IMPERATIVE PROGRAMMING

UNIT 1

Types of computer programming languages

There are basically three types of computer programming languages, they are

- Low level programming languages
- Middle level programming languages
- High level programming languages

LOW LEVEL LANGUAGES

- These are machine dependent programming languages such as Binary (Machine code) and Assembly language.
- Since computer only understand the Binary language that means instructions in the form of 0’s and 1’s (Signals - that can be either High or Low), so these programming languages are the best way to give signals (Binary Instructions) to the computer directly.
- Machine Code (Binary Language) does not need any interpreter or compiler to convert language in any form because computer understands these signals directly.
- But, Assembly language needs to be converted in equivalent Binary code, so that computer can understand the instructions written in Assembly. Assembler is used to convert an assembly code to its equivalent Binary code.
- The codes written in such kind of languages are difficult to write, read, edit and understand; the programs are not portable to any other computer system.

MIDDLE LEVEL PROGRAMMING LANGUAGE

- Since, there is no such category of computer programming languages, but the programming languages that have features of low level and high level programming languages come under this category.
- Hence, we can say that **the programming languages which have features of Low Level as well as High Level programming languages known as "Middle Level" programming language**.
- C programming languages is the best example of **Low Level Programming languages** as it has features of low level and high level programming languages both.

High level programming languages

- These are the machine independent programming languages, which are easy to write, read, edit and understand.
- The languages like Java, .Net, Pascal, COBOL, C++, C, C# and other (which are very popular now to develop user end applications). These languages come under the high level programming language category.
- High level programming languages have some special keywords, functions and class libraries by using them we can easily build a program for the computer.
- Computer does not understand program written in such languages directly, as I have written above that computer understands only Machine code. So, here programming translators are required to convert a high level program to its equivalent Machine code.
- Programming translators such as Compilers and Interpreters are the system software's which converts a program written in particular programming languages to its equivalent Machine code.

FEATURES OF HIGH LEVEL PROGRAMMING LANGUAGES

- The programs are written in High Level programming languages and are independent that means a program written on a system can be run on another system.
- Easy to understand - Since these programming languages have keywords, functions, class libraries (which are similar to English words) we can easily understand the meaning of particular term related to that programming language.
- Easy to code, read and edit - The programs written in High Level programming languages are easy to code, read and edit. Even we can edit programs written by other programmers easily by having little knowledge of that programming language.
- Since, High Level language programs are slower than Low level language programs; still these programming languages are popular to develop User End Applications.

IMPERATIVE PROGRAMMING

- **Imperative programming** is a paradigm of computer programming in which the program describes a sequence of steps that change the state of the computer.
- Unlike declarative programming, which describes "what" a program should accomplish, imperative programming explicitly tells the computer "how" to accomplish it.
- Programs written this way often compile to binary executables that run more efficiently since all CPUinstructions are themselves imperative statements.
- To make programs simpler for a human to read and write, imperative statements can be grouped into sections known as code blocks.

FEATURES OF IP

- Procedural programming is a type of imperative programming in which the program is built from one or more procedures (also termed subroutines.
- To make programs simpler for a human to read and write, imperative statements can be grouped into sections known as code blocks.

HISTORY OF IP

- Earlier computers had fixed programs: they were hardwired to do one thing.
- Sometimes external programs were implemented with paper tape or by setting switches.
- First imperative languages: assembly languages
- 1954-1955: Fortran (FORmula TRANslator)
  John Backus developed for IBM 704
- Late 1950's: Algol (ALGOrithmic Language)
- 1958: Cobol (COmmon Business Oriented Language) Developed by a government committee; Grace Hopper very influential.

- The earliest imperative languages were the machine languages of the original computers.
- In these languages, instructions were very simple, which made hardware implementation easier, but hindered the creation of complex programs.

EVOLUTION OF PROGRAMMING MODEL

- **C programming language** was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.
- **Dennis Ritchie** is known as the **founder of the c language**.
- It was developed to overcome the problems of previous languages such as B, BCPL, etc.
- Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL.
- Let's see the programming languages that were developed before C language.

EVOLUTION OF PROGRAMMING MODEL

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

SIX GENERATIONS OF PROGRAMMING LANGUAGES

Gen.	1st	2nd	3rd	4th	5th	6th
Period	1951-58	1958-64	1964-71	1977-88	1988-	1993-
Type	Low-level	Low-level, High-level	High-level	Very high-level	Object-oriented	Web tools
Example	Machine language, Assembly language	Assembly, COBOL, FORTRAN	BASIC, Pascal	C++, Turbo Pascal, 4GLs	Visual BASIC, OOP, CASE	HTML, Front Page, Java

IMPERATIVE PROGRAMMING

- **Advantage:**
- Very simple to implement
- Better encapsulation
- Bugs free code
- It contains loops, variables etc.

- **Disadvantage:**
- Complex problem cannot be solved
- Less efficient and less productive
- Parallel programming is not possible

ALGORITHM

- In programming, algorithm is a set of well defined instructions in sequence to solve the problem.
- The word "algorithm" relates to the name of the mathematician Al-khowarizmi, which means a procedure or a technique
- Software Engineer commonly uses an algorithm for planning and solving the problems.
- An algorithm is a sequence of steps to solve a particular problem or algorithm is an ordered set of unambiguous steps that produces a result and terminates in a finite time

QUALITIES OF A GOOD ALGORITHM

- Input and output should be defined precisely.
- Each steps in algorithm should be clear and unambiguous.
- Algorithm should be most effective among many different ways to solve a problem.
- An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages

ADVANTAGES OF ALGORITHM

- It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
- An algorithm uses a definite procedure.
- It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
- Every step in an algorithm has its own logical sequence so it is easy to debug.

HOW TO WRITE ALGORITHMS

- Step 1 Define your algorithms input: Many algorithms take in data to be processed, e.g. to calculate the area of rectangle input may be the rectangle height and rectangle width.
- Step 2 Define the variables: Algorithm's variables allow you to use it for more than one place. We can define two variables for rectangle height and rectangle width as HEIGHT and WIDTH (or H & W).
- Step 3 Outline the algorithm's operations: Use input variable for computation purpose, e.g. to find area of rectangle multiply the HEIGHT and WIDTH variable and store the value in new variable (say) AREA. An algorithm's operations can take the form of multiple steps and even branch, depending on the value of the input variables.
- Step 4 Output the results of your algorithm's operations: In case of area of rectangle output will be the value stored in variable AREA. if the input variables described a rectangle with a HEIGHT of 2 and a WIDTH of 3, the algorithm would output the value of 6.

WRITE AN ALGORITHM TO ADD TWO NUMBERS ENTERED BY USER.

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

ALGORITHM FOR AREA OF RECTANGLE

- **START**
- Input Length of Rectangle as **L**
- Input Breadth of Rectangle as **B**
- Calculate **Area= L * B**
- Print Value of **Area**
- **END / STOP**

PSEUDOCODE

  - Artificial, informal language used to develop algorithms
  - Similar to everyday English
- Not executed on computers
  - Used to think out program before coding
    - Easy to convert into C++ program
  - Only executable statements
  - No need to declare variables
  - It's simply an implementation of an algorithm in the form of annotations and informative text written in plain English. It has no syntax like any of the programming language and thus can't be compiled or interpreted by the computer.

ADVANTAGES OF PSEUDOCODE

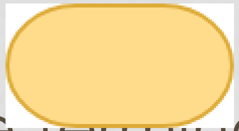
- Improves the readability of any approach. It’s one of the best approaches to start implementation of an algorithm.
- Acts as a bridge between the program and the algorithm or flowchart. Also works as a rough documentation, so the program of one developer can be understood easily when a pseudo code is written out. In industries, the approach of documentation is essential. And that’s where a pseudo-code proves vital.
- The main goal of a pseudo code is to explain what exactly each line of a program should do, hence making the code construction phase easier for the programmer.

FLOWCHART

- A flowchart is a diagrammatic representation that illustrates the sequence of operations to be performed to get the solution to a problem.
- It can be seen from the definition that a flow always accompanies with business or transaction.
- Not all of the flows, however, are appropriate to be expressed by flowcharts, unless these flows are based on some fixed routines and stable links.

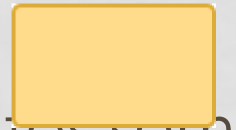
FLOWCHART SYMBOLS

- Terminator



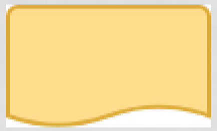
The Terminator symbol represents the starting or ending point of the system.

- Process



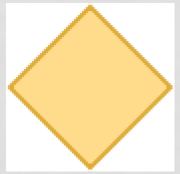
A box indicates some particular operation.

- Document



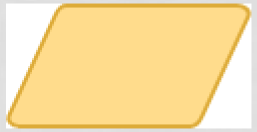
This represents a printout, such as a document or a report.

- Decision



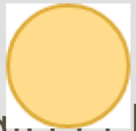
A diamond represents a decision or branching point. Lines coming out from the diamond indicates different possible situations, leading to different sub-processes.

- Data

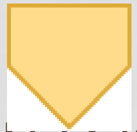


It represents information entering or leaving the system. An input might be an order from a customer. An output can be a product to be delivered.

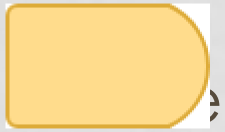
- On-Page Reference



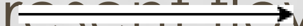
This symbol would contain a letter inside. It indicates that the flow continues on a matching symbol containing the same letter somewhere else on the same page.

- Off-Page Reference 

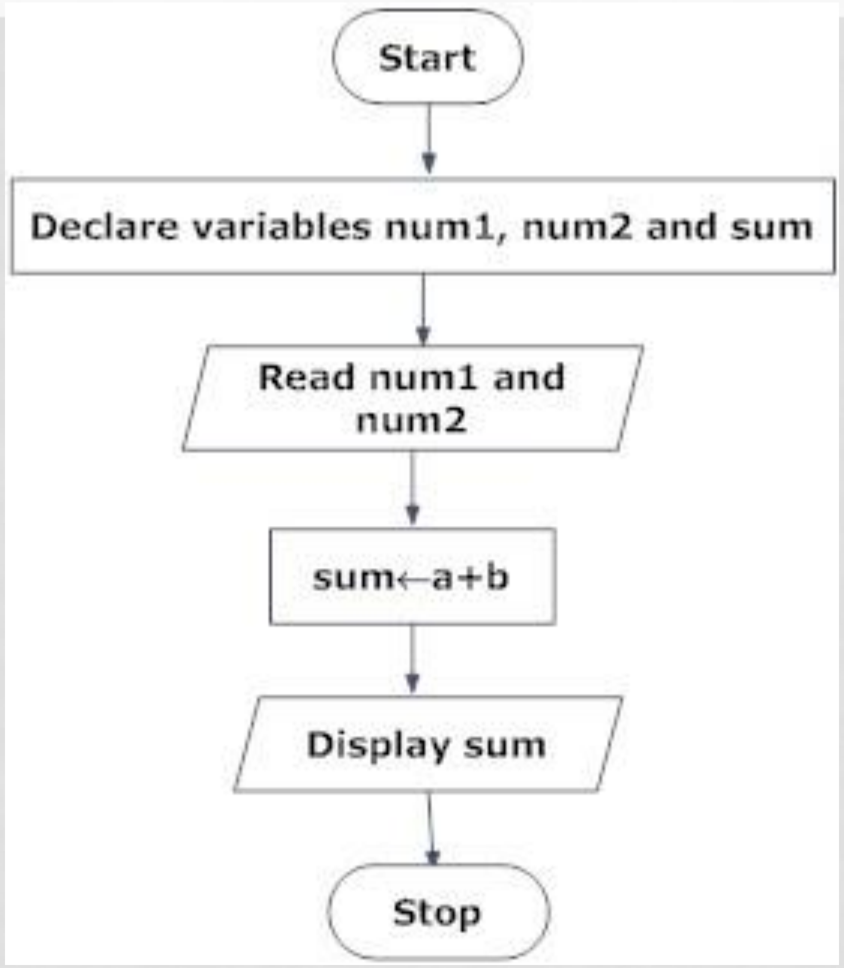
This symbol would contain a letter inside. It indicates that the flow continues on a matching symbol containing the same letter somewhere else on a different page.

- Delay or Bottleneck 

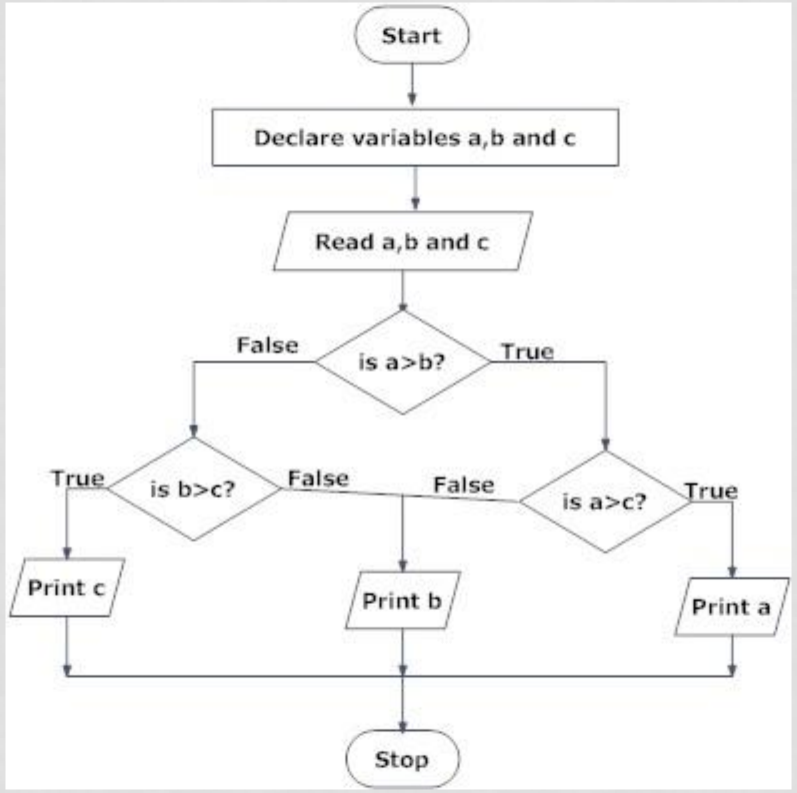
Identifies a delay or a bottleneck.

- Flow
- Lines represent flow of the sequence and direction of a process. 

DRAW A FLOWCHART TO ADD TWO NUMBERS ENTERED BY USER.



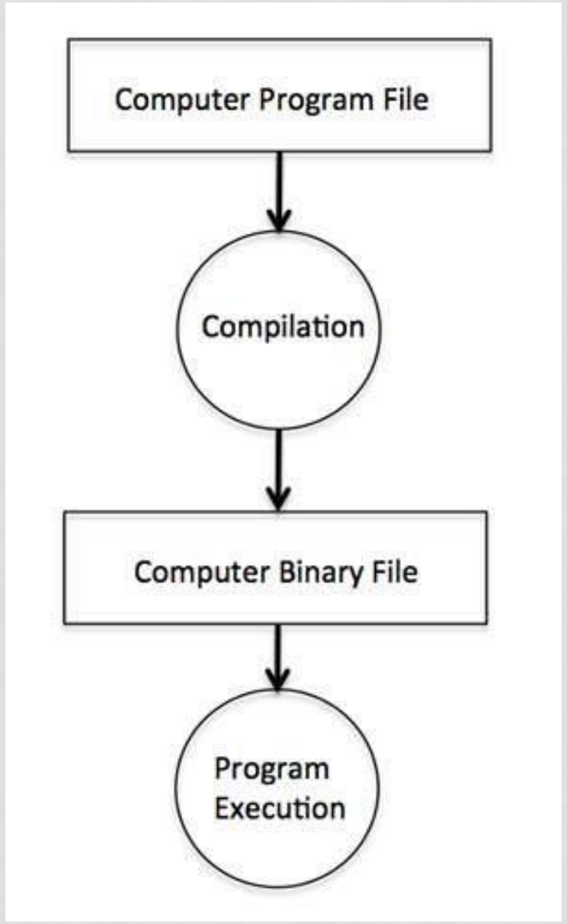
DRAW FLOWCHART TO FIND THE LARGEST AMONG THREE DIFFERENT NUMBERS ENTERED BY USER.



COMPILER

- Actually, the computer cannot understand your program directly given in the text format, so we need to convert this program in a binary format, which can be understood by the computer.
- The conversion from text program to binary file is done by another software called Compiler and this process of conversion from text formatted program to binary format file is called program compilation. Finally, you can execute binary file to perform the programmed task.
- Advantages of Compiler
- A compiler runs much faster.
- The program can be distributed to many people who don't have compilers since a compiler is not needed after compiling occurs.
- A compiler is stored as an executable file.

COMPILER


Computer Program File
Compilation
Computer Binary File
Program Execution

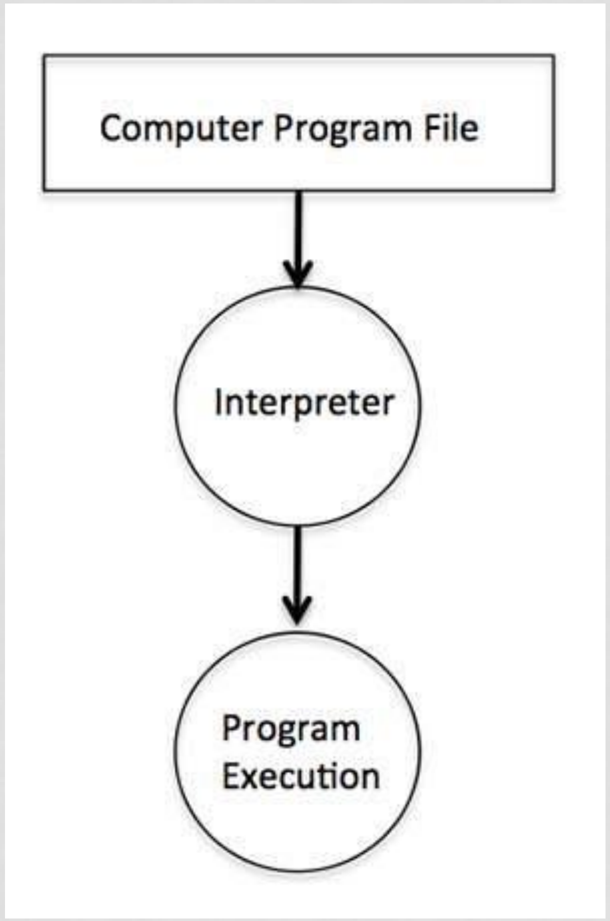
PROGRAMMING SENTINEL VALUE

- In **programming**, **sentinel value** is a special **value** that is used to **terminate** a loop. The **sentinel value** typically is chosen so as to not be a legitimate data **value** that the loop will encounter and attempt to perform with.
- The sentinel value typically is chosen so as to not be a legitimate data value that the loop will encounter and attempt to perform with. For example, in a loop algorithm that computes non-negative integers, the value "-1" can be set as the sentinel value as the computation will never encounter that value as a legitimate processing output.

INTERPRETER

- We just discussed about compilers and the compilation process. Compilers are required in case you are going to write your program in a programming language that needs to be compiled into binary format before its execution.
- There are other programming languages such as Python, PHP, and Perl, which do not need any compilation into binary format, rather an interpreter can be used to read such programs line by line and execute them directly without any further conversion.
- Advantage of an interpreter:
  - It immediately displays feedback when it finds a syntax error.
  - The programmer can correct any errors or debug the code before the interpreter evaluates the next line.
- Disadvantages:
  - Interpreted programs do not run as fast as compiled programs because the program must be translated to machine language each time it is executed

INTERPRETER



PROGRAM FOR "HELLO WORLD"

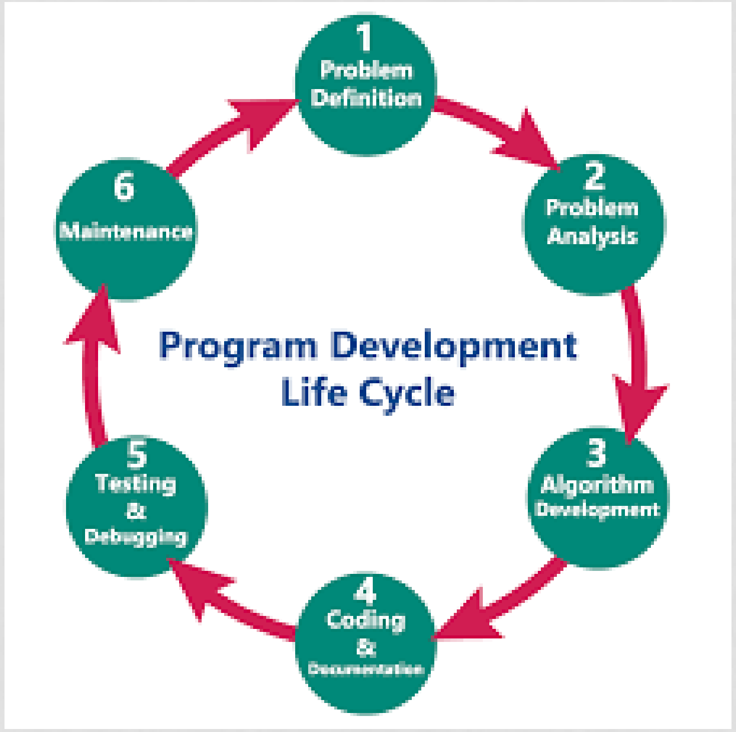
```
• #include <stdio.h>
• int main()
• {
•    // printf() displays the string inside quotation
•    printf("Hello, World!");
•    return 0;
• }
```

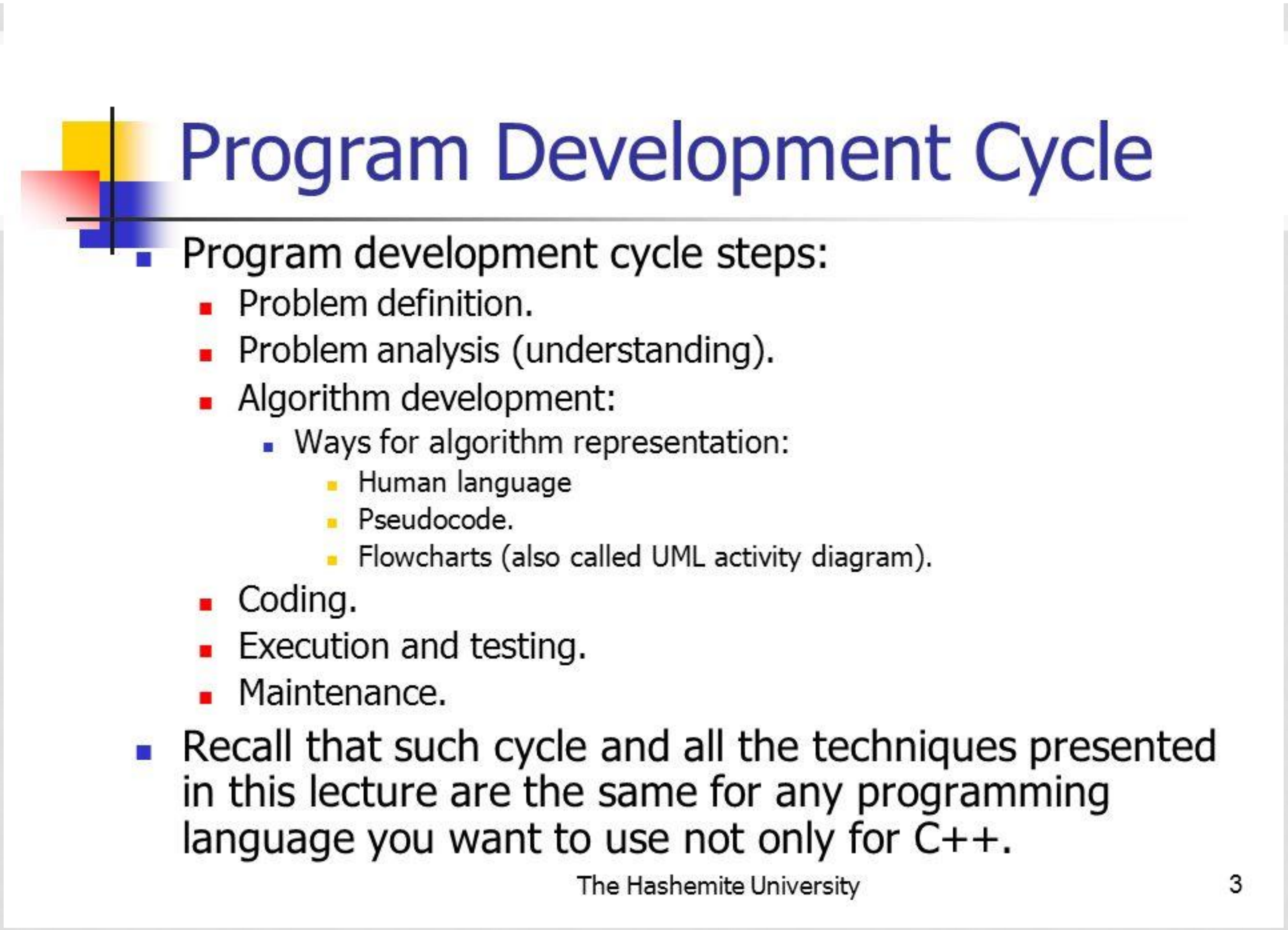
- The #include <stdio.h> is a preprocessor command. This command tells compiler to include the contents of stdio.h (standard input and output) file in the program.
  The stdio.h file contains functions such as scanf() and print() to take input and display output respectively.
  If you use printf() function without writing #include <stdio.h>, the program will not be compiled.
- The execution of a C program starts from the main() function.
- The printf() is a library function to send formatted output to the screen. In this program, the printf() displays Hello, World! text on the screen.
- The return 0; statement is the "Exit status" of the program. In simple terms, program ends with this statement.
-

```
#include<stdio.h>

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

   printf("Enter two numbers to add\n");
   scanf("%d%d", &a, &b);

   c = a + b;

   printf("Sum of the numbers = %d\n", c);

   return 0;
}
```

EVOLUTION OF PROGRAMMING MODELS

PROGRAM DEVELOPMENT LIFE CYCLE





Program Development Cycle

- Program development cycle steps:
  - Problem definition.
  - Problem analysis (understanding).
  - Algorithm development:
    - Ways for algorithm representation:
      - Human language
      - Pseudocode.
      - Flowcharts (also called UML activity diagram).
  - Coding.
  - Execution and testing.
  - Maintenance.
- Recall that such cycle and all the techniques presented in this lecture are the same for any programming language you want to use not only for C++.

- Problem Definition
  - needed output
  - available input
  - definition of how to transform available input into needed output (or processing requirements).
- **Analyze Problem**
  - Review the program specifications package
  - Meet with the systems analyst and users
  - Identify the program’s input, output, and processing requirements (IPO)

- **Design Programs**
- Group the program activities into modules
- Devise a solution algorithm for each module
- Test the solution algorithms
- Top-down design
- Identify the major activity of the program
- Break down the original set of program specifications into smaller, more manageable sections which makes it easier to solve that the original one.
- Continue to break down subroutines into modules which is a section of a program dedicated to performing a single function.
- Hierarchy Chart or Top-Down Charts

➢ Developing an Algorithm

- Programmers begin solving a problem by developing an algorithm.
- An algorithm is a step-by-step description of how to arrive at a solution. You can think of an algorithm as a recipe or a how-to sheet

➢ Program Code

- Writing the actual program is called **coding.**
- This is where the programmer translates the logic of the pseudocode into actual program code.
- Programs should be written on paper first.
- A good program is one that is:
  - Reliable
  - Works under most conditions
  - Catches obvious and common input errors.

➢**Test Programs**

- Types of errors
  - Syntax errors – occurs when the code violates the syntax, or grammar, of the programming language.
    - Misspelling a command
    - Leaving out required punctuation
    - Typing command words out of order
  - Logic errors a flaw in the design that generates inaccurate results.
- All programs must be tested for errors. This is a process known as debugging.
- There are two types of errors that must be eliminated from a program before it can be used in a real-time computing environment. They are:
  - Syntax Errors
  - Logic Errors

- **Debugging**
- There are several methods of debugging a program. They include:
  - Desk checking
  - Manual testing with sample data
  - Testing sample data on the computer
  - Testing by a select group of potential users.
- **Maintain Programs**
- Correct errors
- Add enhancements
  - Fix errors
  - Modify or expand the program

- **Program Maintenance**

- Programming maintenance activities fall into two categories: operations and changing needs.
- **Operations** - Locating and correcting operational errors and standardizing software.
- **Changing Needs** - Programs may need to be modified for a variety of reasons. For example new tax laws, new company policies.

C PROGRAM STRUCTURE

- Documentations (Documentation Section)
- Preprocessor Statements (Link Section)
- Global Declarations (Definition Section)
- The main() function
  - Local Declarations
  - Program Statements & Expressions
- User Defined Functions

COMPILATION AND EXECUTION OF A PROGRAM

- Let's try to understand the flow of above program by the figure given below.
- 1) C program (source code) is sent to preprocessor first. The preprocessor is responsible to convert preprocessor directives into their respective values. The preprocessor generates an expanded source code.
- 2) Expanded source code is sent to compiler which compiles the code and converts it into assembly code.
- 3) The assembly code is sent to assembler which assembles the code and converts it into object code. Now a simple.obj file is generated.
- 4) The object code is sent to linker which links it to the library such as header files. Then it is converted into executable code. A simple.exe file is generated.
- 5) The executable code is sent to loader which loads it into memory and then it is executed. After execution, output is sent to console

CHARACTER SET

Character Set		
1.	Letters	Uppercase A-Z Lowercase a-z
2.	Digits	All digits 0-9
3.	Special Characters	All Symbols: , . : ; ? ' " ! | \ / ~ _$ % # & ^ * - + < > ( ) { } [ ]
4.	White Spaces	Blank space, Horizintal tab, Carriage return, New line, Form feed

C KEYWORDS

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

IDENTIFIERS

- Identifiers are user-defined names of variables, functions and arrays. It comprises of combination of letters and digits. In C Programming, while declaring identifiers, certain rules have to be followed viz.
- It must begin with an alphabet or an underscore and not digits.
- It must contain only alphabets, digits or underscore.
- A keyword cannot be used as an identifier
- Must not contain white space.
- Only first 31 characters are significant.
- Let us again consider an example
- int age1; float height_in_feet;

DATA TYPES

- Primary Data Types
- Derived Data Types
- User Defined Data Types

PRIMARY DATA TYPES

void	As the name suggests, it holds no value and is generally used for specifying the type of function or what it returns. If the function has a void type, it means that the function will not return any value.
int	Used to denote an integer type.
char	Used to denote a character type.
float, double	Used to denote a floating point type.
int *, float *, char *	Used to denote a pointer type.

DERIVED DATA TYPES

Data Types	Description
Arrays	Arrays are sequences of data items having homogeneous values. They have adjacent memory locations to store values.
Reference s	Function pointers allow referencing functions with a particular signature.
Pointers	These are powerful C features which are used to access the memory and deal with their addresses.

USER DEFINED DATA TYPES

Data Types	Description
Structure	It is a package of variables of different types under a single name. This is done to handle data efficiently. "struct" keyword is used to define a structure.
Union	These allow storing various data types in the same memory location. Programmers can define a union with different members, but only a single member can contain a value at a given time. It is used for
Enum	Enumeration is a special data type that consists of integral constants, and each of them is assigned with a specific name. "enum" keyword is used to define the enumerated data type.

CONSTANTS

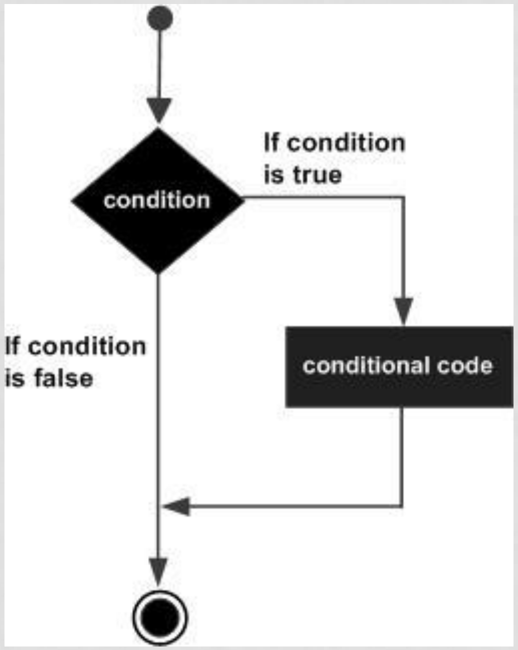
- C Constants is the most fundamental and essential part of the C programming language. Constants in C are the fixed values that are used in a program, and its value remains the same during the entire execution of the program.
- Constants are also called literals.
- Constants can be any of the data types.
- It is considered best practice to define constants using only *upper-case* names.

IMPERATIVE PROGRAMMING

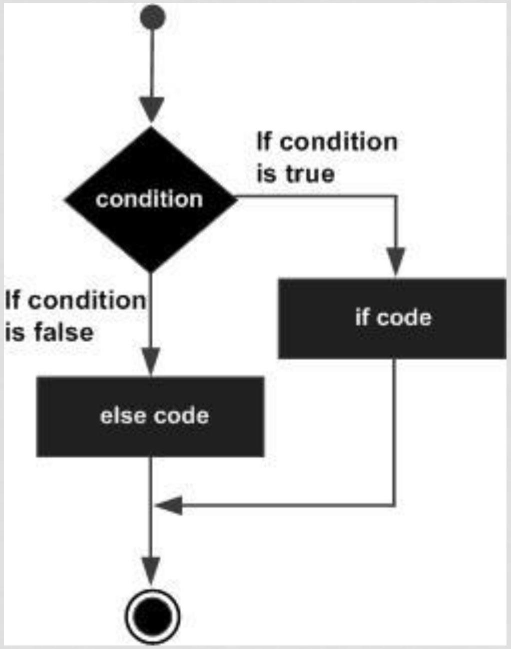
UNIT 2

Compiled by Ms. Prajakta Joshi

IF CONDITIONAL STATEMENT



IF….ELSE STATEMENT



ARITHMETIC OPERATORS

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

```
#include <stdio.h>
 main()
{
 int a = 21; int b = 10; int c ;
c = a + b;
printf("Line 1 - Value of c is %d\n", c );
c = a - b;
printf("Line 2 - Value of c is %d\n", c );
 c = a * b;
printf("Line 3 - Value of c is %d\n", c );
c = a / b;
 printf("Line 4 - Value of c is %d\n", c );
c = a % b;
 printf("Line 5 - Value of c is %d\n", c );
}
```

UNARY OPERATORS IN C

- **Unary operator:** are operators that act upon a single operand to produce a new value.
- **Types of unary operators:**
- unary minus(-)
- increment(++)
- decrement(- -)
- NOT(!)
- Addressof operstor(&)
- sizeof()

```
#include <stdio.h>
int main()
{
int a = 10, b = 100;
float c = 10.5, d = 100.5;
printf("++a = %d \n", ++a);
printf("--b = %d \n", --b);
printf("++c = %f \n", ++c);
printf("--d = %f \n", --d);
return 0;
}
```

SIZEOF OPERATOR

```
#include <stdio.h>
int main()
{
int a;
float b;
double c;
char d;
printf("Size of int=%lu bytes\n",sizeof(a));
printf("Size of float=%lu bytes\n",sizeof(b));
printf("Size of double=%lu bytes\n",sizeof(c));
printf("Size of char=%lu byte\n",sizeof(d));
return 0;
}
```

RELATIONAL OPERATORS

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

```
• #include <stdio.h>
• main()
• {
•  int a = 21; int b = 10; int c ;
•  if( a == b )
•  {
•  printf("Line 1 - a is equal to b\n" );
• }
•  else
•  {
•  printf("Line 1 - a is not equal to b\n" );
• }
• }
```

LOGICAL OPERATORS

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is	!(A && B) is true.

- #include <stdio.h>
- main() {
- int a = 5; int b = 20; int c ;
- if (a && b)
- {
- printf("Line 1 - Condition is true\n");
- }
- if (a | | b)
- {
- printf("Line 2 - Condition is true\n"); }

- /* lets change the value of a and b */
- a = 0; b = 10;
- if (a && b)
- {
- printf("Line 3 - Condition is true\n");
- }
- else
- {
- printf("Line 3 - Condition is not true\n");
- }
- if (!(a && b))
- {
- printf("Line 4 - Condition is true\n");
- }
- }

ASSIGNMENT OPERATORS

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

```
• #include <stdio.h>
• main()
•  {
•  int a = 21; int c ;
• c = a;
• printf("Line 1 - = Operator Example, Value of c =
  %d\n", c );
• c += a;
•  printf("Line 2 - += Operator Example, Value of c =
  %d\n", c );
•  c -= a;
•  printf("Line 3 - -= Operator Example, Value of c =
  %d\n", c );
• c *= a;
• printf("Line 4 - *= Operator Example, Value of c =
```

- c = 200; c %= a;
- printf("Line 6 - %= Operator Example, Value of c = %d\n", c);
- c <<= 2;
- printf("Line 7 - <<= Operator Example, Value of c = %d\n", c);
- c >>= 2;
- printf("Line 8 - >>= Operator Example, Value of c = %d\n", c);
- c &= 2;
- printf("Line 9 - &= Operator Example, Value of c = %d\n", c);
- c ^= 2;
- printf("Line 10 - ^= Operator Example, Value of c = %d\n", c);
- c |= 2;
- printf("Line 11 - |= Operator Example, Value of c = %d\n", c);
- }

CONDITIONAL OR TERNARY OPERATORS IN C

- Conditional operators return one value if condition is true and returns another value is condition is false.
- This operator is also called as ternary operator.
- Syntax : (Condition? true_value: false_value);
- Example : (A > 100 ? 0 : 1);

```
#include <stdio.h>

int main()
{
   int x=1, y ;
   y = ( x ==1 ? 2 : 0 ) ;
   printf("x value is %d\n", x);
   printf("y value is %d", y);
}
```

```
• #include<stdio.h>

• int main()
• {
•   int age;

•   printf(" Please Enter your age here: \n ");
•   scanf(" %d ", &age);

•  (age >= 18) ? printf(" You are eligible to Vote ") :
•               printf(" You are not eligible to Vote ");

•  return 0;
• }
```

TYPE CONVERSION/TYPE CASTING

- Implicit Type Conversion/ Casting
- Explicit Type Conversion/ Casting

IMPLICIT TYPE CONVERSION/ CASTING

- Implicit type casting means conversion of data types without losing its original meaning.
- This type of typecasting is essential when you want to change data types **without** changing the significance of the values stored inside the variable.
- Implicit type conversion happens automatically when a value is copied to its compatible data type.
- During conversion, strict rules for type conversion are applied.
- If the operands are of two different data types, then an operand having lower data type is automatically converted into a higher data type.

EXAMPLE OF ITC

```
#include<stdio.h>
int main()
{
short a=10; //initializing variable of short data type
        int b; //declaring int variable
        b=a; //implicit type casting
        printf("%d\n",a);
        printf("%d\n",b);
}
```

EXAMPLE OF ITC FROM CHAR TO INT

```
#include<stdio.h>
int main()
{
   int x = 10;    // integer x
   char y = 'a';  // character c

   // y implicitly converted to int. ASCII
   // value of 'a' is 97
   x = x + y;

   // x is implicitly converted to float
   float z = x + 1.0;

   printf("x = %d, z = %f", x, z);
   return 0;
}
```

EXPLICIT TYPE CONVERSION

- When interpretation is between a variable having a data type with respect to size & type both, this conversion is not possible for compiler automatically.
- It is performed by the programmer.
- In this type casting programmer tells compiler to type cast one data type to another data type using type casting operator.
- but there is some risk of information loss is there, so one needs to be careful while doing it.

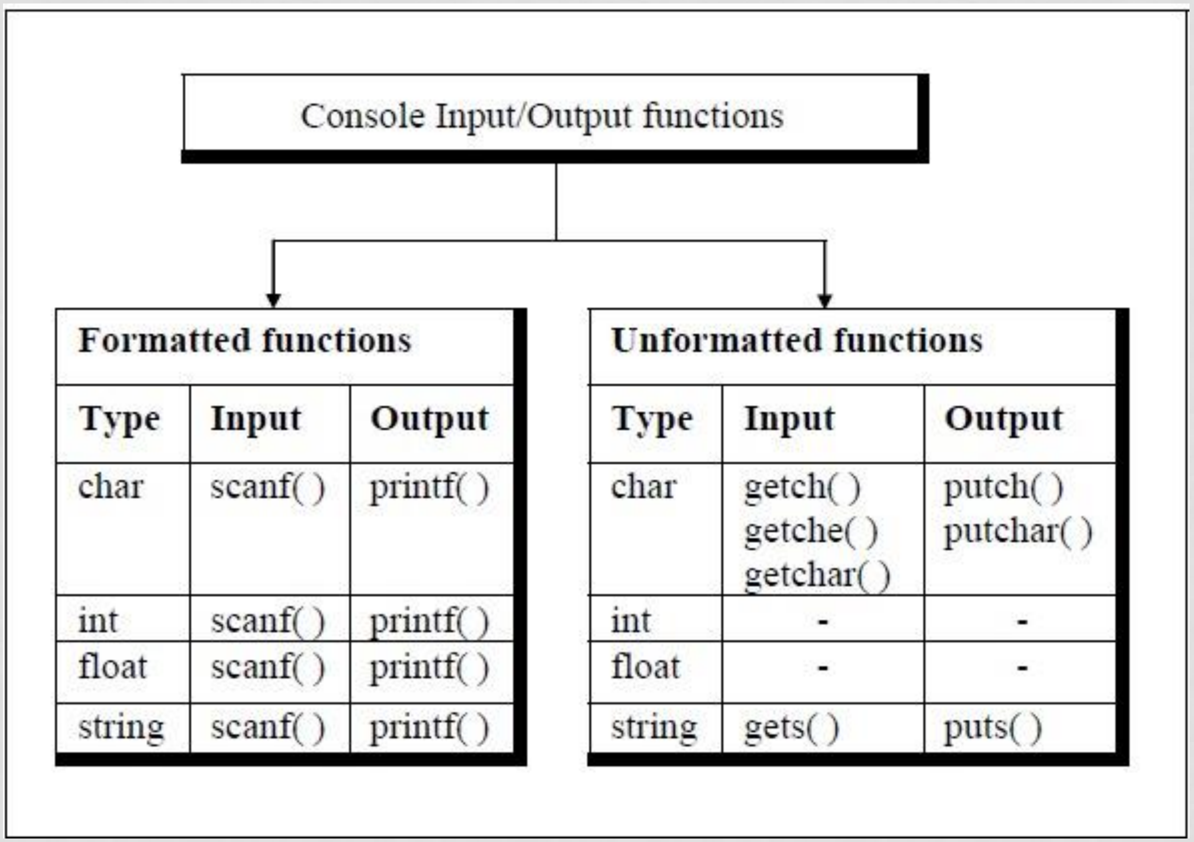
C PROGRAM TO DEMONSTRATE EXPLICIT TYPE CASTING

```
#include<stdio.h>

int main()
{
    double x = 1.2;

    // Explicit conversion from double to int
    int sum = (int)x + 1;

    printf("sum = %d", sum);

    return 0;
}
```

C PROGRAM TO DEMONSTRATE EXPLICIT TYPE CASTING

```
• #include<stdio.h>
• int main()
• {
•       float a = 1.2;
•       //int b  = a; //Compiler will throw an error for this
•       int b = (int)a + 1;
•       printf("Value of a is %f\n", a);
•       printf("Value of b is %d\n",b);
•       return 0;
• }
```

INPUT & OUTPUT FUNCTIONS


Console Input/Output functions

Formatted functions

Type	**Input**	**Output**
char	scanf( )	printf( )
int	scanf( )	printf( )
float	scanf( )	printf( )
string	scanf( )	printf( )

Unformatted functions

Type	**Input**	**Output**

char	getch( )
	getche( )
	getchar( )
		putch( )
		putchar( )

int	-	-
float	-	-
string	gets( )	puts( )

GETCH() FUNCTION

- The getch() function reads the alphanumeric character input from the user. But, that the entered character will not be displayed.

```
#include <stdio.h>
#include <conio.h>
int main() {
 printf("\nHello, press any alphanumeric character to exit
");
getch();
 return 0; }
```

GETCHE() FUNCTION

- getche() function reads the alphanumeric character from the user input. Here, character you entered will be echoed to the user until he/she presses any key.

```
#include <stdio.h> //header file section
#include <conio.h>
int main() {
 printf("\nHello, press any alphanumeric character or
symbol to exit \n ");
 getche();
return 0; }
```

GETCHAR() FUNCTION

- The getchar() function reads character type data form the input.
- The getchar() function reads one character at a time till the user presses the enter key.

```
#include <stdio.h> //header file section #include
<conio.h>
 int main()
{
char c;
 printf("Enter a character : ");
c = getchar();
printf("\nEntered character : %c ", c);
return 0;
 }
```

GETS() FUNCTION

- The gets() function can read a full string even blank spaces presents in a string.
- But, the scanf() function leave a string after blank space space is detected.
- The gets() function is used to get any string from the user.

GETS() FUNCTION

```
#include <stdio.h>
#include <conio.h>
int main()
 {
char c[25];
 printf("Enter a string : ");
gets(c);
printf("\n%s is awesome ",c);
return 0;
}
```

PUTCH() FUNCTION

- The putch() function prints any alphanumeric character.

```
#include <stdio.h> //header file section
 #include <conio.h>
int main() {
char c;
printf("Press any key to continue\n ");
 c = getch();
printf("input : ");
putch(c);
 return 0; }
```

PUTCHAR() FUNCTION

- putchar() function prints only one character at a time.

```
#include <stdio.h> //header file section #include
<conio.h>
int main() {
 char c = 'K';
putchar(c);
return 0;
 }
```

PUTS() FUNCTION

- The puts() function prints the charater array or string on the console. The puts() function is similar to printf() function, but we cannot print other than characters using puts() function.

PUTS() FUNCTION

```
#include <stdio.h>
#include <conio.h>
 int main()
{
char c[25];
 printf("Enter your Name : ");
gets(c);
puts(c);
return 0;
}
```