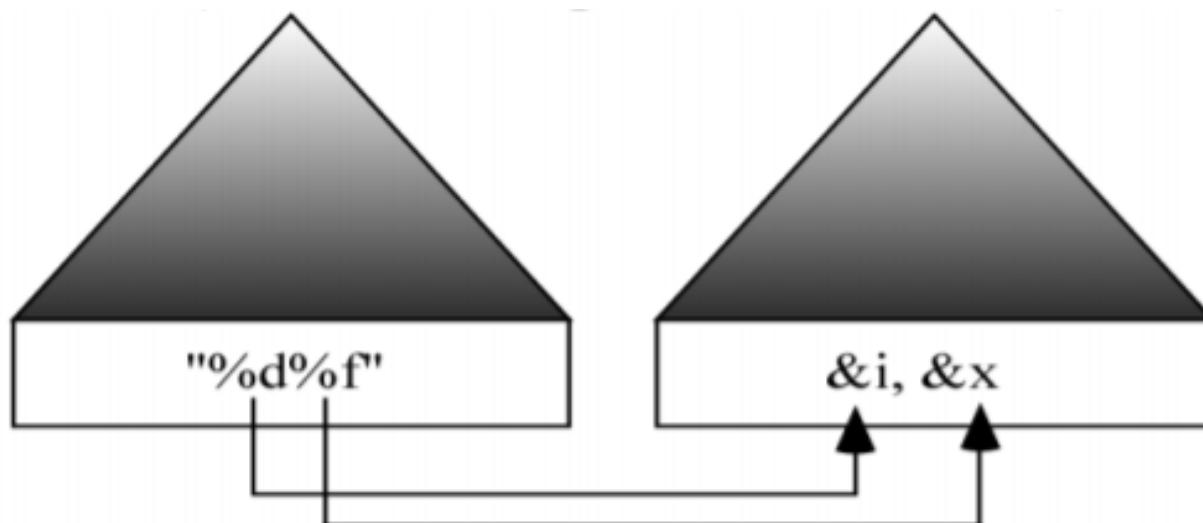
scanf()

- `scanf(“control string” , arg1, arg2....., argn);`
- The control specifies type and format of data obtained from the keyboard and stored in memory locations pointed by the arguments.
- The arguments are actually variable addresses where each piece of data are to be stored.
- Ignores blank spaces, tabs, newline

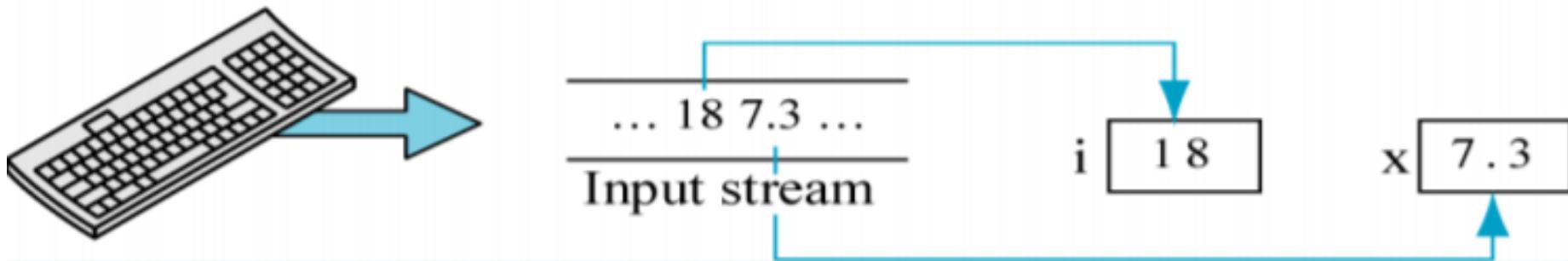
scanf()

```
#include <stdio.h>

int main()
{ int i ;
  float x;
  scanf("%d%f", &i, &x);
  return 0;
}
```



```
scanf("%d %f", &i, &x);
```



Example

```
#include <stdio.h>
int main()
{
    printf(" \n Result: %d %c%f", 12, 'a', 2.3);
return 0;
}
```

Result: 12a2.3

More I/O statements in C

Input: gets()

Output: puts()

```
main(){  
    char s[20]; //string is array of char  
    puts("Enter a string: "); gets(s);  
    puts("Here is your string"); puts(s);  
}
```

C operators

- To build an expression, operators are used with operands like $4 + 5$, Here, 4 and 5 are operands and '+' is operator

C contains following group of operators –

- 1) Arithmetic
- 2) Assignment
- 3) Logical/Relational
- 4) Bitwise
- 5) Miscellaneous

Arithmetic Operators

Symbol	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder)
--	Decrement
++	Increment

Pre increment/Decrement (++i or --i):

First increment/decrement current value by 1 and then use the incremented/decremented value

Post increment/Decrement (i++ or i--):

First use the current value and then increment/decrement current value by 1 for next time use

Arithmetic operators Example

```
#include<stdio.h>
void main()
{
int i=7,j=5;
printf("%d %d %d %d \n", i+j, i-j, i*j, i/j );      12  2  35 1
printf("%d \n", i%j );                                2
printf("%d %d \n", i++, ++j );                      7  6
}
```

Assignment Operators

- Each expression contains two parts: lvalue and rvalue,
- Arithmetic operations are performed to calculate rvalue,
- Assignment operators are used to assign the rvalue to lvalue

Example:

$i = i + 1$

can also be reduced to

$i += 1$

Symbol	Name
=	Assign
*=	Multiply
/=	Divide
-=	Subtract
+=	Add
%=	Modulus

Assignment Operators Example

```
#include<stdio.h>
void main()
{
int i=6;
printf("%d \n",i=8);
i+=1;
printf("%d \n",i);
i*=2;
printf("%d \n",i);
i/=3;
printf("%d \n",i);
i%=4;
printf("%d \n",i);

}
```

8
9
18
6
2

Relational Operators

A **relational operator** is a programming language construct or operator that tests or defines some kind of relation between two entities.

Symbol	Name
<code>==</code>	Equal to
<code><</code>	Less than
<code>></code>	Greater Than
<code><=</code>	Less or equal
<code>>=</code>	Greater or equal
<code>!=</code>	Not equal

- If condition is true it returns 1 else returns 0.

Relational Operators (Cont..)

```
#include<stdio.h>
void main()
{
int i=7,j=1,k=0;
printf("%d \n",i==j);          0
printf("%d \n",i<j);          0
printf("%d \n",i>j);          1
printf("%d \n",i<=j);         0
printf("%d \n",i>=j);         1
printf("%d \n",i!=j);         1
}
```

Relational Operators (Cont..)

```
/* C program to find largest number using if...else statement */
#include <stdio.h>

int main()
{   float a, b, c;
    printf("Enter three numbers: ");
    scanf("%f %f %f", &a, &b, &c);
    if (a>=b) {
        if(a>=c)
            printf("Largest number = %f",a);
        else
            printf("Largest number = %f",c);
    }
    else {
        if(b>=c)
            printf("Largest number = %f",b);
        else
            printf("Largest number = %f",c);
    }
    return 0;
}
```

Enter three numbers: 10
30
60
Largest number = 60.000000

Logical Operators

- Allow a program to make a decision based on multiple conditions.
- Each operand is considered a condition that can be evaluated to a true or false value.
- **Logical operator returns 0 or 1.**
- The Logical operators are: `&&`, `||` and `!`
- **op1 && op2**-- Performs a logical AND of the two operands.
- **op1 || op2**-- Performs a logical OR of the two operands.
- **!op1**-- Performs a logical NOT of the operand.
- Op1 and op2 may be one condition or single operand
- **C considers non-zero value to true and zero to false.**
- Examples:($a>b$)`&&`($a>c$) , $a\&\&b$, $5\&\&8$ etc.

Logical Operators

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

OR

A	not A
0	1
1	0

NOT

&&(Logical AND)

- The && operator is used to determine whether both operands or conditions are true
 - For example:
 - a=50,b=9
 - if (a == 10 && b == 9)
 printf("Hello!");
- If either of the two conditions is false or incorrect, then the printf command is bypassed.

Example of Logical operator

```
/* C program to find largest number using if statement only */

#include <stdio.h>
int main(){
    float a, b, c;
    printf("Enter three numbers: ");
    scanf("%f %f %f", &a, &b, &c);
    if(a>=b && a>=c)
        printf("Largest number = %.2f", a);
    if(b>=a && b>=c)
        printf("Largest number = %.2f", b);
    if(c>=a && c>=b)
        printf("Largest number = %.2f", c);
    return 0;
}
```

Output:

```
Enter three numbers: 12.2
13.452
10.193
Largest number = 13.45
```

|| (Logical OR)

- The || operator is used to determine whether either of the condition is true.
- For example:
- a=50,b=9
- if (a== 10 || b == 9) // either of the condition should be true for printing hello!
 printf("Hello!");

If both of the two conditions are false, then only the printf command is bypassed.

Ex2: if(0| |9)

```
    printf("Correct !");  
else  
    printf("Incorrect !");
```

Caution: If the first operand of the || operator evaluates to true, the second operand will not be evaluated. This could be a source of bugs if you are not careful.

- For instance, in the following code fragment:
- if (9 || X++) {
- print("X=%d\n",X); } variable X will not be incremented because first condition is evaluates to true.

|| (Logical OR)

Caution: If the first operand of the || operator evaluates to true, the second operand will not be evaluated. This could be a source of bugs if you are not careful.

For instance, in the following code fragment:

```
if (9 || X++) {  
    print("X=%d\n",X); }
```

variable X will not be incremented because first condition is evaluates to true.

! (Logical NOT)

- The ! operator is used to convert true values to false and false values to true. In other words, it inverts a value
- For example:
- a=50,b=9
- if (!(a== 10)) //
 printf("Hello!");

Ex2: if(!(0 | 9))
 printf("Correct !");
else
 printf("Incorrect !");

Bitwise Operators

- In the C programming language, operations can be performed on a bit level using bitwise operators.
- Following are the bitwise Operators

Symbol	Operator
&	bitwise AND
	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	bitwise NOT (one's complement) (unary)

Bitwise AND

- A **bitwise AND** takes two binary representations of equal length and performs the logical AND operation on each pair of corresponding bits.

The result in each position is 1 if the first bit is 1

and the second bit is 1; otherwise, the result is 0.

bit a	bit b	a & b (a AND b)
0	0	0
0	1	0
1	0	0
1	1	1

Example: a=5, b=3;
`printf("%d",a&b);` will print 1

0101--→5

& 0011--→3
0001--→1

Bitwise AND

- & operation may be used to determine whether a particular bit is *set* (1) or *clear* (0). For example,
- given a bit pattern 0011 (decimal 3), to determine whether the second bit is set; we use a bitwise AND with a bit pattern containing 1 only in the second bit:
 - 0011 (decimal 3)
 - & 0010 (decimal 2)
 - 0010 (decimal 2)

Bitwise OR(|)

A **bitwise OR** takes two bit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding bits.

The result in each position is 1 if the first bit is 1 *or* the second bit is 1 *or* both bits are 1; otherwise, the result is 0.

Example: a=5, b=3;
`printf("%d",a|b);`
will print 7

0101--→5
| 0011--→3
0111--→7

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

OR

Bitwise XOR(^)

A **bitwise XOR(^)** takes two bit patterns of equal length and performs the logical exclusive OR operation on each pair of corresponding bits.

The result in each position is 1 if only the first bit is 1 *or* only the second bit is 1, but will be 0 if both are 0 or both are 1.

`printf("%d",a^b);` will print 6

0101--→5
^ 0011--→3
0110--→6

**XOR Truth
Table**

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Shift Operators

Left Shift Operator (`<<`): The left shift operator will shift the bits towards left for the given number of times.

```
int a=2<<1;           will print 4  
Printf("%d",a);//
```

If you left shift like `2<<2`, then it will give the result as 8.

Therefore left shifting n time, is equal to multiplying the value by 2^n .

Shift Operators

Right shift Operator (>>)

The right shift operator will shift the bits towards right for the given number of times

```
int b=4>>1           will print 2  
printf("%d",b);//
```

Right shifting n time, is equivalent to dividing the value by 2^n .

Bitwise Operators Example

```
#include<stdio.h>
void main()
{
    int i=5,j=7,c;
    c=i & j;
    printf("%d \n",c);
    c=i | j;
    printf("%d \n",c);
    c=i ^ j;
    printf("%d \n",c);
    i-=1;
    printf("%d %d\n",i<<=3,i>>=2);

}
```

Miscellaneous Operators

- There are few other important operators including **sizeof** and **? :** supported by C Language.

Operator	Description	Example
sizeof()	Returns the size of an variable.	sizeof(a), where a is interger, will return 4.
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

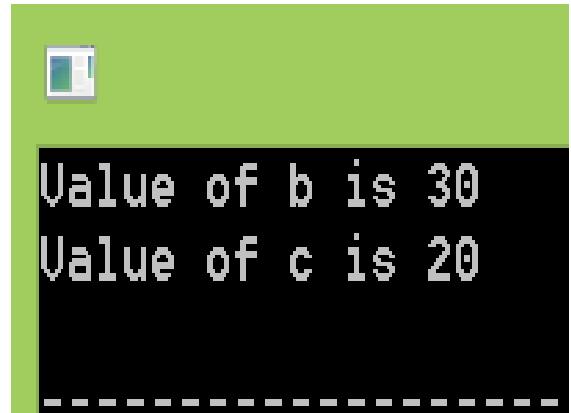
Examples of sizeof()

```
#include <stdio.h>
main() {
int a = 4;
float b=50.45;
char c;
printf("Size of variable a = %d\n", sizeof(a) );
printf("Size of variable b = %d\n", sizeof(b) );
printf("Size of variable c= %d\n", sizeof(c) );
printf("Size of 233= %d\n", sizeof(233) );
printf("Size of 20.55= %d\n", sizeof(20.55) );
printf("Size of 'y'= %d\n", sizeof('y') );
printf("Size of unsigned int= %d\n", sizeof(unsigned int) );
}
```

Size of variable a = 4
Size of variable b = 4
Size of variable c= 1
Size of 233= 4
Size of 20.55= 8
Size of 'y'= 1
Size of unsigned int= 4

Examples of conditional operator(Ternary Operator)

```
main() {  
    int a , b,c;  
    a = 10;  
    b = (a == 1) ? 20: 30;  
    printf( "Value of b is %d\n", b );  
    c = (a == 10) ? 20: 30;  
    printf( "Value of c is %d\n", c );  
}
```



WAP to check even or odd using conditional operator

```
#include<stdio.h>
main() {
int n;
printf("Enter number n");
scanf("%d",&n);
(n%2 == 0) ? printf("%d is even",n): printf("%d is odd",n);
}
```

Operators available in C can be classified in following ways:

1. unary – that requires only one operand.

For example: &a, ++a, *a ,--b etc.

2. binary - that requires two operands.

For example: a + b, a * b, a<<2

3. ternary - that requires three operands.

For example: (a>b) ? a : b [?:]

Precedence and Associativity

- If more than one operators are involved in an expression then, C language has predefined rule of priority of operators. This rule of priority of operators is called **operator precedence**.
- Associativity** indicates in which order two operators of same precedence (priority) executes.

Operators	Associativity
(expression) [index] → .	LR
! ~ ++ -- (type) sizeof Unary * &	RL
* / %	LR
+ -	LR
>> <<	LR
> >= < <=	LR
== !=	LR
Binary &	LR
Binary ^	LR

Operators	Associativity
Binary	LR
&&	LR
	LR
+= -= *= /= %= etc. (all assignments)	RL
,	LR

LR : Left to Right

RL: Right to Left

Examples of precedence of operators

```
(a>b+c&&d)
```

This expression is equivalent to:

```
((a>(b+c))&&d)
```

i.e, $(b+c)$ executes first

then, $(a>(b+c))$ executes

then, $((a>(b+c))&&d)$ executes

- precedence of arithmetic operators(*,%,/,+,-) is higher than relational operators(==,!>,>,<,>=,<=) and precedence of relational operator is higher than logical operators(&&, || and !).

Example of Associativity of Operators

Example 1:

`a==b!=c`

Here, operators == and != have same precedence.

The associativity of both == and != is left to right, i.e, the expression in left is executed first and execution take place towards right.

Thus, `a==b!=c` equivalent to :

`(a==b)!=c`

Example of Associativity of Operators

Example 2:

Associativity is important, since it changes the meaning of an expression. Consider the division operator with integer arithmetic, which is left associative

$$4 / 2 / 3 \rightarrow (4 / 2) / 3 \rightarrow 2 / 3 = 0$$

If it were right associative, it would evaluate to an undefined expression, since you would divide by zero

$$4 / 2 / 3 \rightarrow 4 / (2 / 3) \rightarrow 4 / 0 = \text{undefined}$$

What will be the value of i

i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8

i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8 operation: *

i = 1 + 4 / 4 + 8 - 2 + 5 / 8 operation: /

i = 1 + 1 + 8 - 2 + 5 / 8 operation: /

i = 1 + 1 + 8 - 2 + 0 operation: /

i = 2 + 8 - 2 + 0 operation: +

i = 10 - 2 + 0 operation: +

i = 8 + 0 operation : -

i = 8 operation: +

Arithmetic expressions vs C expressions

- Although Arithmetic instructions look simple to use one often commits mistakes in writing them.
- following points should keep in mind while writing C programs:
 - (a) C allows only one variable on left-hand side of = **That is, $z = k * l$ is legal, whereas $k * l = z$ is illegal.**
 - (b) Other than division operator(/) C has modulus operator(%). Thus the expression $10 / 2$ yields 5, whereas, $10 \% 2$ yields 0.
 - Note that the modulus operator (%) **cannot be applied on a float.**
 - Also note that on using % the sign of the remainder is always same as the sign of the numerator. Thus $-5 \% 2$ yields -1 , whereas, $5 \% -2$ yields 1.

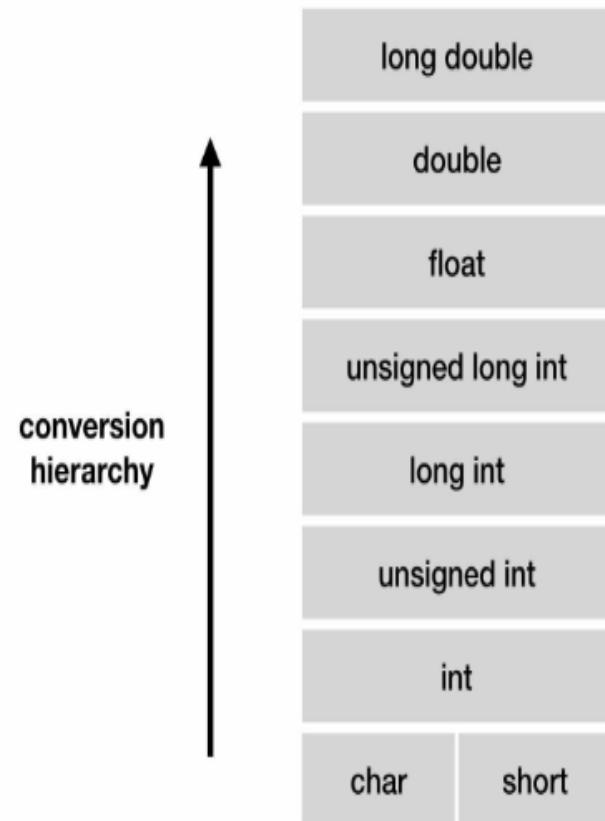
Type Conversion & Type Casting

- Type conversion or type casting of variables refers to changing a variable of one data type into another.
- Type conversion is done implicitly whereas,
- Type casting has to be done explicitly

Type Conversion

It is done when the expression has variables of different data types.

To evaluate the expression, the data type is promoted from lower to higher level where the hierarchy of data types is given in the figure



Type Conversion

```
float x;  
int y = 3;  
x = y;
```

```
#include<stdio.h>  
int main(){  
    float x=81.2;  
    printf("%f",x);  
}  
// 81.199997 due to digital  
// conversion error
```

Output:

x = 3.000000

Type Conversion Example

```
#include <stdio.h>
main()
{ int number = 1;
  char character = 'k';
  int sum;
  sum = number + character;
  printf("Value of sum : %d\n", sum );
}
```

OUTPUT
Value of sum :
10 8

Important regarding Type Conversion

- Type conversion is also called as implicit or standard type conversion.
- We do not require any keyword or special statements in implicit type casting.
- Converting from smaller data type into larger data type is also called as **type promotion**.
- The type conversion always happens with the compatible data types.

Important regarding Type Conversion

- Converting float to an int will truncate the fraction part hence losing the meaning of the value.
- Converting double to float will round up the digits.
- Converting long int to int will cause dropping of excess high order bits.

Type Casting

- Type casting is also known as forced conversion or explicit type conversion
- It is done when the value of a higher data type has to be converted into the value of a lower data type.

```
float salary = 10000.00;  
int sal;  
sal = (int) salary;
```

Type Casting

```
res = (int) 9.5;
```

```
res = (int) 12.3 / (int) 4.2;
```

```
res = (double) total / n;
```

```
res = (int) a + b;
```

```
res = cos ( (double) x);
```

Thank You

C Data Types

In this tutorial, you will learn about basic data types such as int, float, char etc. in C programming.

Video: Data Types in C Programming

In C programming, data types are declarations for variables. This determines the type and size of data associated with variables. For example,

```
int myVar;
```

Here, `myVar` is a variable of `int` (integer) type. The size of `int` is 4 bytes.

Basic types

Here's a table containing commonly used types in C programming for quick access.

Type	Size (bytes)	Format Specifier
int	at least 2, usually 4	%d, %i
char	1	%c
float	4	%f
double	8	%lf
short int	2 usually	%hd
unsigned int	at least 2, usually 4	%u
long int	at least 4, usually 8	%ld, %li

long long int	at least 8	%lld, %lli
unsigned long int	at least 4	%lu
unsigned long long int	at least 8	%llu
signed char	1	%c
unsigned char	1	%c
long double	at least 10, usually 12 or 16	%Lf

int

Integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, 0, -5, 10
We can use `int` for declaring an integer variable.

```
int id;
```

Here, `id` is a variable of type integer.

You can declare multiple variables at once in C programming. For example,

```
int id, age;
```

The size of `int` is usually 4 bytes (32 bits). And, it can take 2^{32} distinct states from -2147483648 to 2147483647.

float and double

`float` and `double` are used to hold real numbers.

```
float salary;
double price;
```

In C, floating-point numbers can also be represented in exponential. For example,

```
float normalizationFactor = 22.442e2;
```

What's the difference between `float` and `double`?

The size of `float` (single precision float data type) is 4 bytes. And the size of `double` (double precision float data type) is 8 bytes.

char

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

```
char test = 'h';
```

The size of the character variable is 1 byte.

void

`void` is an incomplete type. It means "nothing" or "no type". You can think of `void` as **absent**.

For example, if a function is not returning anything, its return type should be `void`.

Note that, you cannot create variables of `void` type.

short and long

If you need to use a large number, you can use a type specifier `long`. Here's how:

```
long a;  
long long b;
```

```
long double c;
```

Here variables `a` and `b` can store integer values. And, `c` can store a floating-point number.

If you are sure, only a small integer ($[-32,767, +32,767]$ range) will be used, you can use `short`.

```
short d;
```

You can always check the size of a variable using the `sizeof()` operator.

```
#include <stdio.h>
int main() {
    short a;
    long b;
    long long c;
    long double d;

    printf("size of short = %d bytes\n", sizeof(a));
    printf("size of long = %d bytes\n", sizeof(b));
    printf("size of long long = %d bytes\n", sizeof(c));
    printf("size of long double= %d bytes\n", sizeof(d));
    return 0;
}
```

signed and unsigned

In C, `signed` and `unsigned` are type modifiers. You can alter the data storage of a data type by using them:

- `signed` - allows for storage of both positive and negative numbers
- `unsigned` - allows for storage of only positive numbers

For example,

```
// valid codes
unsigned int x = 35;
int y = -35; // signed int
int z = 36; // signed int

// invalid code: unsigned int cannot hold negative integers
unsigned int num = -35;
```

Here, the variables `x` and `num` can hold only zero and positive values because we have used the `unsigned` modifier.

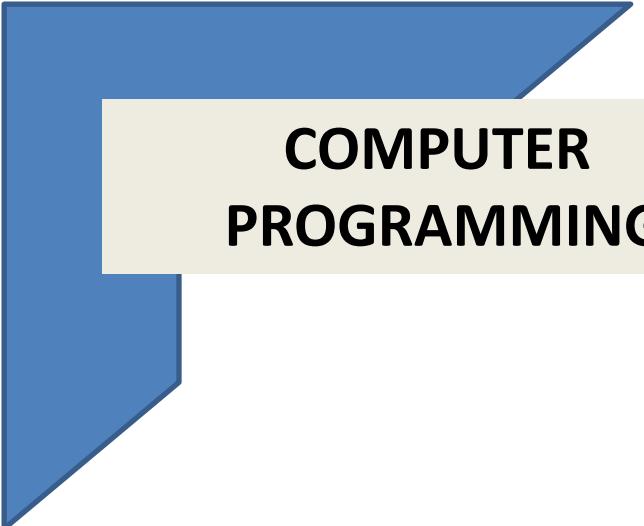
Considering the size of `int` is 4 bytes, variable `y` can hold values from -2^{31} to $2^{31}-1$, whereas variable `x` can hold values from 0 to $2^{32}-1$.

Derived Data Types

Data types that are derived from fundamental data types are derived types. For example: arrays, pointers, function types, structures, etc.

We will learn about these derived data types in later tutorials.

- bool type
- Enumerated type
- Complex types



COMPUTER PROGRAMMING

Conditional and Iterative Statements

COMPUTER PROGRAMMING

**Conditional
Statements**

Branching

Topics covered

- Conditional branching statements, iterative statements, nested loops, break and continue statements.

Branching

- ❖ C program may require that a logical test be carried out at some particular point within the program.
- ❖ One of several possible actions will then be carried out, depending on the output of the logical test.

This is known as Branching.

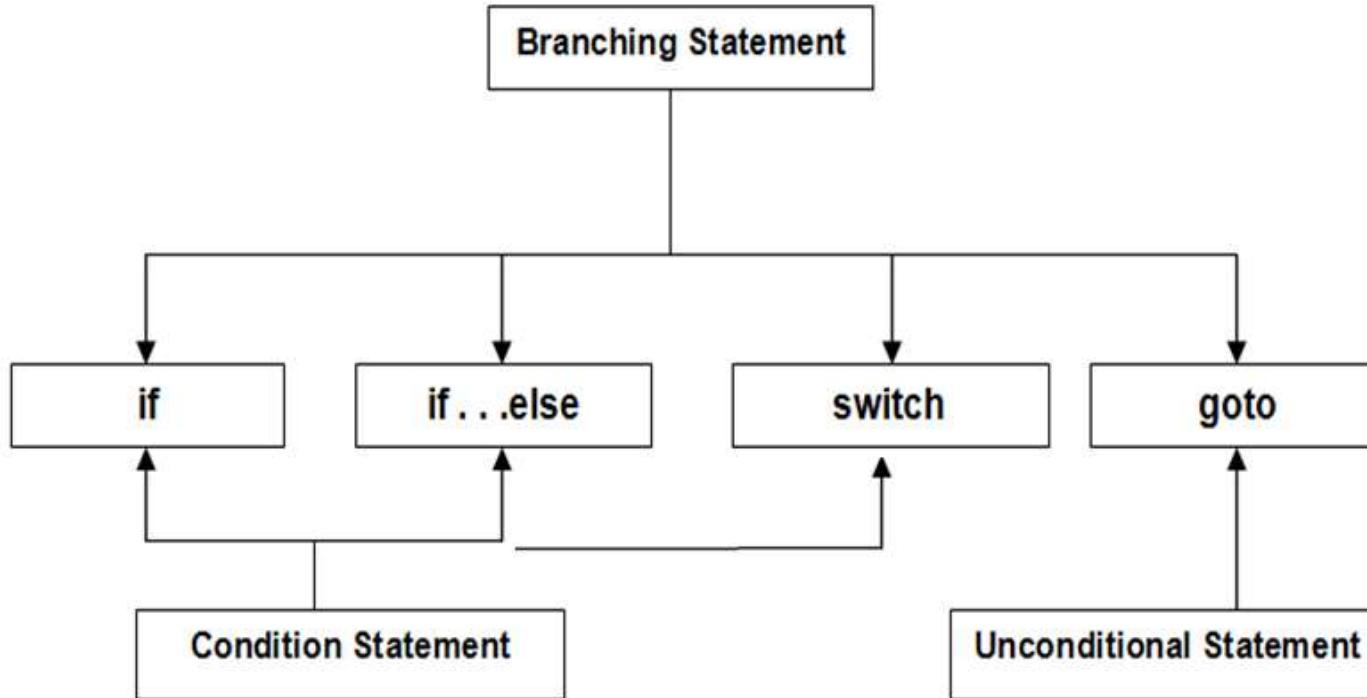
Branching

- ❖ There is also special kind of branching, called Selection, in which one group of statements is selected from several available groups.

DECISION MAKING

- ❖ Decision-making structures require that the programmer **specifies one or more conditions to be evaluated** or tested by the program,
- ❖ along with a statement or **statements to be executed if the condition is determined to be true**, and
- ❖ optionally, **other statements to be executed if the condition is determined to be false**.

Branching Statements



All of these operations can be carried out using various control statements included in C

C programming language provides the following types of decision-making statements.

Statement	Description
if statement	An if statement consists of a boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement , which executes when the Boolean expression is false.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).
switch statement	A switch statement allows a variable to be tested for equality against a list of values.
nested switch statements	You can use one switch statement inside another switch statement(s).

COMPUTER PROGRAMMING

**Conditional
Statements**

if Statement

if Statements

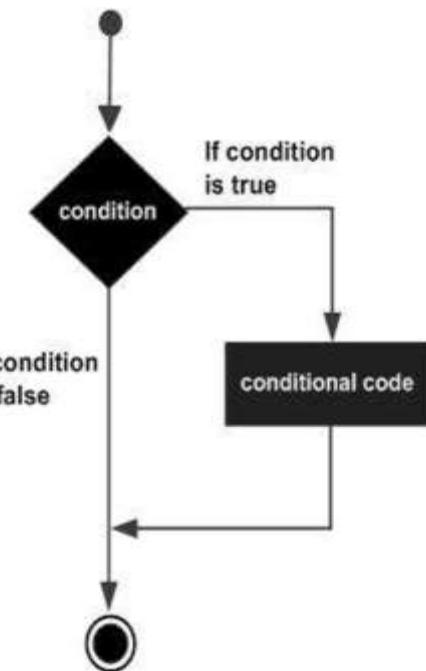
An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax

The syntax of an 'if' statement in C programming language is:

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
```

If the Boolean expression evaluates to **true**, then the block of code inside the 'if' statement will be executed. If the Boolean expression evaluates to **false**, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed.



if Statements

Example

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* check the boolean condition using if statement */
    if( a < 20 )
    {
        /* if condition is true then print the following */
        printf("a is less than 20\n" );
    }
    printf("value of a is : %d\n", a);

    return 0;
}
```

If..else Statements

if...else Statement

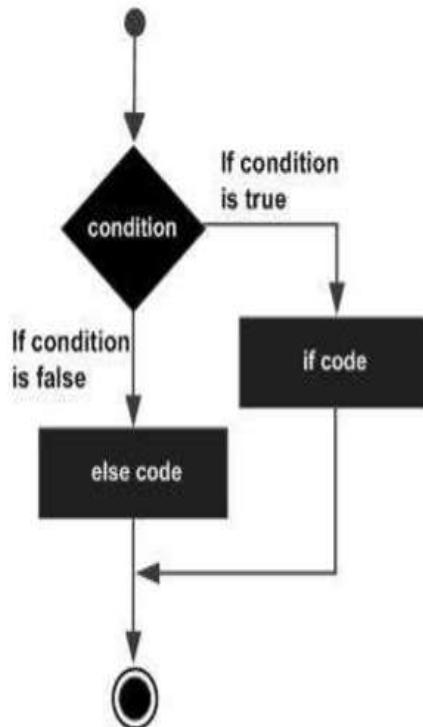
An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

Syntax

The syntax of an **if...else** statement in C programming language is:

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
else
{
    /* statement(s) will execute if the boolean expression is false */
}
```

If the Boolean expression evaluates to **true**, then the **if block** will be executed, otherwise, the **else block** will be executed.



....example

// IF statements

```
main()
{
    int a=10;
    if(a<20)
    {
        printf("a<20\n");
    }
    if(a>20)
    {
        printf("a>20\n");
    }
}
// {...} may be avoided for single statement under if-else
```

....example

// IF-ELSE statements

```
main()
{
    int a=10;
    if(a<20)
    {
        printf("a<20\n");
    }
    else
    {
        printf("a>20\n");
    }
}
```

// {...} may be avoided for single statement under if-else

....example

// Number is even or odd using if-else

```
#include <stdio.h>
int main(){
    int x;
    printf("enter the value of x:");
    scanf("%d", &x);
    if (x%2==0)
        printf("x is an even number\n");
    else
        printf("x is an odd number\n");
return 0;
}
```

COMPUTER PROGRAMMING

**Conditional
Statements**

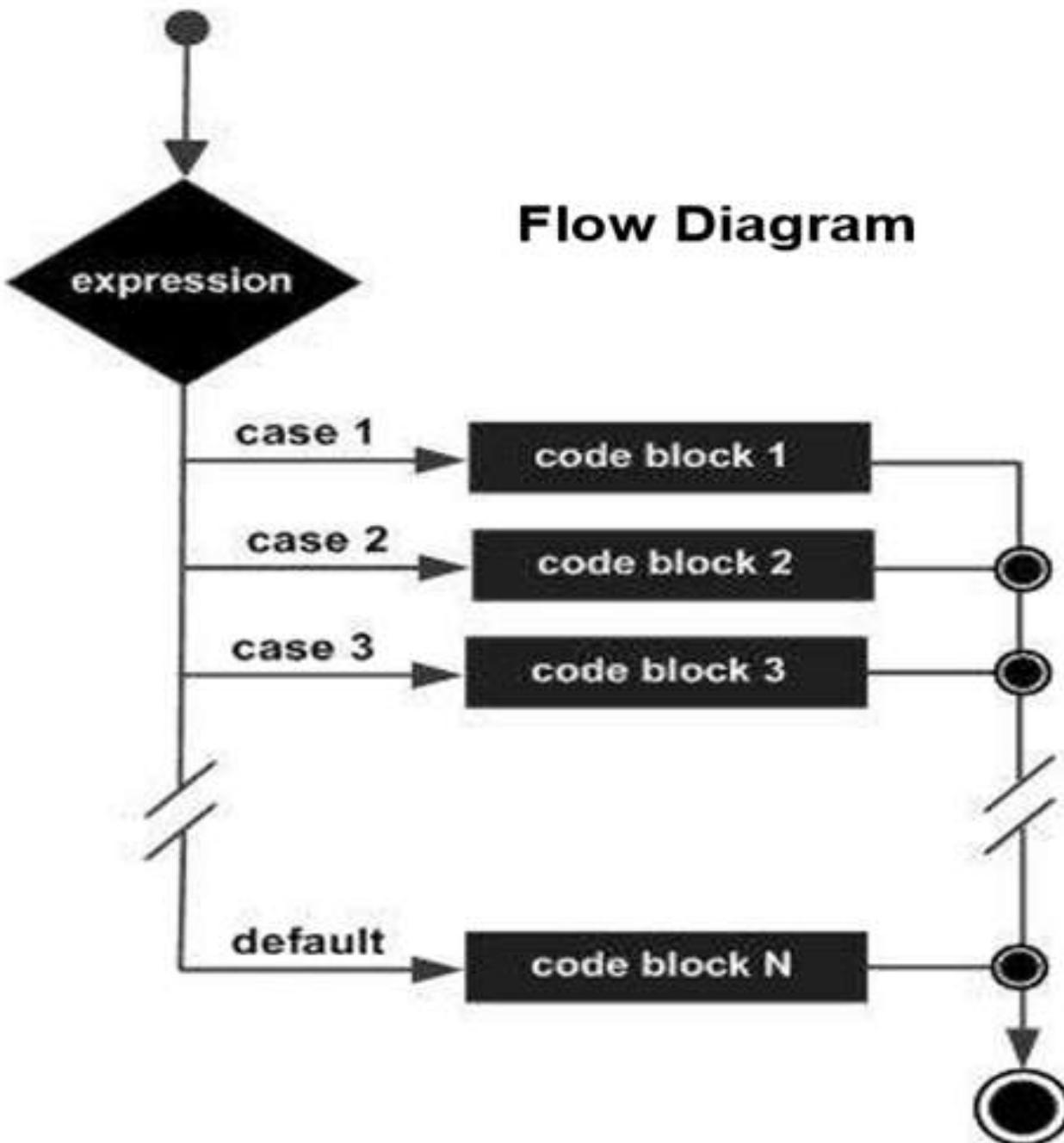
switch Statement

The control statement that allows us to make a decision from the number of choices is called a **switch**, or more correctly a **switch-case-default**, since these three keywords go together to make up the control statement. They most often appear as follows:

```
switch ( integer expression )
{
    case constant 1 : statements of case 1; break;
    case constant 2 : statements of case 2; break;
    ...
    default :         statements of default case ;
}
```

A switch allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

// The break; statement causes an immediate exit from the switch



Flow Diagram

switch Statement

Consider the following program:

```
main( )
{
    int i = 2 ;

    switch ( i )
    {
        case 1 :
            printf ( "I am in case 1 \n" );
        case 2 :
            printf ( "I am in case 2 \n" );
        case 3 :
            printf ( "I am in case 3 \n" );
        default :
            printf ( "I am in default \n" );
    }
}
```

The output of this program would be:

I am in case 2

I am in case 3

I am in default

switch Statement

The output is definitely not what we expected! We didn't expect the second and third line in the above output. The program prints case 2 and 3 and the default case. Well, yes. We said the **switch** executes the case where a match is found and all the subsequent **cases** and the **default** as well.

If you want that only case 2 should get executed, it is upto you to get out of the **switch** then and there by using a **break** statement. The following example shows how this is done. Note that there is no need for a **break** statement after the **default**, since the control comes out of the **switch** anyway.

switch Statement

- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which may appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

switch Statement

```
main( )
{
    int i = 2;

    switch ( i )
    {
        case 1 :
            printf ( "I am in case 1 \n" );
            break ;
        case 2 :
            printf ( "I am in case 2 \n" );
            break ;
        case 3 :
            printf ( "I am in case 3 \n" );
            break ;
        default :
            printf ( "I am in default \n" );
    }
}
```

The output of this program would be:
I am in case 2

switch Statement

```
main( )
{
    int i = 22 ;

    switch ( i )
    {
        case 121 :
            printf ( "I am in case 121 \n" ) ;
            break ;
        case 7 :
            printf ( "I am in case 7 \n" ) ;
            break ;
        case 22 :
            printf ( "I am in case 22 \n" ) ;
            break ;
        default :
            printf ( "I am in default \n" ) ;
    }
}
```

The output of this program would be:

switch Statement

```
char c = 'x';

switch ( c )
{
    case 'v' :
        printf ( "I am in case v \n" );
        break ;
    case 'a' :
        printf ( "I am in case a \n" );
        break ;
    case 'x' :
        printf ( "I am in case x \n" );
        break ;
    default :
        printf ( "I am in default \n" );
}
```

The output of this program would be:

I am in case x

switch vs if

Switch	If Else
The Expression is tested for equality only .	The expression can be tested for inequality as well. (<,>) This expression is evaluated to TRUE or FALSE
Only one value is used to match against all case labels.	Multiple expression can be tested for branching.
Switch case is not effective when checking for a range value.	If else is a better option to check ranges.
Switch case cannot handle floating point values	If else can handle floating point values
The case label must be constant(Characters of integers).	If else can use variables also for conditions.
Switch statement provides a better way to check a value against a number of constants .	If else is not suitable for this purpose .

COMPUTER PROGRAMMING

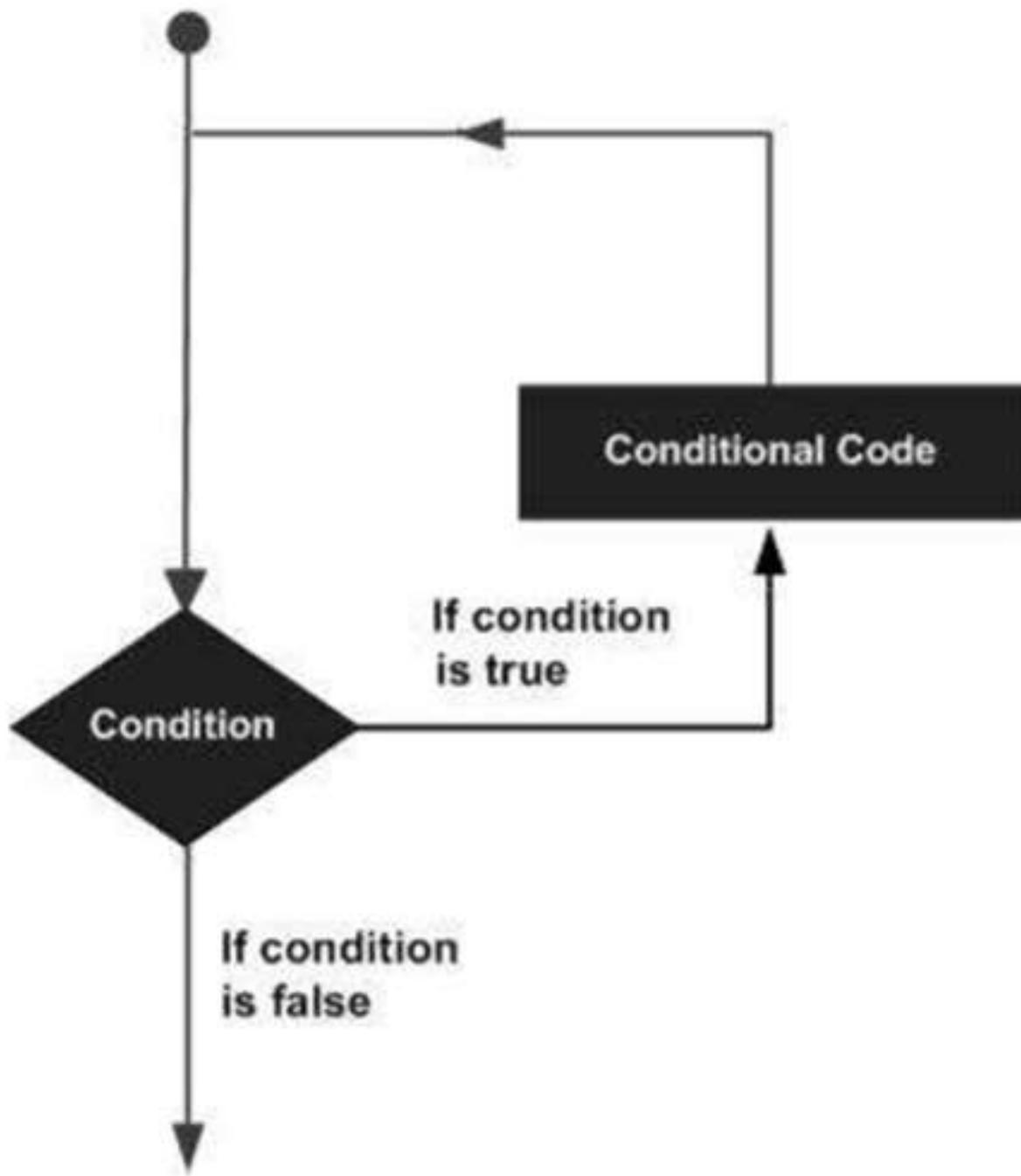
**Iterative
Statements**

Looping

Looping

- ❖ A program may require that a group of instructions be executed repeatedly, till the logical condition is true.
This is known as Looping.

- ❖ *Sometimes the required number of repetitions is known in advance; and sometimes the computation continues indefinitely until the logical condition becomes false.*



**Iteration statement
or Loop**

while

do-while

for

COMPUTER PROGRAMMING

**Iterative
Statements**

while loop

While Loop

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.

Syntax

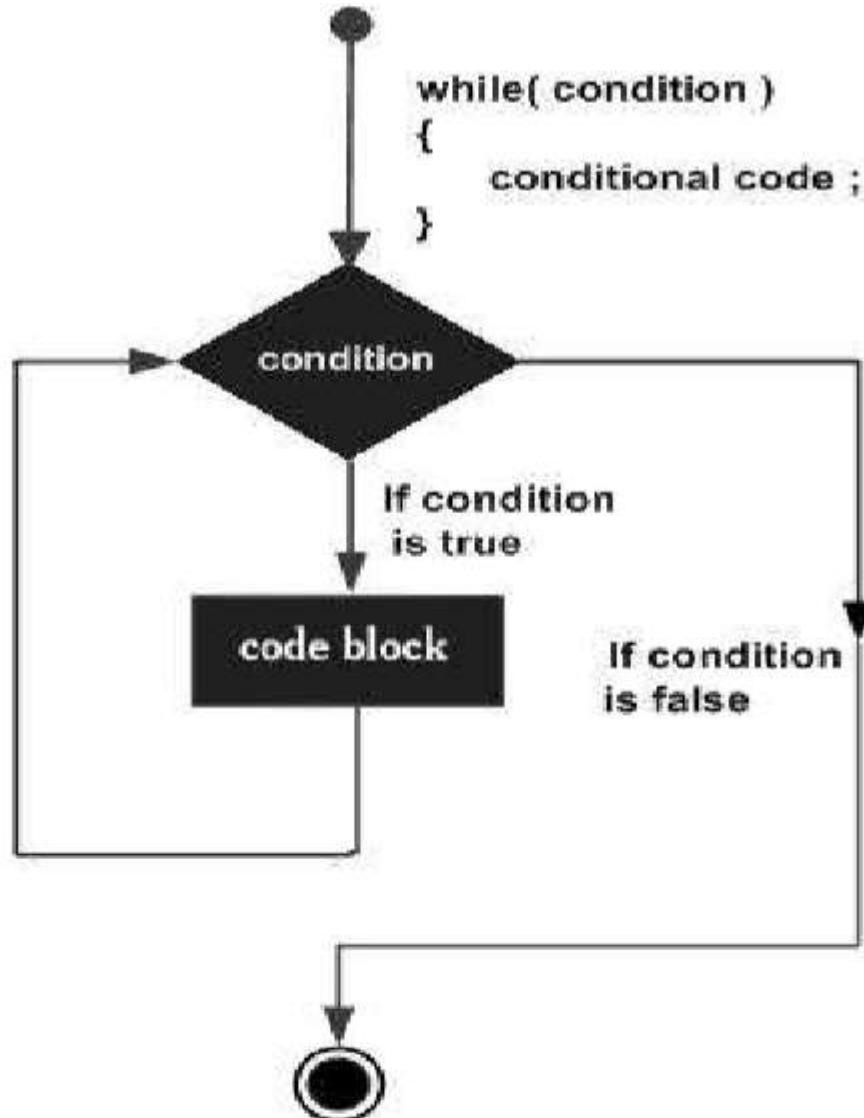
The syntax of a **while** loop in C programming language is:

```
while(condition)
{
    statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true if any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

While Loop



```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* while loop execution */
    while( a < 20 )
    {
        printf("value of a: %d\n", a);
        a++;
    }
    return 0;
}
```

While Loop

```
value of a: 10  
value of a: 11  
value of a: 12  
value of a: 13  
value of a: 14  
value of a: 15  
value of a: 16  
value of a: 17  
value of a: 18  
value of a: 19
```

While Loop

The condition being tested may use relational or logical operators as shown in the following examples:

```
while ( i <= 10 )
```

```
while ( i >= 10 && j <= 15 )
```

```
while ( j > 10 && ( b < 15 || c < 20 ) )
```

While Loop

```
/* Calculation of simple interest for 3 sets of p, n and r */
main( )
{
    int p, n, count ;
    float r, si ;

    count = 1 ;

    while ( count <= 3 )
    {
        printf ( "\nEnter values of p, n and r " ) ;
        scanf ( "%d %d %f", &p, &n, &r ) ;
        si = p * n * r / 100 ;
        printf ( "Simple interest = Rs. %f", si ) ;

        count = count + 1 ;
    }
}
```

While Loop

And here are a few sample runs...

Enter values of p, n and r 1000 5 13.5

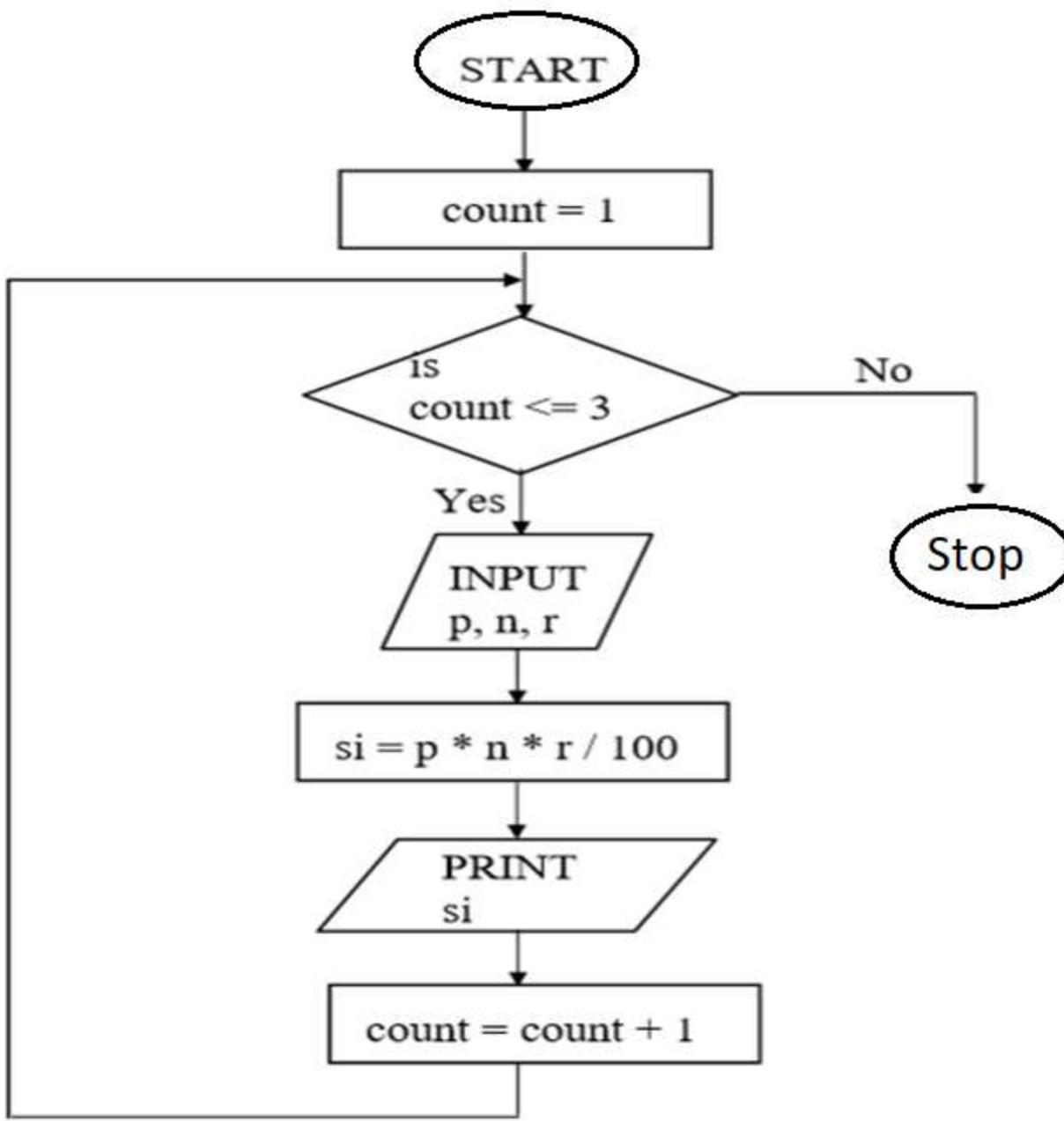
Simple interest = Rs. 675.000000

Enter values of p, n and r 2000 5 13.5

Simple interest = Rs. 1350.000000

Enter values of p, n and r 3500 5 3.5

Simple interest = Rs. 612.500000



While Loop

```
#include<stdio.h>

int main() {
    int i=1,n,f=1;
    printf("\n enter value of n=");
    scanf("%d", &n);
    while(i<=n){
        f = f*i;
        i++;
    }
    printf("Factorial of %d = %d",n, f);
}
```

COMPUTER PROGRAMMING

**Iterative
Statements**

for loop

for Loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for** loop in C programming language is:

```
for ( init; condition; increment )  
{  
    statement(s);  
}
```

Here is the flow of control in a 'for' loop:

1. The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.

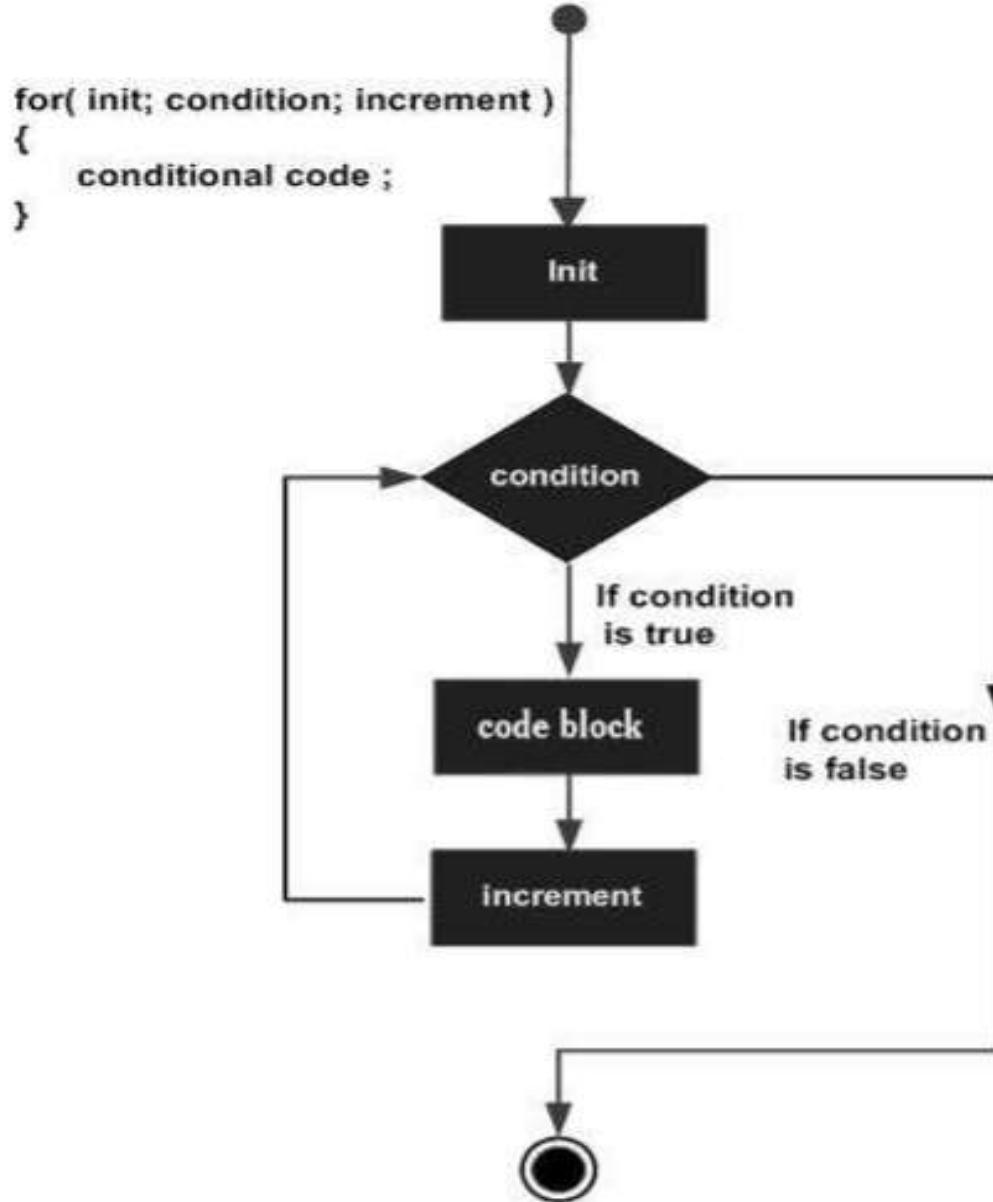
for Loop

3. After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

for Loop

Flow Diagram



for Loop

parenthesis

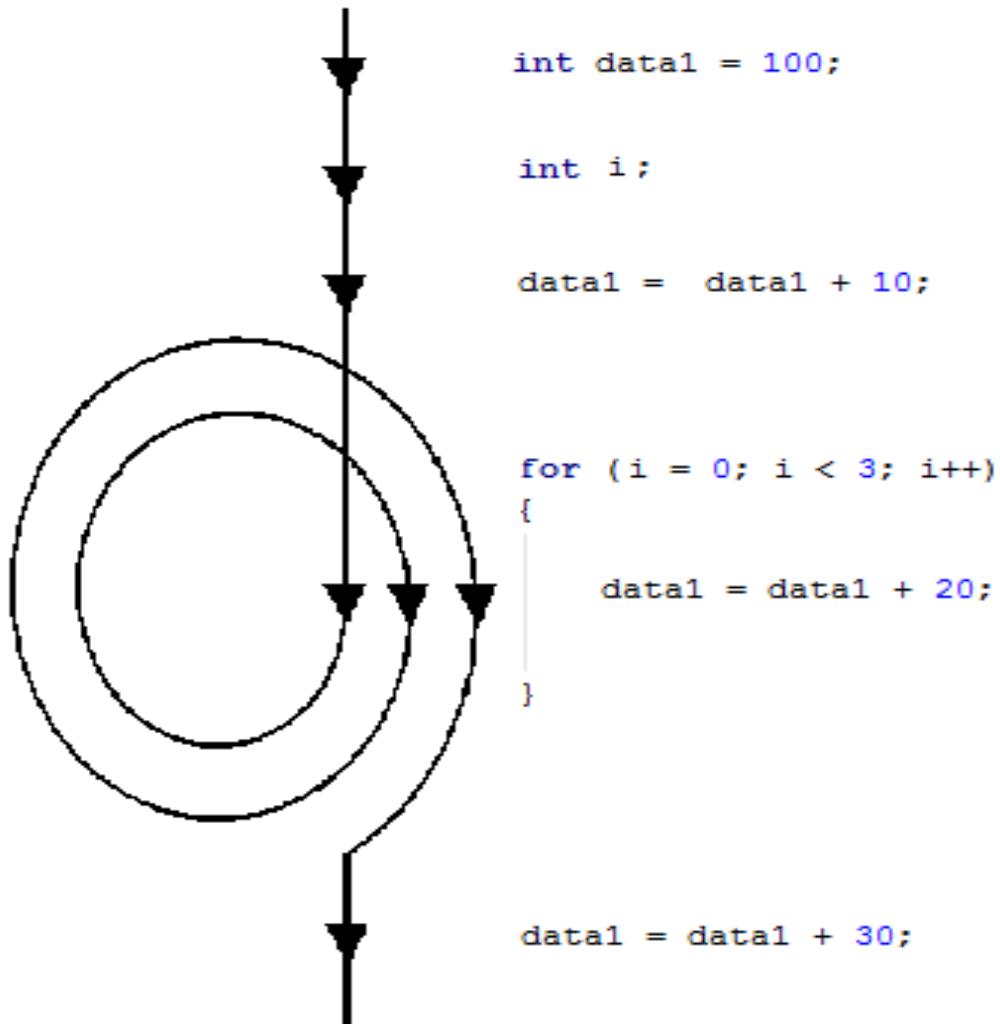
initialize

test

increment or
decrement

```
for( x = 0 ; x < 100 ; x++ ) {  
}
```

for Loop



for Loop

```
#include <stdio.h>

int main ()
{
    /* for loop execution */
    for( int a = 10; a < 20; a = a + 1 )
    {
        printf("value of a: %d\n", a);
    }
    return 0;
}
```

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

for Loop

Factorial using for loop

```
main()
{
int i,n=5,f=1;
for(i=1;i<=n;i++)
{
    f = f*i;
}
printf("Factorial is=%d", f);
}
```

COMPUTER PROGRAMMING

**Iterative
Statements**

do-while looping

do-while Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax

The syntax of a **do...while** loop in C programming language is:

```
do
{
    statement(s);

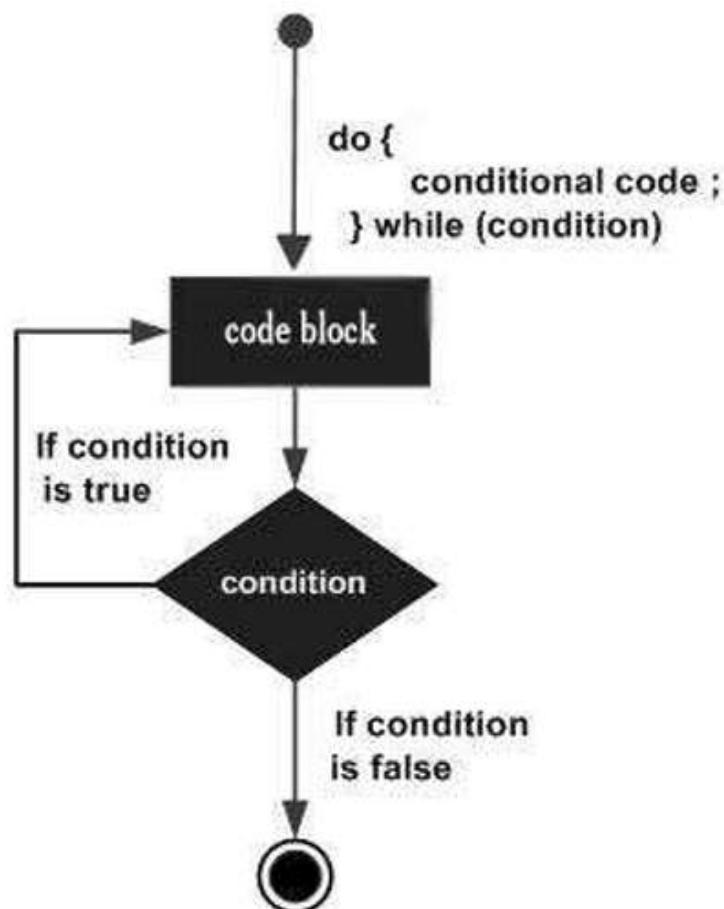
}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

do-while Loop

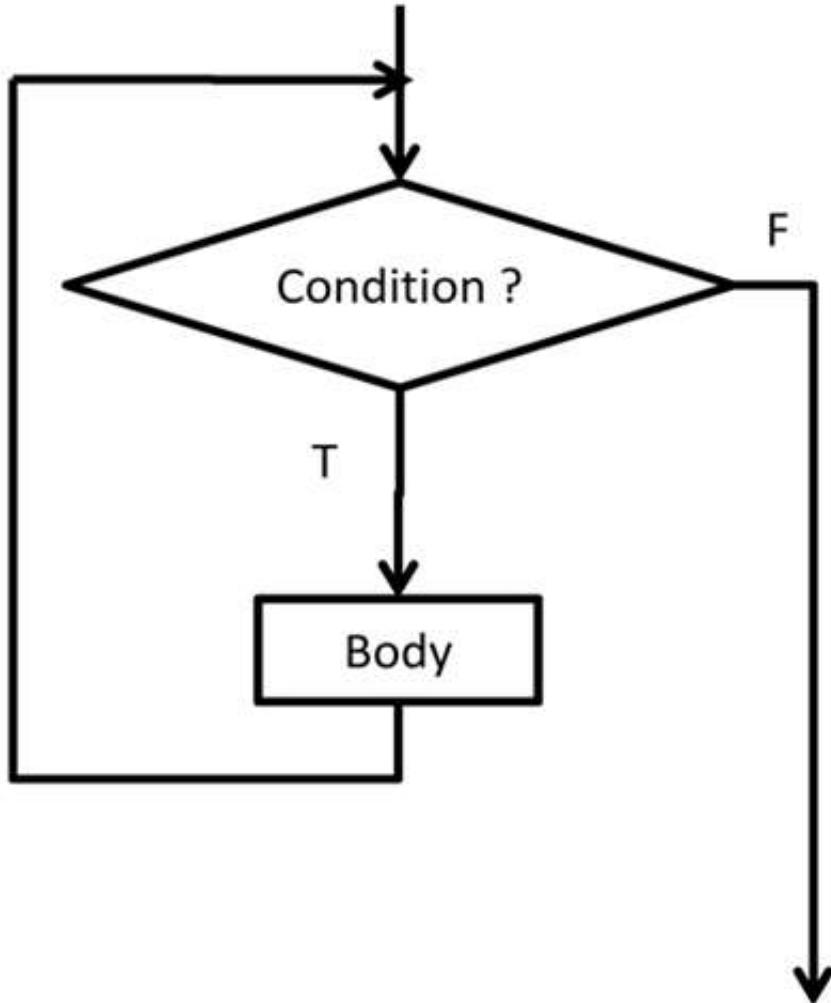
If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

Flow Diagram

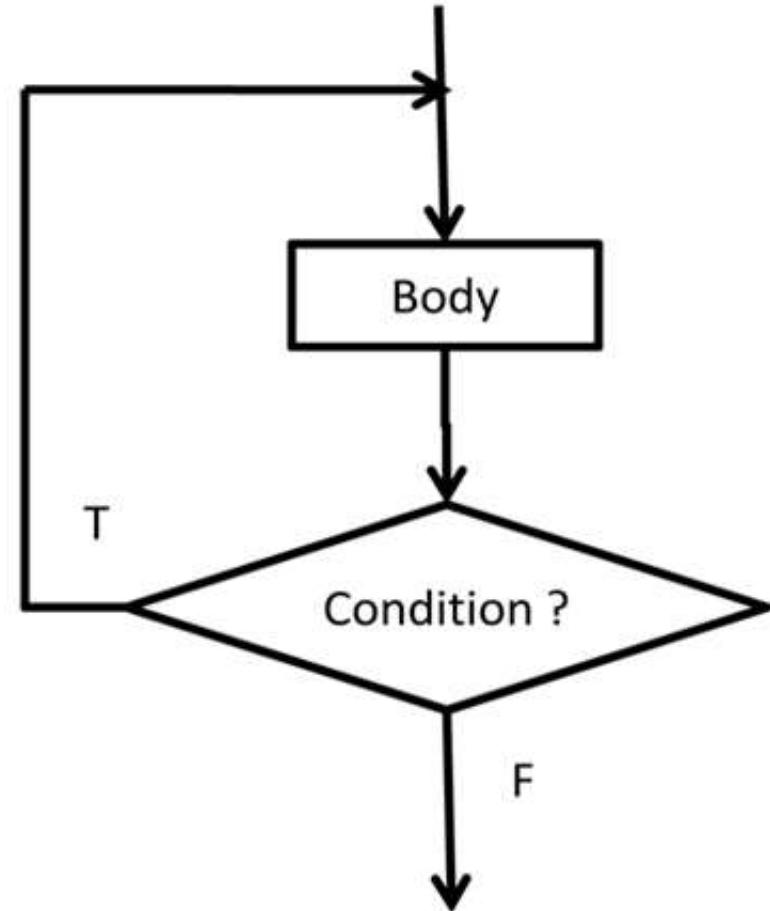


While vs do-while

```
while( condition )  
    body;
```



```
do {  
    body;  
} while( condition );
```



do-while Loop

Example

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* do loop execution */
    do
    {
        printf("value of a: %d\n", a);
        a = a + 1;
    }while( a < 20 );

    return 0;
}
```

do-while Loop

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

do-while Loop

Factorial using do-while

```
main()
{
int i=1,n=5,f=1;
do{
    f = f*i;
    i++;
} while(i<=n);      // semi colon is must
printf("Factorial is=%d",f);
}
```

COMPUTER PROGRAMMING

**Iterative
Statements**

Nested Loops

Nested Loops

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

The syntax for a **nested for loop** statement in C is as follows:

```
for ( init; condition; increment )
{
    for ( init; condition; increment )
    {
        statement(s);
    }
    statement(s);
}
```

Nested Loops

The syntax for a **nested while loop** statement in C programming language is as follows:

```
while(condition)
{
    while(condition)
    {
        statement(s);
    }
    statement(s);
}
```

Nested Loops

The syntax for a **nested do...while loop** statement in C programming language is as follows:

```
do
{
    statement(s);
    do
    {
        statement(s);
    }while( condition );
}
}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

Nested Loops

Example

```
#include <stdio.h>
int main()
{
    int i=1,j;
    while (i <= 5)
    {
        j=1;
        while (j <= i )
        {
            printf("%d ",j);
            j++;
        }
        printf("\n");
        i++;
    }
    return 0;
}
```

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

Nested Loops

```
#include <stdio.h>
int main()
{
    int i=1,j;
    do
    {
        j=1;
        do
        {
            printf("*");
            j++;
        }while(j <= i);
        i++;
        printf("\n");
    }while(i <= 5);
    return 0;
}
```

Example

```
*  
**  
***  
****  
*****
```

Nested Loops

```
#include<stdio.h>
main(){
int i,j,n=6;
for(i=1;i<=n;i++)
{
    for(j=1;j<=i;j++)
    {
        printf("*");
    }
    printf("\n");
}
}
```



The code demonstrates nested loops. The outer loop iterates from 1 to 6. The inner loop iterates from 1 to the current value of i. Each iteration of the inner loop prints an asterisk (*). After each row is printed, a new line character (\n) is printed to move to the next line.

*
**

Nested Loops

Practice

Write Programs for printing bellow output

*

* *

* * *

* * * *

* * * * *

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

* * * * *

* * * *

* * *

* *

*

1 2 3 4 5

2 3 4 5

3 4 5

4 5

5

COMPUTER PROGRAMMING

**Loop Control
Statements**

**break and continue
Statements**

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements.

Control Statement	Description
break statement	Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
goto statement	Transfers control to the labeled statement.

break Statement

The **break** statement in C programming has the following two usages:

- When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in C is as follows:

```
break;
```

break Statement

- ❖ In any loop **break** is used to jump out of loop skipping the code below it without caring about the test condition.
- ❖ It interrupts the flow of the program by breaking the loop and continues the execution of code which is outside the loop.
- ❖ The common use of break statement is in switch case where it is used to skip remaining part of the code.

break Statement

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

```
for (initialization; condition; modification)
```

```
{
    ....
    break ;
    ....
}
```

break Statement

```
for ( expression )
{
    statement1;
    ....
    if (condition) true
        break;
    ....
    statement2;
}
```

out of the loop

```
while (test condition)
{
    statement1;
    ....
    if (condition) true
        break;
    ....
    statement2;
}
```

out of the loop

break Statement

Example: C program to take input from the user until user enters zero.

```
#include <stdio.h>
int main ()
{
    int a;
    while (1)
    {
        printf("enter the number:");
        scanf("%d", &a);
        if ( a == 0 )
            break;
    }
    return 0;
}
```

```
#include <stdio.h>

int main () {
    int a = 10;

    /* while loop execution */
    while( a < 20 )
    {
        printf("value of a: %d\n", a);

        a++;

        if( a > 15 )
            /* terminate the loop using break statement */
            break;
    }
}
```

break Statement

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

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

```
#include <stdio.h>

int main()
{
    int n;
    printf("Enter the number of the day :");
    scanf(" %d ", &n);
    switch (n)
    {
        case 1:
            printf("Sunday");
            break;
        case 2:
            printf("Monday");
            break;
        case 3:
            printf("Tuesday");
            break;
    }
}
```

**Example: C
program to print
the day of the
week according to
the number of
days entered**

```
case 4:  
    printf("Wednesday");  
    break;  
  
case 5:  
    printf("Thrusday");  
    break;  
  
case 6:  
    intf("Friday");  
    break;  
  
case 7:  
    printf("Saturday");  
    break;  
  
default:  
    printf("You entered wrong day");  
    exit(0);  
}  
  
return 0;  
}
```

Continued...

continue Statement

The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control to pass to the conditional tests.

Syntax

The syntax for a **continue** statement in C is as follows:

```
continue;
```

continue Statement

- ❖ Like a **break** statement, **continue** statement is also used with **if** condition inside the loop to alter the flow of control.
- ❖ When used in **while**, **for** or **do...while** loop, it skips the remaining statements in the body of that loop and checks the condition for next iteration.
- ❖ Unlike **break** statement, **continue** statement when encountered doesn't terminate the loop, rather interrupts a particular iteration.

continue Statement

top of the loop

```
for ( expression )
{
    statement1;
    ...
    if (condition)
        continue;
    ...
    statement2;
}
```

top of the loop

```
while (test condition)
{
    statement1;
    ...
    if (condition)
        continue;
    ...
    statement2;
}
```

continue Statement

```
#include <stdio.h>

int main (){
    int a, sum = 0;
    for (a = 0; a < 10; a++) {
        if ( a % 2 == 0 )
            continue;
        sum = sum + a;
    }
    printf("sum = %d",sum);
    return 0; }
```

```
int main () {
    int a = 10;

    do {
        if( a == 15)  {

            /* skip the iteration */

            a = a + 1;

            continue;
        }

        printf("value of a: %d\n", a);

        a++;
    }while( a < 20 );

    return 0;
}
```

continue Statement

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

```
value of a: 10  
value of a: 11  
value of a: 12  
value of a: 13  
value of a: 14  
value of a: 16  
value of a: 17  
value of a: 18  
value of a: 19
```

COMPUTER PROGRAMMING

**Loop Control
Statements**

**goto
statements**

goto Statement

A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

NOTE: Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them.

Syntax

The syntax for a **goto** statement in C is as follows:

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

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

```
main( )
{
    int goals ;

    printf ( "Enter the number of goals scored against India" ) ;
    scanf ( "%d", &goals ) ;

    if ( goals <= 5 )
        goto sos ;
    else
    {
        printf ( "About time soccer players learnt C\n" ) ;
        printf ( "and said goodbye! adieu! to soccer" ) ;
        exit( ) ; /* terminates program execution */
    }

    sos :
    printf ( "To err is human!" ) ;
}
```

And here are two sample runs of the program...

Enter the number of goals scored against India 3

To err is human!

Enter the number of goals scored against India 7

About time soccer players learnt C

and said goodbye! adieu! to soccer

FUNCTIONS

Topics Covered

Functions: Declaration, Definition, Call and return, Call by value, Call by Reference, Scope of variables, Storage classes, Recursive functions, Recursion vs. Iteration.

Function

- Function is a self contained block of statements that perform a coherent task of some kind.
- Every C program can be thought of the collection of functions.
- `main()` is also a function.

Types of Functions

- Library functions
 - These are the in-built functions of ‘C’ library.
 - e.g. printf() ; is a function which is used to print an output. It is declared in ‘stdio.h’ file .
- User defined functions.
 - Programmers can create their own functions in C to perform specific task

Terms

- Actual arguments:- The arguments of calling function are actual arguments.
- Formal arguments:- Arguments of called function are formal arguments.
- Argument list:- Means variable name enclosed within the parenthesis. They must be separated by comma
- Return value:- It is the outcome of the function. The result obtained by the function is sent back to the calling function through the return statement. Function can return only a single value.

Need of Function

- Writing functions avoids rewriting of the same code again and again in the program.
- Function makes program structured.
- Using functions, large programs can be reduced to smaller ones. It is easy to debug and find out the errors in it.

Syntax

Function Declaration/Prototype

```
ret_type func_name (data_type par1,data_type par2...data_type parn);
```

Function Definition

```
ret_type func_name (data_type par1,data_type par2...data_type parn)
{
}
```

Function Call

```
func_name (par1, par2... parn);
```

Function declaration/prototype

- A prototype statement helps the compiler to check the return type and arguments type of the function.
- A prototype function consist of the functions return type, name and argument list.
- Example:

```
void sum( int x, int y);
```

```
#include <stdio.h>

void functionName()
{
    ...
}

int main()
{
    ...
    functionName();
    ...
}
```

How does function work?

Example

```
#include<stdio.h>
void sum(int, int); // function declaration/prototype
int main()
{
    int a, b;
    printf("enter the two no");
    scanf("%d %d", &a, &b);
    sum(a,b); // function calling
}
void sum( int x, int y) // function definition
{
    int c;
    c=x+y;
    printf ("sum is %d", c);
}
```

Example

```
#include<stdio.h>
int sum(int, int);
int main()
{
    int a=10,b=20, c;
    c=sum(a,b);           /*actual arguments
    printf("sum is %d", c);
}

int sum(int x, int y)      /*formal arguments
{
    int s;
    s=x+y;
    return(s);           /*return value
}
```

Categories of functions

- A function with no parameter and no return value
- A function no parameter and with return value
- A function with parameter and no return value
- A function with parameter and with return value

A function with no parameter and no return value

```
#include<stdio.h>
void print();           //func declaration
int main()
{
    printf ("no parameter and no return value");
    print();           //func calling

}
void print()           //func definition
{
    for(int i=1;i<=30;i++)
    {
        printf("*");
    }
    printf("\n");
}
```

A function with no parameter and no return value (Contd...)

- There is no data transfer between calling and called function
- The function is only executed and nothing is obtained
- Such functions may be used to print some messages, etc.

A function with parameters and no return value

```
#include<stdio.h>
void mul(int, int);
int main()
{
    int a=10,b=20;

    mul(a,b);           //actual arguments

}

void mul(int x, int y)      //formal arguments
{
    int s;
    s=x*y;
    printf ("mul is %d", s);
}
```

A function with no parameters and with return value

```
#include<stdio.h>
int sum();
int main()
{
    int c=sum();
    printf("sum is %d", c);
}
int sum()
{
    int x=10,y=30;
    return(x+y);      //return value
}
```

A function with parameter and with return value

```
#include<stdio.h>
int max(int, int);
int main()
{
    int a=10,b=20,c;
    c=max(a,b);
    printf ("greatest no is %d", c);
}
int max(int x, int y)
{
    if(x>y)
        return(x);
    else
        return(y);
}
```

- Discuss few more programs of functions....

Argument passing techniques

- Call By Value
- Call By Reference

Call By Value

- It is a default mechanism for argument passing.
- When an argument is passed by value then the copy of argument is made known as formal arguments which is stored at separate memory location
- Any changes made in the formal argument are not reflected back to actual argument, rather they remain local to the block which are lost once the control is returned back to calling program

Example

```
void swap(int,int);
int main()
{
int a=10,b=20;
printf("before function calling a =%d b= %d", a, b);
swap(a,b);
printf("after function calling a= %d b=%d", a, b);
return 0;
}
void swap(int x, int y)
{
int z;
z=x;
x=y;
y=z;
printf("Value of x is %d and y is %d ", x, y);
}
```

Output:

before function calling a=10 b=20
value of x is 20 and y is 10
after function calling a=10 b=20

Call By Reference

- In this instead of passing value, address are passed.
- Here formal arguments are pointers to the actual arguments
- Hence change made in the argument are permanent.
- The address of arguments is passed by preceding the address operator(&) with the name of the variable whose value you want to modify.
- The formal arguments are processed by asterisk (*) which acts as a pointer variable to store the addresses of the actual arguments

Example

```
void swap(int *,int *);  
int main()  
{  
int a=10 ,b=25;  
printf("before function calling a =%d b= %d", a, b);  
swap(&a, &b);  
printf("after function calling a= %d b= %d", a, b);  
return 0;  
}  
void swap(int *x, int *y)  
{  
int z;  
z=*x;  
*x=*y;  
*y=z;  
printf("Value of x is %d and y is %d ", *x, *y);  
}
```

Output:

before function calling a= 10 b= 25
value of x is 25 and y is 10
after function calling a=25 b= 10

- Call by value: This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- Call by reference: This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Scope of variables

- A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed.

There are two places where variables can be declared in C programming language –

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.

Local versus global

```
#include<stdio.h>
int g = 20;
int main () {
printf("g = %d\n",g); //global
int g = 10; //local
printf("g = %d\n",g); //local
}
```

Storage Classes in C

Storage Classes are used to describe the features of a variable.

These features basically include:

- (a) Where the variable would be stored.
- (b) What will be the initial value
- (c) What is the scope of the variable
- (d) How long would the variable exist.

Four Storage classes

- 1) auto
- 2) register
- 3) static
- 4) extern

auto

auto	
Storage	memory
Initial value	Garbage value
Scope	Local
Life	Till within the scope

auto – example 1

```
main()
{
    auto int i, j;
    printf("%d, %d", i, j);
}
```

Garbage value will be printed.

auto– example 2

```
main( ){
    auto int i = 1 ;
    {
        {
            printf ( "\n%d ", i ) ;
        }
        printf ( "%d ", i ) ;
    }
    printf ( "%d", i ) ;
}
```

auto– example 3

```
main( ){
    auto int i = 1 ;
{
    auto int i = 2 ;
{
    auto int i = 3 ;
        printf ( "\n%d ", i ) ;
}
    printf ( "%d ", i ) ;
}
printf ( "%d", i ) ;
}
```

register – fast but limited in number

REGISTER	
Storage	CPU Registers
Initial value	Garbage value
Scope	Local
Life	Till within the scope

register – example 1

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

static – value persists and shared among functions

STATIC	
Storage	memory
Initial value	zero
Scope	Local
Life	value persists among different functions

```
main( )
{
    increment( );
    increment( );
    increment( );
}

increment( )
{
    auto int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}
```

```
main( )
{
    increment( );
    increment( );
    increment( );
}

increment( )
{
    static int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}
```

The output of the above programs would be:

1
1
1

1
2
3

extern – global variables

EXTERN	
Storage	memory
Initial value	zero
Scope	global
Life	as long as the program

extern – example 1

```
int i ;
```

```
increment( ) {i++; printf ( "i = %d\n", i ) ; }
```

```
decrement( ) {i--; printf ( "i = %d\n", i ) ;}
```

```
main( ){  
printf ( "\n i = %d", i ) ;  
increment( ) ; increment( ) ;  
decrement( ) ; decrement( ) ;  
}
```

extern – 1 variable and 2 files

//file 1.c

```
#include<stdio.h>
int a;
void fun()
{
    a=a+2;
    printf("%d",a);
}
```

//file2.c

```
#include<stdio.h>
#include"file1.c"
int main()
{
    extern int a;
    a=7;
    fun();
}
```

```
#include<stdio.h>
main(){
    extern int i;
    printf("%d",i);
}
//Error – undefined integer i
```

Recursion

- Recursion is the process of repeating items in a self-similar way.
- In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion()
{ recursion(); /* function calls itself */
}
int main()
{
recursion();
}
```

Recursion (Contd...)

- The C programming language supports recursion, i.e., a function to call itself.
- But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.
- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Example

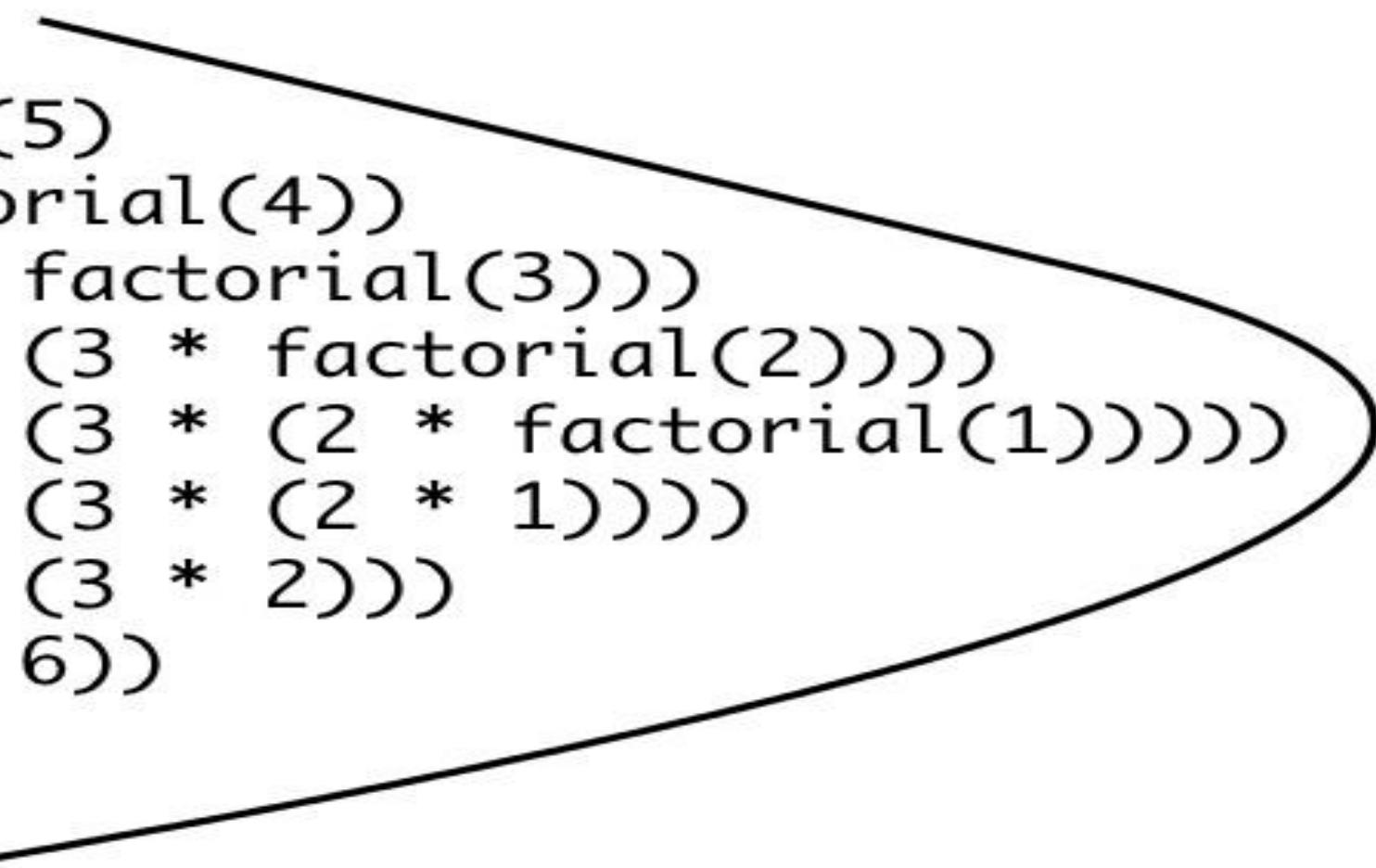
```
#include<stdio.h>
int fact(int);

int main(){
int n=5;
printf("Factorial = %d\n",fact(n));
}

int fact(int n){
if(n>1) return n*fact(n-1); //calling itself with n-1
else return 1;
}
```

Recursive

```
factorial(6)
6 * factorial(5)
6 * (5 * factorial(4))
6 * (5 * (4 * factorial(3)))
6 * (5 * (4 * (3 * factorial(2))))
6 * (5 * (4 * (3 * (2 * factorial(1)))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
720
```



Factorial(5)



return 5 * Factorial(4) = 120



return 4 * Factorial(3) = 24



return 3 * Factorial(2) = 6



return 2 * Factorial(1) = 2

1

- Discuss Dev C++ Debugging with the help of program

Recursion programs to try

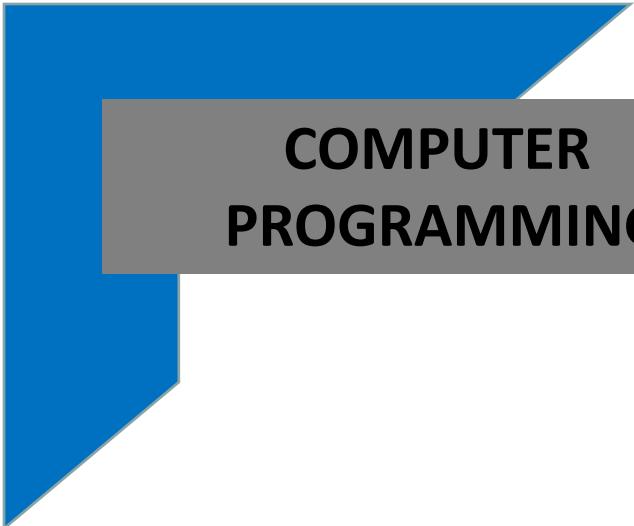
- Finding summation of n numbers
- Fibonacci series in which $a_0=0$, $a_1=1$, $a(n) = a(n-2) + a(n-1)$ i.e. any term is sum of previous two terms. So series = 0,1,1,2,3,5,8,13...
- Tower of Hanoi game <https://www.youtube.com/watch?v=7AUG7zXe-KU>

Iteration versus recursion

Iteration involves repetition of same structure. Recursion involves repetition through calling the function itself with different inputs.

Iteration needs less memory and time. Recursion consumes more memory and is slower but makes the program simpler.

Thank You



**COMPUTER
PROGRAMMING**



Arrays

Arrays

- ❖ An array is simply a collection of variables of the same data type that are referred to by a common name.
- ❖ A specific element in an array is accessed by an index.
- ❖ All array consist of contiguous memory locations where the lowest address corresponds to the first element whereas the highest address corresponds to the last element.

Arrays

- ❖ The **first element in the array is numbered 0**, so the **last element is 1 less than the size of the array.**
- ❖ An array is also known as a **subscripted variable**.
- ❖ Before using an array **its type and dimension must be declared.**
- ❖ Index always starts from **0** in an array.

Arrays

Types of Array :

1. One-dimensional arrays
2. Two-dimensional arrays
3. Multidimensional arrays

One-dimensional Arrays

An array of one dimension is known as a one-dimensional array or 1-D array

- A variable which represent the list of items using only **one index (subscript)** is called one-dimensional array.
- For Example , if we want to represent a set of five numbers say(35,40,20,57,19), by an array variable number, then number is declared as follows

```
int number [5] ;
```

One-dimensional Arrays

Like other variables an array needs to be declared so that the compiler will know what kind of an array and how large an array we want.

type arr_name[size];

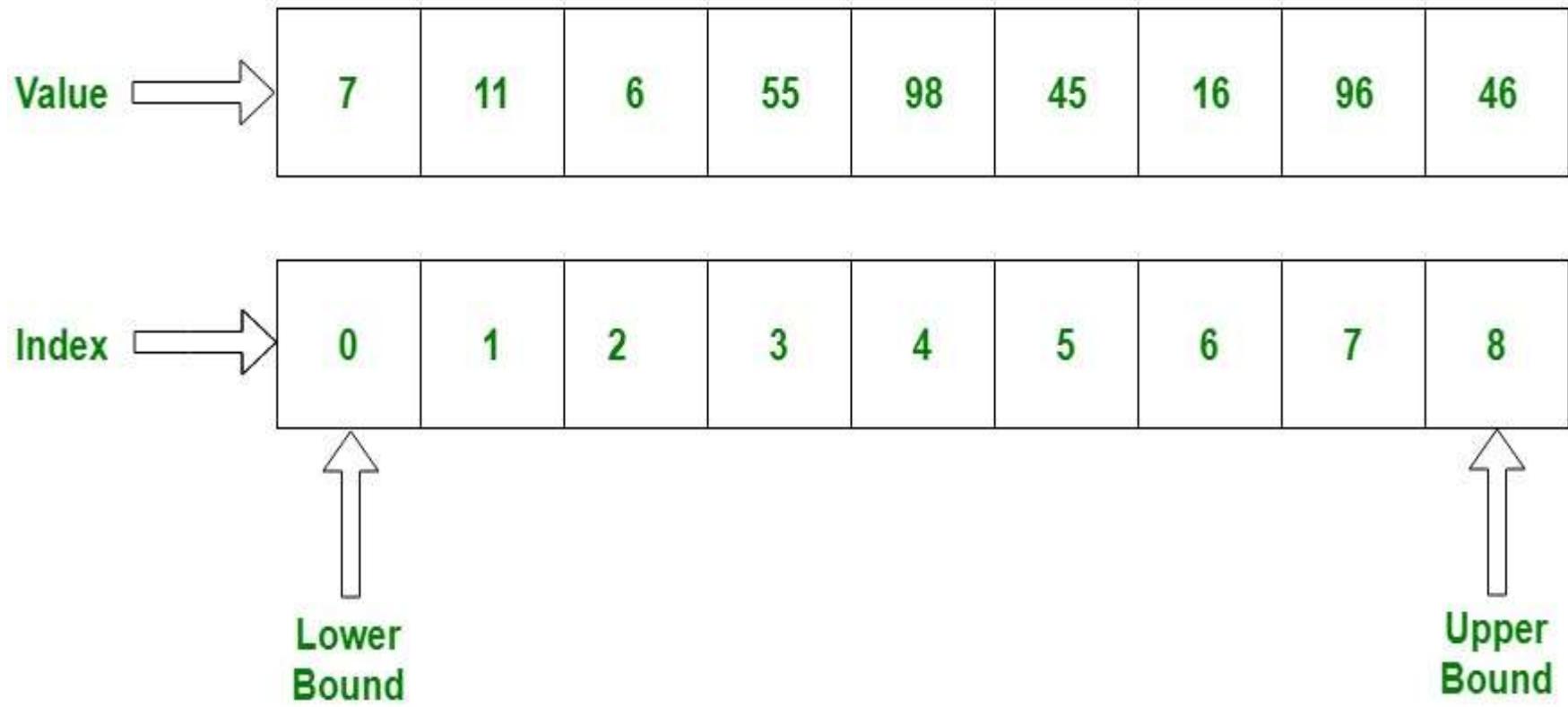
Here **type** is any valid data type,

arr_name is the name of the array you give

and **size** is the array size.

In other word, size specify that how many element, array can hold.

One-dimensional Arrays



Array Length = 9

One-dimensional Arrays

e.g.

```
int marks[30] ;
```

Here, *int* specifies the type of the variable,

The *number 30* tells how many elements of the type *int* will be in our array. This number is often called the "dimension" of the array.

The *bracket i.e. []* tells the compiler that we are dealing with an array.

Arrays

1 D ARRAY:

C	O	D	I	N	G	E	E	K
0	1	2	3	4	5	6	7	8

single row of elements

2 D ARRAY:

	j	0	1	2
i	0	A	A	A
row 0	1	B	B	B
row 1	2	C	C	C
row 2				

rows

column

array elements

One-dimensional Arrays

Initializing array in C

It is possible to initialize an array during declaration. For example,

```
1. int mark[5] = {19, 10, 8, 17, 9};
```

You can also initialize an array like this.

```
1. int mark[] = {19, 10, 8, 17, 9};
```

Here, we haven't specified the size. However, the compiler knows its size is 5 as we are initializing it with 5 elements.

mark[0] mark[1] mark[2] mark[3] mark[4]

19	10	8	17	9
----	----	---	----	---

One-dimensional Arrays

Initializing Array in C

```
int num[6] = { 2, 4, 12, 5, 45, 5 } ;
```

```
int n[ ] = { 2, 4, 12, 5, 45, 5 } ;
```

```
float press[ ] = { 12.3, 34.2, -23.4, -11.3 } ;
```

```
int b[ 100 ], x[ 27 ];
```

One-dimensional Arrays

Accessing Elements of an Array

- ❖ This is done **with subscript**, the number in the brackets following the array name.
- ❖ This number specifies the **element's position** in the array.
- ❖ All the array elements are numbered, **starting with 0**.
- ❖ Thus, **arr [2]** is not the second element of the array, **but the third**.

```
int arr[ ] = {1, 2, 3, 4, 5};
```

```
val = arr[2]; // val=3
```

One-dimensional Arrays

- Array elements are like normal variables

c[0] = 3;

printf("%d", c[0]);

- Perform operations in subscript. If x equals 3

c[5-2]==c[3]==c[x]

One-dimensional Arrays

1-D Array Input/Output

/* Program to take 5 values from the user and store them in an array & Print the elements stored in the array */

```
#include <stdio.h>
int main() {
    int values[5];
    printf("Enter 5 integers: ");

    // taking input and storing it in an array
    for(int i = 0; i < 5; ++i)
    {
        scanf("%d", &values[i]);
    }
}
```

One-dimensional Arrays

1-D Array Input/Output

```
// printing elements of an array
```

```
for(int i = 0; i < 5; ++i)
```

```
{
```

```
printf("%d\n", values[i]);
```

```
}
```

```
return 0;
```

```
}
```

One-dimensional Arrays

/*Compute the sum of the elements of the array */

```
#include <stdio.h>
int main( void )
{ /* use initializer list to initialize array */
int a[12] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
int i; /* counter */
int total = 0; /* sum of array */
/* sum of contents of array a */
for ( i = 0; i < SIZE; i++ )
{
    total += a[ i ];
} /* end for */
printf( "Total of array element values is %d\n", total );
return 0;
}
```

Output : Total of array element values is 383

One-dimensional Arrays

Passing a single array element to a function

```
#include<stdio.h>

void display(int a);

int main()
{
    int myArray[] = { 2, 3, 4 };
    display(myArray[2]);      //Passing array element myArray[2] only.
    return 0;
}

void display(int a)
{
    printf("%d", a);
}
```

OUTPUT

4

One-dimensional Arrays

Passing 1-d array to function using call by value method

```
#include <stdio.h>
int main()
{
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
'j'};
    for (int x=0; x<10; x++)
    {
        /* I'm passing each element one by one
using subscript*/
        disp (arr[x]);
    }

    return 0;
}
```

```
void disp( char ch)
{
    printf("%c ", ch);
}
```

OUTPUT:

a b c d e f g h i j

One-dimensional Arrays

Passing a 2D array as a parameter

```
#include<stdio.h>
void displayArray(int arr[3][3]);
int main()
{
    int arr[3][3], i, j;
    printf("Please enter 9 numbers for
the array: \n");
    for (i = 0; i < 3; ++i)
    {
        for (j = 0; j < 3; ++j)
        {
            scanf("%d", &arr[i][j]);
        }
    }
    // passing the array as argument
    displayArray(arr);
    return 0;
}
```

```
void displayArray(int arr[3][3])
{
    int i, j;
    printf("The complete array is: \n");
    for (i = 0; i < 3; ++i)
    {
        // getting cursor to new line
        printf("\n");
        for (j = 0; j < 3; ++j)
        {
            printf("%d\t", arr[i][j]);
        }
    }
}
```

OUTPUT:

```
Please enter 9 numbers for the array:
1
2
3
4
5
6
7
8
9
The complete array is:
1 2 3
4 5 6
7 8 9
```

Practice question:

Find maximum and minimum element in
an array

One-dimensional Arrays

\Find maximum and minimum element in an array

```
#include <stdio.h>
int main()
{
    int arr1[100];
    int i, max, min, n;
printf("Input the number of elements to be stored in the array :");
scanf("%d",&n);

printf("Input %d elements in the array :\n",n);
    for(i=0;i<n;i++)
    {
        printf("element - %d : ",i);
        scanf("%d",&arr1[i]);
    }
```

One-dimensional Arrays

\Find maximum and minimum element in an array

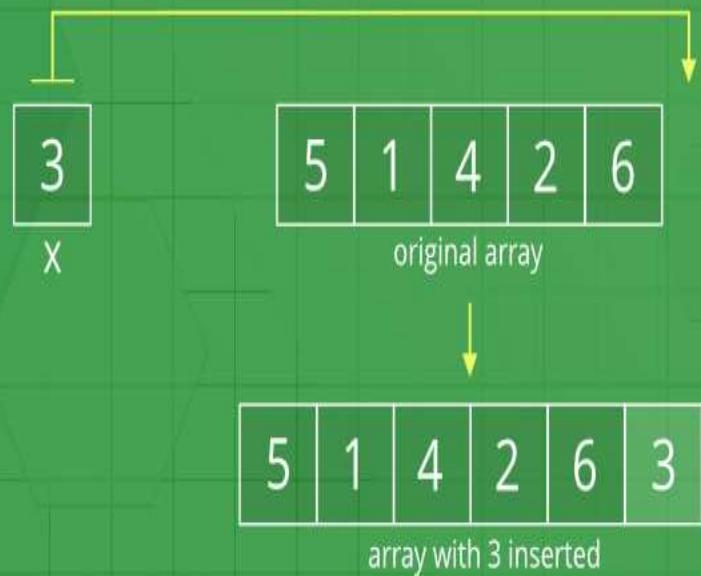
```
max = arr1[0];
min = arr1[0];
//finding max
for(i=1; i<n; i++)
{
if(arr1[i]>max)
{
max = arr1[i];
}
//finding min
if(arr1[i]<min)
{
min = arr1[i];
}
}
```

Practice question:

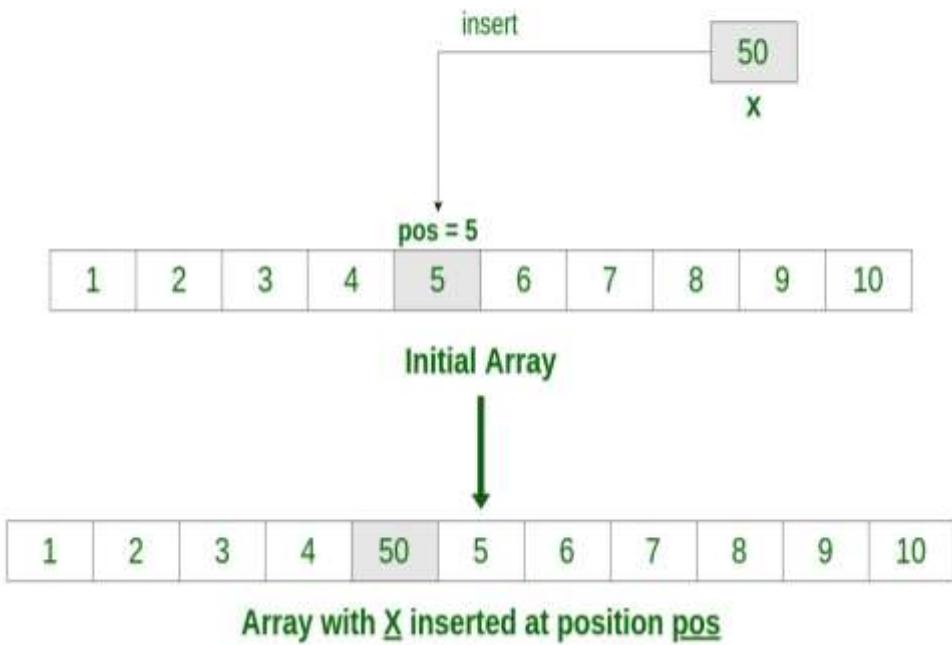
Insert an element in an Array

Practice question: Insert an element in an Array

Insert Operation in Unsorted Array



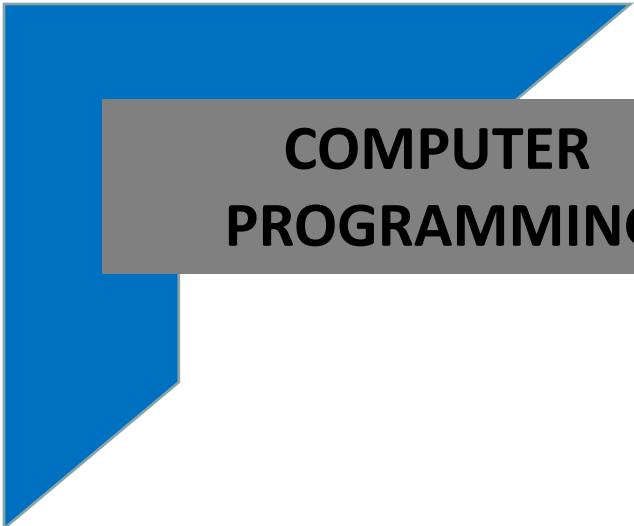
Insert an element at a specific position in an Array



```
// C Program to Insert an element at a  
//specific position in an Array
```

```
#include <stdio.h>  
int main()  
{  
    int arr[100];  
    int i, x, pos, n = 10;  
  
    // initial array of size 10  
    for (i = 0; i < 10; i++)  
        scanf("%d", &arr[i]);  
  
    // print the original array  
    for (i = 0; i < n; i++)  
        printf("%d ", arr[i]);  
    printf("\n");  
  
    // element to be inserted  
    x = 50;
```

```
// position at which element  
// is to be inserted  
pos = 5;  
  
// increase the size by 1  
n++;  
  
// shift elements forward  
for (i = n; i >= pos; i--)  
    arr[i] = arr[i - 1];  
  
// insert x at pos  
arr[pos - 1] = x;  
  
// print the updated array  
for (i = 0; i < n; i++)  
    printf("%d ", arr[i]);  
printf("\n");  
  
return 0;  
}
```



**COMPUTER
PROGRAMMING**



Multidimensional Arrays

Multiple-Subscripted Arrays

- **Multiple subscripted arrays**
 - Tables with rows and columns (m by n array)
 - Like matrices: specify row, then column
- **Declaration of two-dimensional array in C:**
`int arr[10][5];`
 - So the above array have 10 rows and 5 columns.

Two- Dimensional Arrays

Double-subscripted array (2-D) with three rows and four column.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Diagram illustrating the indexing of a 2D array:

- Column index: Points to the second column (Column 1).
- Row index: Points to the second row (Row 1).
- Array name: Points to the first element (a[0][0]).

Two- Dimensional Arrays

➤ Initialization

int arr[2][2] = { { 1, 2 }, { 3, 4 } };

– Initializers grouped by row in braces

```
int arr[2][2]={{1,2}, {3,4}};
```

```
arr[0][0]=1           arr[0][1]=2
```

```
arr[1][0]=3           arr[1][1]=4
```

Two- Dimensional Arrays

➤ Initialization

- If not enough, unspecified elements set to zero

```
int arr[2][2] = { {1}, {3, 4} };
```

➤ Referencing elements

- Specify row, then column

```
printf( "%d", arr[0][1] );
```

Two- Dimensional Arrays

```
/* Example- initializing and displaying elements*/
#include<stdio.h>
void main()
{
    int arr[10][5];
    int i, j;
// e.g. initializing 2-D array elements by 1
    for(i=0; i<10; i++)
    {
        for(j=0; j<5; j++)
        {
            arr[i][j] = 1;
        }
    }
}
```

Cont....

Two- Dimensional Arrays

...

```
// displaying 2-D array  
for(i=0; i<10; i++)  
{  
    for(j=0; j<5; j++)  
    {  
        printf("arr[%d][%d]=%d\t", i, j, arr[i][j]);  
    }  
    printf("\n");  
}  
}
```

Matrix

Entering and displaying Matrix elements

```
#include<stdio.h>
void main()
{
    int arr[5][3];
    int i, j;
    printf("Enter 5*3 Matrix: ");
    for(i=0; i<5; i++)
    {
        for(j=0; j<3; j++)
        {
            scanf("%d", &arr[i][j]);
        }
    }
}
```

Cont....

Cont....

//Displaying the Matrix

```
printf("\nThe Matrix is:\n");
for(i=0; i<5; i++)
{
    for(j=0; j<3; j++)
    {
        printf("%d\t", arr[i][j]);
    }
    printf("\n");
}
}
```

Matrix

$$A = \begin{pmatrix} 5 & 10 & 20 \\ 8 & 6 & 5 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 8 & 5 \\ 2 & 9 & 3 \end{pmatrix}$$

Addition of two Matrices

$$A + B = \begin{pmatrix} 5+3 & 10+8 & 20+5 \\ 8+2 & 6+9 & 5+3 \end{pmatrix} = \begin{pmatrix} 8 & 18 & 25 \\ 10 & 15 & 8 \end{pmatrix}$$

Matrix

Multiplication of two Matrices

$$\text{Matrix 1} \left\{ \begin{array}{ccc} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{array} \right\} \quad \text{Matrix 2} \left\{ \begin{array}{ccc} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{array} \right\}$$

$$\text{Matrix 1} \cdot \text{Matrix 2} \left\{ \begin{array}{ccc} 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 & 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 & 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 \\ 2 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 & 2 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 & 2 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 \\ 3 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 & 3 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 & 3 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 \end{array} \right\}$$

$$\text{Matrix 1} \cdot \text{Matrix 2} \left\{ \begin{array}{ccc} 6 & 6 & 6 \\ 12 & 12 & 12 \\ 18 & 18 & 18 \end{array} \right\}$$

Matrix

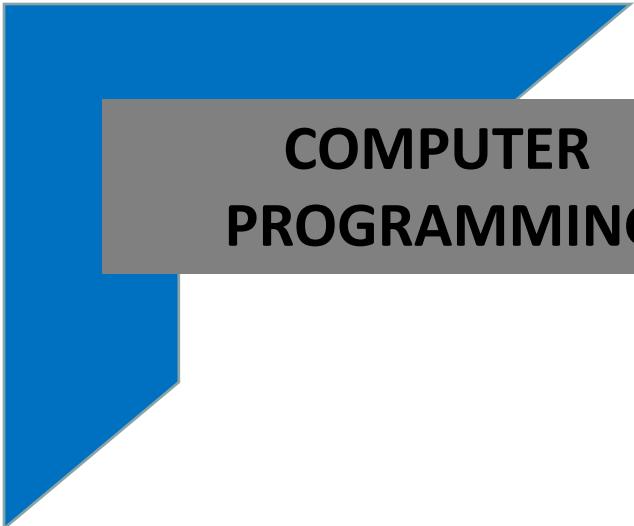
Transpose of a Matrix

ORIGINAL MATRIX

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

TRANSPOSE MATRIX

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$



**COMPUTER
PROGRAMMING**



Linear and Binary Search

Linear Search

Linear search

Search number **10**

List of numbers **2 | 1 | 5 | 10 | 12**

Step 1



Step 2



Step 3



Step 4



Linear Search

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the first element in the list.

Step 3 - If both are matched, then display "Given element is found!!!"
and terminate the function

Step 4 - If both are not matched, then compare search element with
the next element in the list.

Step 5 - Repeat steps 3 and 4 until search element is compared with
last element in the list.

Step 6 - If last element in the list also doesn't match, then display
"Element is not found!!!" and terminate the function.

Linear Search

```
#include <stdio.h>
int main()
{
    int array[100], search, c, n;
    printf("Enter number of elements in array\n");
    scanf("%d", &n);
    printf("Enter %d integer(s)\n", n);
//Entering array elements
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
//Enter number to be searched
    printf("Enter a number to search\n");
    scanf("%d", &search);
```

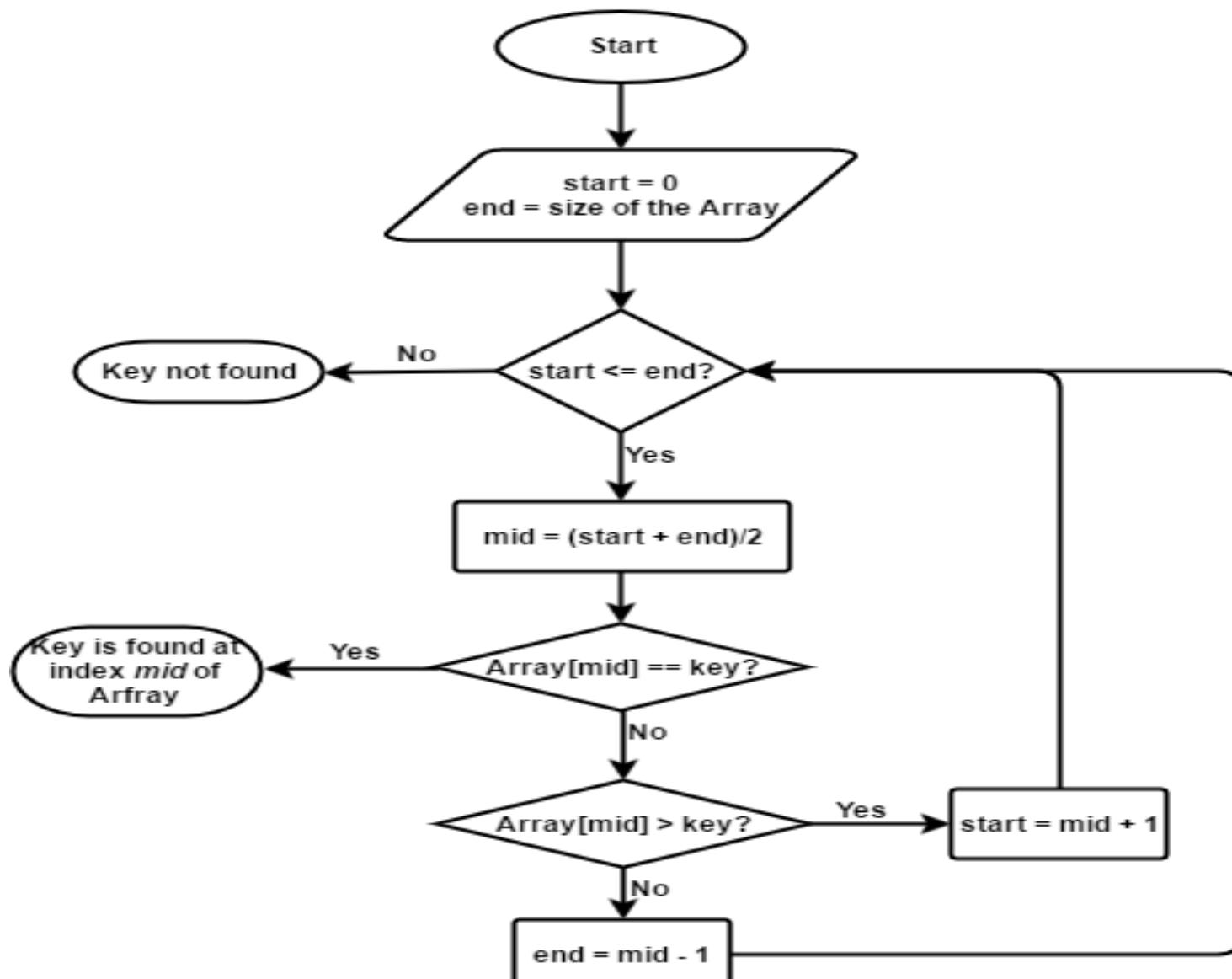
Linear Search

```
//Linear search
for (c = 0; c < n; c++)
{
    if (array[c] == search)
        // If required number is found
        { printf("%d is present at location %d.\n", search, c+1);
            break;
        }
}
// for not found
if (c == n)
    printf("%d isn't present in the array.\n", search);
return 0;
}
```

Binary Search

Binary Search

Binary search algorithm: find *key* in a sorted *Array*



Binary Search

```
/* C Program - Binary Search */
#include<stdio.h>
void main()
{
    int n, i, arr[50], search, first, last, middle;
    printf("Enter total number of elements :");
    scanf("%d",&n);
    printf("Enter %d number :", n);
    for (i=0; i<n; i++)
    {
        scanf("%d",&arr[i]);
    }
    printf("Enter a number to find :");
    scanf("%d", &search);
    first = 0;
    last = n-1;
    middle = (first+last)/2;
```

Cont....

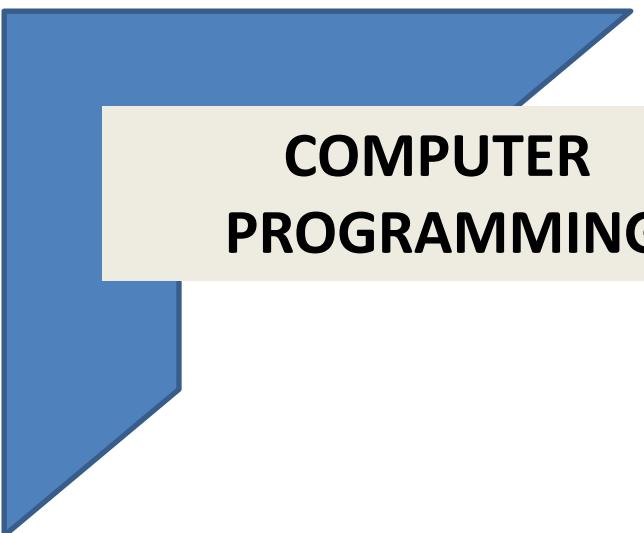
```
/* C Program - Binary Search */
while (first <= last)
{
    if(arr[middle] < search)
    {
        first = middle + 1;
    }
    else if(arr[middle] == search)
    {
        printf("%d found at location %d\n", search,
middle+1);
        break;
    }
}
```

Cont....

```
else
{
    last = middle - 1;
}
middle = (first + last)/2;
}
if(first > last)
{
printf("Not found! %d is not present in the list.",search);
}
}
```

Practice questions:

- Deletion of an array element
- Merging two arrays



COMPUTER PROGRAMMING

Pointers and Strings

Topics to be covered

Reading, writing and manipulating Strings,
Understanding computer memory, accessing via
pointers, pointers to arrays, dynamic
allocation, drawback of pointers.

String definition

- String is a 1-d array of characters **terminated by a null ('\0')**
- `char C[] = {'A','B','C','\0'};`
- `char S[] = "ABC";`
- Both C and S have length = 3
- You can also have **NULL** instead of '\0' defined in stdio.h and string.h

String input including blanks

```
main(){  
char name[25] ;  
printf ( "Enter name: " ) ;  
scanf ( "%s", name ) ;  
printf("hello %s",name);  
}
```

Another way

```
main(){  
char C[20];  
gets(C); // accepts blank spaces  
puts(C);  
}
```

NULL and strlen()

```
main(){  
int i=0;  
char C[10];  
gets(C);  
while(C[i]!=NULL) printf("%c",C[i++]);  
//OR while(i<strlen(C)) printf("%c",C[i++]);  
}
```

Few Inbuilt functions

```
main( ) {  
char A[] = "wxYZ", B[]={},C[20];  
printf("length of %s = %d",A,strlen(A));  
if(strcmp(A,B)==0) printf("\nA,B same strings");  
printf("\nC = %s",strcpy(C,A));  
printf("\nUppercase of A = %s",strupr(A));  
printf("\nConcat A+C = %s",strcat(C,A));  
printf("\n Reverse A = %s",strrev(A));  
}
```

2-d character array

```
char Names[6][10] = {"akshay",
"parag",
"raman"};
```

```
puts(Names[0]); //akshay
```

char Names[6][10] storage

65454	a	k	s	h	a	y	\0			
65464	p	a	r	a	g	\0				
65474	r	a	m	a	n	\0				
65484	s	r	i	n	i	v	a	s	\0	
65494	g	o	p	a	l	\0				
65504	r	a	j	e	s	h	\0			

65513
(last location)

- Reading, writing and manipulating Strings,
Understanding computer
memory, accessing via pointers,
pointers to arrays, dynamic
allocation, drawback of
pointers.

What is a pointer

Sweet Home



- Own Property
- Fixed space

Smart Hotel



- ✓ Leased Property
- ✓ Space can be increased dynamically

Variables

vs.

pointers

Advantages of pointers

- Dynamic memory allocation
- System level programming
- To modify a variable in another function
- Illusion of multiple returns
- Passing and returning arrays

Variable, value and location

i → location name



A diagram illustrating memory organization. A variable 'i' is shown pointing to a memory location. The memory location is represented by a rectangular box containing the value '3'. A horizontal arrow points from the variable 'i' to the memory location, and another arrow points from the memory location to the text 'value at location'.

3 → value at location

65524 → location number

& - address of operator

```
main( ){  
int i = 3 ;  
printf ( "\nAddress of i = %u", &i ) ;  
printf ( "\nValue of i = %d", i ) ;  
}
```

//%u means unsigned integer, what is %p,%i,%e

* is value at address

```
main( ) {  
int i = 3 ;  
printf ( "\nAddress of i = %u", &i ) ;  
printf ( "\nValue of i = %d", i ) ;  
printf ( "\nValue of i = %d", *( &i ) ) ;  
}  
  
// output: some address, 3 and 3
```

```
int *j ;
```

- This declaration tells the compiler that j will be used to store the *address* of an integer value.

Problem 1

```
main( ) {  
int i = 3 ,*j ;  
j = &i ;  
printf ( "\nAddr of i = %u", &i ) ;  
printf ( "\nAddr of i = %u", j ) ;  
printf ( "\nAddr of j = %u", &j ) ;  
printf ( "\n j = %u", j ) ; printf ( "\n i = %d", i ) ;  
printf ( "\n i = %d", *( &i ) ) ; printf ( "\n i = %d", *j ) ;  
}
```

Answer

Addr of i = 65524

Addr of i = 65524

Addr of j = 65522

j = 65524

i = 3

i = 3

Value of i = 3

Pointers for other datatypes

- int *alpha ;
- char *ch ;
- float *s ;

Pointers are variables that contain addresses,
and since addresses are whole numbers,
**pointers would always contain whole
numbers.**

```
char *ch ;
```

- *ch will contain a character
- ch is an unsigned int containing the address

Problem 2

```
main(){  
int i=3,*j,**k;  
j=&i; k=&j;  
printf("\naddr of i=%u",&i);  
printf("\naddr of i=%u",j);  
printf("\naddr of i=%u",*k);  
printf("\naddr of j=%u",&j);  
printf("\naddr of j=%u",k);  
printf("\naddr of k=%u",&k);  
printf("\nj=%u",j); printf("\nk=%u",k);  
printf("\ni=%d",i); printf("\ni=%d",*(&i));  
printf("\ni=%d",*j); printf("\ni=%d",**k);  
}
```

Situation

i

3

65524

j

65524

65522

k

65522

65520

Answer

addr of i=6487628

addr of i=6487628

addr of i=6487628

addr of j=6487616

addr of j=6487616

addr of k=6487608

j=6487628

k=6487616

i=3

i=3

i=3

i=3

Problem 3

```
main() {  
    int Var = 10;  
    int *ptr = &Var;  
    printf("Var = %d\n", *ptr);  
    printf("Var Addr = %u\n", ptr);  
    *ptr = 20;  
    printf("Var = %d\n", Var);  
}
```

Answer

Var = 10

Var Addr = 6487620

Var = 20

Problem 4

```
main() {  
    int v[3] = {10, 100, 200}, i;  
    int *ptr;  
    ptr = v;  
    for(i = 0; i < 3; i++) {  
        printf("*ptr = %d\n", *ptr);  
        printf("ptr = %u\n", ptr);  
        ptr++;  
    }  
}
```

Answer

*ptr = 10

ptr = 6487600

*ptr = 100

ptr = 6487604

*ptr = 200

ptr = 6487608

Problem 5

```
main( ) {  
int i=3,*x; float j=1.5,*y; char k='c',*z ;  
printf("\n i=%d, j=%f, k=%c",i,j,k) ;  
x = &i ; y = &j ; z = &k ;  
printf("\n Addr of x,y,z = %u, %u, %u",x,y,z);  
x++; y++; z++;  
printf("\n New Addr of x,y,z = %u, %u, %u",x,y,z);  
}
```

Answer

- i=3, j=1.500000, k=c
- Addr of x,y,z = 6487604, 6487600, 6487599
- New Addr of x,y,z = 6487608, 6487604,
6487600

Problem 6

```
main( ) {  
int arr[] = { 10, 20, 30, 45, 67, 56, 74 };  
int *i, *j;           *i i           *j j  
i = &arr[1]; j = &arr[5];  
printf("%d %d", j-i, *j-*i);  
}
```

Answer

- 4 36

Problem 7

```
main(){  
int arr[]={10,20,36,72,45,36};  
int*j,*k;  
j=&arr[4];  
k=(arr+4);  
if(j==k)  
printf("Same locations");  
else printf("Different locations");  
}
```

Answer

- Same locations

Problem 8

```
main(){  
int a = 10, b = 20 ;  
swapv ( a, b ) ;  
printf ( "\na = %d b = %d", a, b ) ;  
}
```

sid

```
swapv ( int x, int y ) {  
int t = x ;  
x = y ; y = t ;  
printf ( "\nx = %d y = %d", x, y ) ;  
}
```

Because
Program
not get
the value
of a and
b

Answer

$x = 20$ $y = 10$

$a = 10$ $b = 20$

Problem 9: Pointer swap

```
main( ) {  
    int a = 10, b = 20 ;  
    swapr ( &a, &b ) ;  
    printf ( "\na = %d, b = %d", a, b ) ;  
}  
  
swapr( int *x, int *y ) {  
    int t = *x ; *x = *y ; *y = t ;  
}
```

Problem 10: One * for 1-d

```
main( ) {  
int a[3]={1,2,3},*b,i;  
b=a;  
for(i=0;i<3;i++) {  
printf("\n %d %d",a[i],b[i]);  
}  
}
```

Output

- 1 1
- 2 2
- 3 3

Problem 11: Multiple returns

```
AreaPeri(int r, float *A, float *P){  
    *A = 3.14*r*r;  *P = 2*3.14*r;  
}  
  
main( ){  
    int r;  float A,P;  
    printf("\nEnter radius: " );  scanf("%d",&r);  
    AreaPeri(r,&A,&P);  printf("Area = %f, Peri = %f",A,P);  
}
```

Same things

- $\text{name}[i]$
- $*(\text{name} + i)$
- $*(i + \text{name})$
- $i[\text{name}]$

Problem 12: Array vs *

```
main( ){  
char A[] = "Hello", B[10] ;  
char *C = "Good Morning", *D;  
//B = A ; /* error */  
D = C ; /* Okay */  
printf("%s",D);  
} // cannot assign array to array, but can assign  
pointer to pointer
```

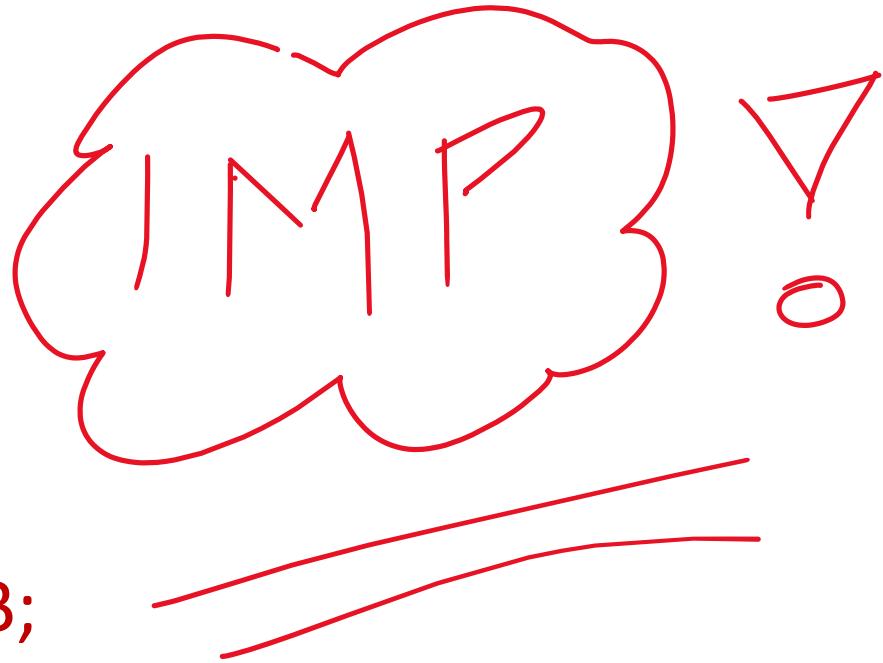
Problem 13: * are dynamic

```
main( ) {  
    char A[] = "Hello" ;  
    char *B = "Hello" ;  
    //A = "Bye" ; /* error */  
    B = "Bye" ; /* works */  
    printf("%s",B);  
}
```

Problem 14: Returning an array

```
int *incr(int X[3],int k){  
    int i;  
    for(i=0;i<k;i++) ++X[i];  
    return X;  
}
```

```
main(){  
    int A[] = {1,2,3},n=3,i,*B;  
    B = incr(A,n);  
    for(i=0;i<n;i++) printf(" %d",B[i]);  
}
```



Output

- 2 3 4

Array of pointers

- `char*names[]{"akshay","parag","raman","sri nivas", "gopal","rajesh"};`
- It contains base address of various names

Array of pointers

akshay\0

182

raman\0

195

srinivas\0

201

gopal\0

210

rajesh\0

216

parag\0

189

names[]

182	189	195	201	210	216
-----	-----	-----	-----	-----	-----

65514

65516

65518

65520

65522

65524

Problem 15: Dynamic allocation

```
main(){  
    int *ptr, n,i;  
    printf("Size of array? "); scanf("%d",&n);  
    ptr = (int*)malloc(n*sizeof(int));  
    for(i=0;i<n;i++) ptr[i]=i;  
    for(i=0;i<n;i++) printf(" %d",ptr[i]);  
    free (ptr);  
}
```

Output

Size of array? 10

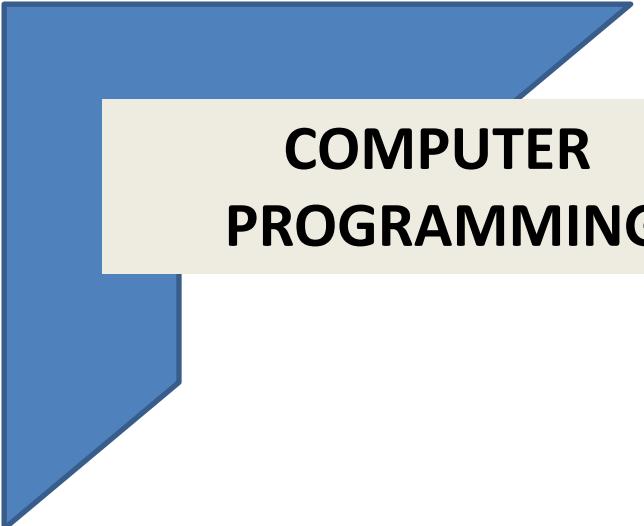
0 1 2 3 4 5 6 7 8 9

Drawback of pointers

- Complicated to use and debug
- Uninitialized pointers might cause segmentation fault
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to **memory leak** (occupied but unused memory)
- If pointers are updated with incorrect values, it might lead to **memory corruption**

Topics covered were

Reading, writing and manipulating **Strings**,
Understanding computer memory,
accessing via pointers, **pointers** to arrays,
dynamic allocation, drawback of pointers.



COMPUTER PROGRAMMING

Structures and Union

Topics to be covered

- ❑ Defining a Structure
- ❑ Declaring a structure variables
- ❑ Accessing Structure Elements
- ❑ Union

Need of Structure

- Problem:
 - How to group together a collection of data items of different types that are logically related to a particular entity??? (~~Array~~)

*Solution: **Structure***

Structure

- A structure is a collection of variables of different data types under a single name.
- The variables are called **members** of the structure.
- The structure is also called a user-defined data type.

Defining a Structure

```
struct structure_name {  
    member 1;  
    member 2;  
    :  
    member m;  
};
```

- `struct` is the required C keyword
- `structure_name` is the name of the structure
- `member 1`, `member 2`, ... are individual member declarations
- *The members of a structure do not occupy memory until they are associated with a `structure_variable`.*

Structure variables

- Once a structure has been defined, the individual structure-type variables can be declared as:

```
struct structure_name var_1, var_2, ..., var_n;
```

Example

- A structure definition

```
struct item {  
    char name[30];  
    int itemcode;  
    float price;  
};
```

- Defining structure variables:

```
struct item a1, a2, a3;
```



A new data-type

Another way of declaring structure variables

- It is possible to combine the declaration of the structure with that of the structure variables:

```
struct item
{
    member 1;
    member 2;
    :
    member m;
} var_1, var_2,..., var_n;
```

- Declares three variables of type **struct item**

Accessing Structure Elements

- The member variables are accessed using the dot (.) operator or member operator.
- A structure member can be accessed by writing
`variable.member`
where `variable` refers to the name of a structure-type variable, and `member` refers to the name of a member within the structure.
- Examples:
`a1.name, a2.name, a1.itemcode, a3.price`

Example

```
#include <stdio.h>

struct complex
{
    int real, img;
};

int main()
{
    struct complex a, b, c;

    printf("Enter a and b where a + ib is the first complex number.\n");
    scanf("%d%d", &a.real, &a.img);
    printf("Enter c and d where c + id is the second complex number.\n");
    scanf("%d%d", &b.real, &b.img);

    c.real = a.real + b.real;
    c.img = a.img + b.img;

    printf("Sum of the complex numbers: (%d) + (%di)\n", c.real, c.img);

    return 0;
}
```

Arrays of Structures

- Once a structure has been defined, we can declare an array of structures

```
struct item a[5];  
          ^  
          type name
```

- The individual members can be accessed as:

a[i].name

a[2].price;

Arrays within Structures

- A structure member can be an array

```
struct student
{
    char name[30];
    int roll_number;
    int marks[5];
    char dob[10];
} a1, a2, a3;
```

- The array element within the structure can be accessed as:

a1.marks[2], a1.dob[3],...

Structure Initialization

- Structure variables may be initialized following similar rules of an array. The values are provided within the second braces separated by commas
- An example:

```
struct complex a={1.0,2.0}, b={-3.0,4.0};
```



```
a.real=1.0; a.imag=2.0;  
b.real=-3.0; b.imag=4.0;
```

Union

- A **union** is a special data type available in C that allows to store different data types in the same memory location.
- You can define a union with many members, but only one member can contain a value at any given time.

Union (Contd...)

- A union is declared using the keyword ***union*** as:
union student

```
{  
    char name[20];  
    int roll_no;  
    float marks;  
    char section;  
};
```

union student s;

- While accessing union members, we should make sure that we are accessing the member whose value is currently residing in the memory. Otherwise we will get erroneous output (which is machine dependent).

Example 1

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};
int main( )
{
    union Data data;
    printf( "Memory size occupied by data : %d\n", sizeof(data));
    return 0;
}
```

Output:
Memory size occupied by data : 20

Example 2

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};

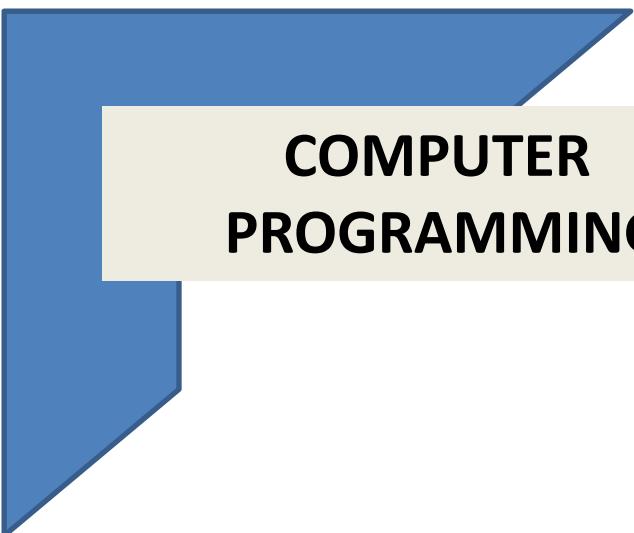
int main( )

{
    union Data data;
    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");
    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);
    return 0;
}
```

Structure vs Union

- Both **structure** and **unions** are used to group a number of different **variables** together. Syntactically both structure and unions are exactly same. The main difference between them is in storage.
- In structures, each member has its own memory location but all members of union use the **same memory location** which is equal to the greatest member's size.

Thank You



COMPUTER PROGRAMMING

File Handling in C

Files

- **File – place on disc where group of related data is stored**
 - E.g. your C programs
- **High-level programming languages support file operations:**
 - *Naming*
 - *Opening*
 - *Reading*
 - *Writing*
 - *Closing*

Defining and Opening file

- *Filename (e.g. sort.c, input.data)*
- *Data structure (e.g. FILE)*
- *Purpose (e.g. reading, writing, appending)*

Filename

- String of characters that make up a valid filename
- May contain two parts
 - *Primary*
 - *Optional period with extension*
- Examples: *a.out, prog.c, temp, text.out*

Different operations that can be performed on a file

- 1. Creation of a new file (fopen with attributes as “a” or “a+” or “w” or “w++”)**
- 2. Opening an existing file (fopen)**
- 3. Reading from file (fscanf or fgetc)**

Different operations that can be performed on a file

4. Writing to a file (`fprintf` or `fputs`)
5. Moving to a specific location in a file (`fseek`, `rewind`)
6. Closing a file (`fclose`)

The text highlighted in the brackets denotes the functions used for performing those operations.

General format for opening file

FILE *filepointer;

/*variable filepointer is pointer to type FILE*/

So, the file can be opened as:

filepointer = fopen(“filename”, “mode”);

/*opens file with name *filename* , assigns identifier to fp */

e.g. filePointer = fopen(“fileName.txt”, “w”)

General format for opening file

- **filepointer**
 - contains all information about file
 - Communication link between system and program
- **Mode can be**
 - **r** open file for reading only
 - **w** open file for writing only
 - **a** open file for appending (adding) data

Modes

- **Writing mode**
 - if file already exists then *contents are deleted*,
 - else new file with specified name created
- **Appending mode**
 - if file already exists then file opened with contents safe
 - else new file created
- **Reading mode**
 - if file already exists then opened with contents safe
 - else error occurs.

Modes

- “r” – Searches file. If the file is opened successfully `fopen()` loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened `fopen()` returns NULL.
- “w” – Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.

Modes

- “a” – Searches file. If the file is opened successfully fopen() loads it into memory and sets up a pointer that points to the last character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.

Example.....

```
FILE *p1, *p2;
```

```
p1 = fopen("data","r");
```

```
p2= fopen("results", w');
```

Additional modes

- **r+** open to beginning for both reading/writing
- **w+** same as w except both for reading and writing
- **a+** same as 'a' except both for reading and writing

Closing a file

- File must be closed as soon as all operations on it completed
- This Ensures
 - All outstanding information associated with file flushed out from buffers
 - All links to file broken
 - Accidental misuse of file prevented
- If want to change mode of file, then first close and open again

Examples.....

Syntax: **fclose(file_pointer);**

Example:

```
FILE *p1, *p2;  
p1 = fopen("INPUT.txt", "r");  
p2 = fopen("OUTPUT.txt", "w");
```

.....

.....

```
fclose(p1);  
fclose(p2);
```

Operations on files

Reading from a file

The file read operations can be performed using functions **fscanf** or **fgets**. Both the functions performed the same operations as that of printf and gets but with an additional parameter, the *file pointer*.

So, it depends on you if you want to read the file line by line or character by character.

```
FILE * filePointer;
```

```
filePointer = fopen("fileName.txt", "r");
```

```
fscanf(filePointer, "%s %s %s %d", str1, str2, str3, &year);
```

Writing a file

The file write operations can be performed by the functions **fprintf** and **fputs** with similarities to read operations.

```
FILE *filePointer ;
```

```
filePointer = fopen("fileName.txt", "w");
```

```
fprintf(filePointer, "%s %s %s %d", "We", "are", "in", 2012);
```

- C provides several different functions for reading/writing
- **getc()** – read a character
- **putc()** – write a character
- **fprintf()** – write set of data values
- **fscanf()** – read set of data values
- **getw()** – read integer
- **putw()** – write integer

getc() and putc()

- handle one character at a time like getchar() and putchar()
- syntax: putc(c,fp1);
 - c : a character variable
 - fp1 : pointer to file opened with mode w
- syntax: c = getc(fp2);
 - c : a character variable
 - fp2 : pointer to file opened with mode r
- file pointer moves by one character position after every getc() and putc()
- getc() returns end-of-file marker EOF when file end reached

fscanf() and fprintf()

- similar to scanf() and printf()
- in addition provide file-pointer
- given the following
 - file-pointer f1 (points to file opened in write mode)
 - file-pointer f2 (points to file opened in read mode)
 - integer variable i
 - float variable f
- Example:

```
fprintf(f1, "%d %f\n", i, f);
fprintf(stdout, "%f \n", f);      /*note: stdout refers to screen */
fscanf(f2, "%d %f", &i, &f);
```
- fscanf returns EOF when end-of-file reached

getw() and putw()

- handle one integer at a time
- syntax: `putw(i,fp1);`
 - `i` : an integer variable
 - `fp1` : pointer to file opened with mode **w**
- syntax: `i = getw(fp2);`
 - `i` : an integer variable
 - `fp2` : pointer to file opened with mode **r**
- file pointer moves by one integer position, data stored in binary format native to local system
- `getw()` returns end-of-file marker EOF when file end reached

Random access to files

- how to jump to a given position (byte number) in a file without reading all the previous data?
- `fseek(file-pointer, offset, position);`
- position: 0 (beginning), 1 (current), 2 (end)
- offset: number of locations to move from position
 - Example: `fseek(fp,-m, 1); /* move back by m bytes from current position */`
 - `fseek(fp,m,0); /* move to (m+1)th byte in file */`
 - `fseek(fp, -10, 2); /* what is this? */`
- `ftell(fp)` returns current byte position in file
- `rewind(fp)` resets position to start of file

EXAMPLE: FILE WRITING (FILES & STRUCTURE)

```
#include<stdio.h>
int main( )
{
    FILE *fp ;
    char another = 'Y' ;
    struct emp
    {
        char name[40] ;
        int age ;
        float bs ;
    } ;
    struct emp e ;

    fp = fopen ("record.txt", "w" ) ;

    while ( another == 'Y' || another == 'y' )
    {
        printf ( "\nEnter name, age and basic salary: " ) ;
        scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;
        fprintf ( fp, "%s %d %f\n", e.name, e.age, e.bs ) ;
        printf ( "Add another record (Y/N) " ) ;
        fflush ( stdin ) ;
        scanf("%c",&another);
    }
    fclose ( fp ) ;
    return 0;
}
```

EXAMPLE:
FILE READING (FILES & STRUCTURE)

```
#include<stdio.h>
int main( )
{
    FILE *fp ;
    struct emp
    {
        char name[40] ;
        int age ;
        float bs ;
    } ;
    struct emp e ;

    fp = fopen ("record.txt", "r" ) ;

    while ( fscanf ( fp, "%s %d %f", e.name, &e.age, &e.bs ) != EOF )
        printf ( "\n%s %d %f", e.name, e.age, e.bs ) ;

    fclose ( fp ) ;
    return 0;
}
```

EXAMPLE:
COPY ONE FILE TO ANOTHER FILE

```
#include<stdio.h>
int main( )
{
    FILE *fp,*fp2 ;
    struct emp
    {
        char name[40] ;
        int age ;
        float bs ;
    } ;
    struct emp e ;

    fp = fopen ("record.txt", "r" ) ;
    fp2 = fopen ("copy.txt", "w" ) ;

    while ( fscanf ( fp, "%s %d %f", e.name, &e.age, &e.bs ) != EOF )
        fprintf ( fp2, "%s %d %f\n", e.name, e.age, e.bs ) ;

    fclose (fp) ;
    fclose (fp2) ;
    return 0;
}
```

Example

Program to read/write using getc/putc

```
#include <stdio.h>
main()
{
    FILE *fp1;
    char c;
    f1= fopen("INPUT", "w"); /* open file for writing */

    while((c=getchar()) != EOF) /*get char from keyboard until CTL-Z*/
        putc(c,f1);           /*write a character to INPUT */

    fclose(f1);               /* close INPUT */
    f1=fopen("INPUT", "r");   /* reopen file */

    while((c=getc(f1))!=EOF) /*read character from file INPUT*/
        printf("%c", c);     /* print character to screen */

    fclose(f1);
} /*end main */
```

Examples

C program using getw, putw,fscanf, fprintf

```
#include <stdio.h>
main()
{ int i,sum1=0;
FILE *f1;
/* open files */
f1 = fopen("int_data.bin","w");
/* write integers to files in binary
and text format*/
for(i=10;i<15;i++)      putw(i,f1);
fclose(f1);
f1 = fopen("int_data.bin","r");
while((i=getw(f1))!=EOF)
{ sum1+=i;
printf("binary file: i=%d\n",i);
} /* end while getw */
printf("binary sum=%d,sum1);
fclose(f1);
}
```

```
#include <stdio.h>
main()
{ int i, sum2=0;
FILE *f2;
/* open files */
f2 = fopen("int_data.txt","w");
/* write integers to files in binary and
text format*/
for(i=10;i<15;i++) printf(f2,"%d\n",i);
fclose(f2);
f2 = fopen("int_data.txt","r");
while(fscanf(f2,"%d",&i)!=EOF)
{ sum2+=i; printf("text file:
i=%d\n",i);
} /*end while fscanf*/
printf("text sum=%d\n",sum2);
fclose(f2);
}
```