What are the Phases of Compiler Design?

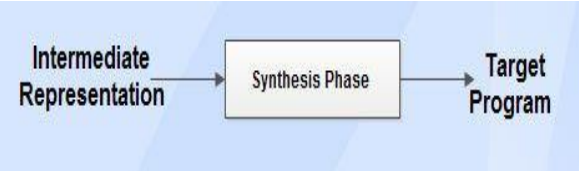
The structure of compiler consists of two parts:

Analysis part

• Analysis part breaks the source program into constituent pieces and imposes a grammatical structure on them which further uses this structure to create an intermediate representation of the source program.

• It is also termed as front end of compiler.

• Information about the source program is collected and stored in a data structure called symbol table.



Synthesis part

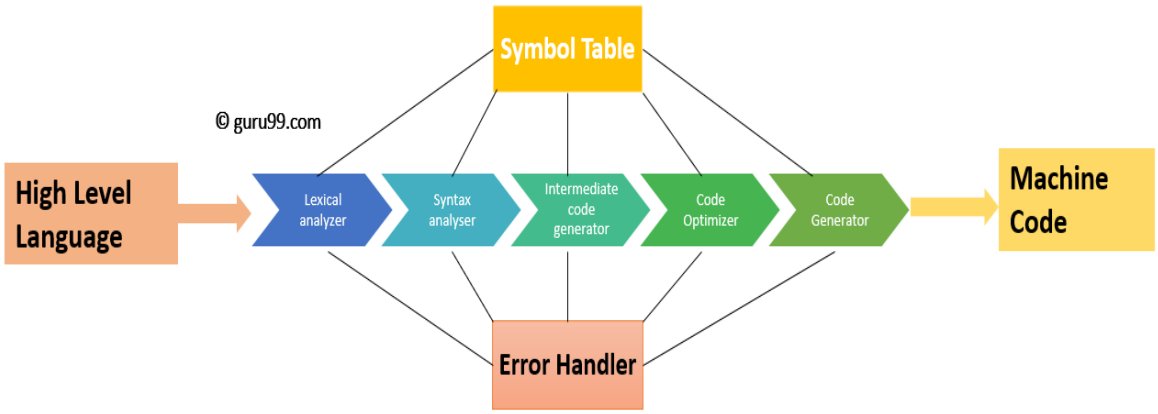
• Synthesis part takes the intermediate representation as input and transforms it to the target program.

• It is also termed as back end of compiler.

The design of compiler can be decomposed into several phases, each of which converts one form of source program into another.

Compiler operates in various phases each phase transforms the source program from one representation to another. Every phase takes inputs from its previous stage and feeds its output to the next phase of the compiler.

There are 6 phases in a compiler. Each of this phase help in converting the high-level langue the machine code. The phases of a compiler are:



Phases of Compiler

All these phases convert the source code by dividing into tokens, creating parse trees, and optimizing the source code by different phases.

Phase 1: Lexical Analysis

Lexical Analysis is the first phase when compiler scans the source code. This process can be left to right, character by character, and group these characters into tokens.

Here, the character stream from the source program is grouped in meaningful sequences by identifying the tokens. It makes the entry of the corresponding tickets into the symbol table and passes that token to next phase.

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A

pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

The primary functions of this phase are:

- Identify the lexical units in a source code
- Classify lexical units into classes like constants, reserved words, and enter them in different tables. It will Ignore comments in the source program
- Identify token which is not a part of the language

Example:

x = y + 10

Tokens

X	identifier
=	Assignment operator
Y	identifier
+	Addition operator
10	Number

Phase 2: Syntax Analysis

Syntax analysis is all about discovering structure in code. It determines whether or not a text follows the expected format. The main aim of this phase is to make sure that the source code was written by the programmer is correct or not.

Syntax analysis is based on the rules based on the specific programing language by constructing the parse tree with the help of tokens. It also determines the structure of source language and grammar or syntax of the language.

Here, is a list of tasks performed in this phase:

- Obtain tokens from the lexical analyzer
- Checks if the expression is syntactically correct or not
- Report all syntax errors
- Construct a hierarchical structure which is known as a parse tree

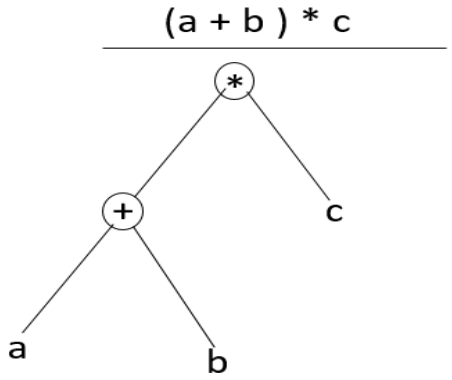
Example

Any identifier/number is an expression

If x is an identifier and y+10 is an expression, then x= y+10 is a statement.

Consider parse tree for the following example

```
(a+b)*c
```



In Parse Tree

- Interior node: record with an operator filed and two files for children
- Leaf: records with 2/more fields; one for token and other information about the token
- Ensure that the components of the program fit together meaningfully
- Gathers type information and checks for type compatibility
- Checks operands are permitted by the source language

Phase 3: Semantic Analysis

Semantic analysis checks the semantic consistency of the code. It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent. It also checks whether the code is conveying an appropriate meaning.

Semantic Analyzer will check for Type mismatches, incompatible operands, a function called with improper arguments, an undeclared variable, etc.

Functions of Semantic analyses phase are:

- Helps you to store type information gathered and save it in symbol table or syntax tree
- Allows you to perform type checking
- In the case of type mismatch, where there are no exact type correction rules which satisfy the desired operation a semantic error is shown
- Collects type information and checks for type compatibility
- Checks if the source language permits the operands or not

Example

```
float x = 20.2;
float y = x*30;
```

In the above code, the semantic analyzer will typecast the integer 30 to float 30.0 before multiplication

Phase 4: Intermediate Code Generation

Once the semantic analysis phase is over the compiler, generates intermediate code for the target machine. It represents a program for some abstract machine.

Intermediate code is between the high-level and machine level language. This intermediate code needs to be generated in such a manner that makes it easy to translate it into the target machine code.

Functions on Intermediate Code generation:

- It should be generated from the semantic representation of the source program
- Holds the values computed during the process of translation
- Helps you to translate the intermediate code into target language
- Allows you to maintain precedence ordering of the source language
- It holds the correct number of operands of the instruction

Example

For example,

```
total = count + rate * 5
```

Intermediate code with the help of address code method is:

```
t1 := int_to_float(5)
t2 := rate * t1
t3 := count + t2
total := t3
```

Phase 5: Code Optimization

The next phase of is code optimization or Intermediate code. This phase removes unnecessary code line and arranges the sequence of statements to speed up the execution of the program without wasting resources. The main goal of this phase is to improve on the intermediate code to generate a code that runs faster and occupies less space.

The primary functions of this phase are:

- It helps you to establish a trade-off between execution and compilation speed
- Improves the running time of the target program
- Generates streamlined code still in intermediate representation
- Removing unreachable code and getting rid of unused variables
- Removing statements which are not altered from the loop

Example:

Consider the following code

```
a = intofloat(10)
b = c * a
d = e + b
f = d
```

Can become

```
b =c * 10.0
f = e+b
```

Phase 6: Code Generation

Code generation is the last and final phase of a compiler. It gets inputs from code optimization phases and produces the page code or object code as a result. The objective of this phase is to allocate storage and generate relocatable machine code.

It also allocates memory locations for the variable. The instructions in the intermediate code are converted into machine instructions. This phase coverts the optimize or intermediate code into the target language.

The target language is the machine code. Therefore, all the memory locations and registers are also selected and allotted during this phase. The code generated by this phase is executed to take inputs and generate expected outputs.

Example:

a = b + 60.0

Would be possibly translated to registers.

```
MOVF a, R1
MULF #60.0, R2
ADDF R1, R2
```

Symbol Table Management

A symbol table contains a record for each identifier with fields for the attributes of the identifier. This component makes it easier for the compiler to search the identifier record and retrieve it quickly. The symbol table also helps you for the scope management. The symbol table and error handler interact with all the phases and symbol table update correspondingly.

Error Handling Routine:

In the compiler design process error may occur in all the below-given phases:

- Lexical analyzer: Wrongly spelled tokens
- Syntax analyzer: Missing parenthesis
- Intermediate code generator: Mismatched operands for an operator
- Code Optimizer: When the statement is not reachable
- Code Generator: Unreachable statements

- Symbol tables: Error of multiple declared identifiers

Most common errors are invalid character sequence in scanning, invalid token sequences in type, scope error, and parsing in semantic analysis.

The error may be encountered in any of the above phases. After finding errors, the phase needs to deal with the errors to continue with the compilation process. These errors need to be reported to the error handler which handles the error to perform the compilation process. Generally, the errors are reported in the form of message.

2. 31) what is a flowchat?

A flowchart is a diagram that depicts a process, system or computer algorithm. They are widely used in multiple fields to document, study, plan, improve and communicate often complex processes in clear, easy-to-understand diagrams. Flowcharts, sometimes spelled as flow charts, use rectangles, ovals, diamonds and potentially numerous other shapes to define the type of step, along with connecting arrows to define flow and sequence. They can range from simple, hand-drawn charts to comprehensive computer-drawn diagrams depicting multiple steps and routes. If we consider all the various forms of flowcharts, they are one of the most common diagrams on the planet, used by both technical and non-technical people in numerous fields.

2. 32) 10.51) what is pseudo code? Discuss with an example.

Pseudo code is a term which is often used in programming and algorithm based fields. It is a methodology that allows the programmer to represent the implementation of an algorithm. Simply, we can say that it's the cooked up representation of an algorithm. Often at times, algorithms are represented with the help of pseudo codes as they can be interpreted by programmers no matter what their programming background or knowledge is. Pseudo code, as the name suggests, is a false code or a representation of code which can be understood by even a layman with some school level programming knowledge.
Algorithm: It's an organized logical sequence of the actions or the approach towards a particular problem. A programmer implements an algorithm to solve a problem. Algorithms are expressed using natural verbal but somewhat technical annotations.
Pseudo code: 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.
here's the **Pseudo Code** for the same.

```
This program calculates the Lowest Common multiple
for excessively long input values

function lcmNaive(Argument one, Argument two){

   Calculate the lowest common variable of Argument
   1 and Argument 2 by dividing their product by their
   Greatest common divisor product

   return lowest common multiple
end
}
```

```
function greatestCommonDivisor(Argument one, Argument two){
   if Argument two is equal to zero
      then return Argument one

   return the greatest common divisor

end
}

{
In the main function

  print prompt "Input two numbers"

  Take the first number from the user
  Take the second number from the user

  Send the first number and second number
  to the lcmNaive function and print
  the result to the user
}
```

2.33) what is the output of an assembler

The output of the assembler program is called the object code or object program relative to the input source program. The sequence of 0's and 1's that constitute the object program is sometimes called machine code. The object program can then be run (or executed) whenever desired. In the earliest computers, programmers actually wrote programs in machine code, but assembler languages or instruction sets were soon developed to speed up programming.

2.34) What is a compiler?

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an *editor*. The file that is created contains what are called the *source statements*. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.

When executing (running), the compiler first parses (or analyzes) all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Traditionally, the output of the compilation has been called *object code* or sometimes an *object module* . (Note that the term "object" here is not related to object-oriented programming.) The object code is machine code that the processor can execute one instruction at a time.

4.11) Describe lexeme, pattern and token with examples

What is a token?
A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

```
Keywords; Examples-for, while, if etc.

Identifier; Examples-Variable name, function name etc.

Operators; Examples '+', '++', '-' etc.

Separators; Examples ',' ';' etc
```

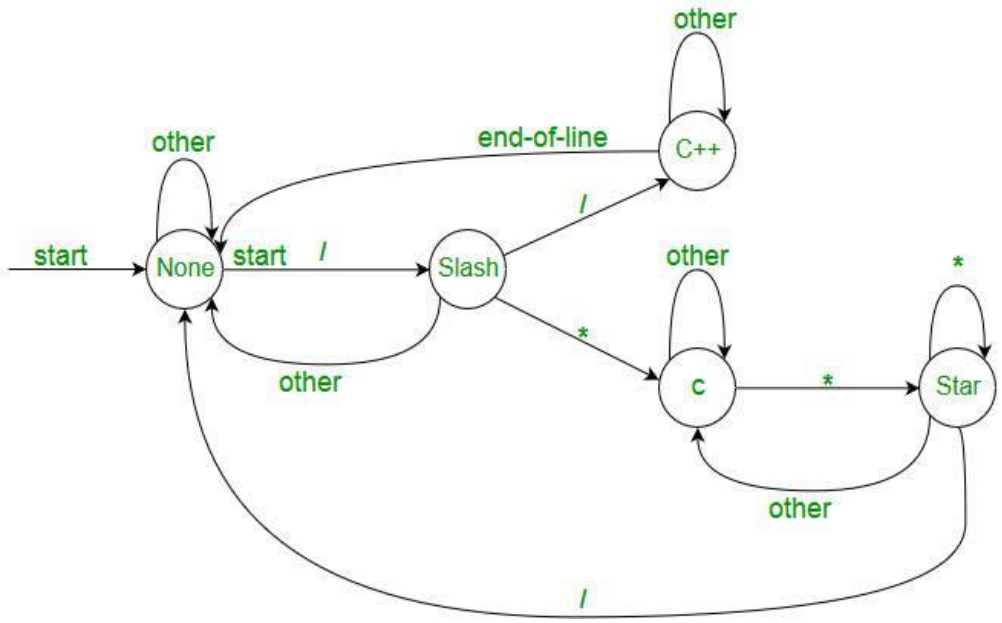
Example of Non-Tokens:

- Comments, preprocessor directive, macros, blanks, tabs, newline etc

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

How Lexical Analyzer functions

1. Tokenization .i.e Dividing the program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error message by providing row number and column number.



- The lexical analyzer identifies the error with the help of automation machine and the grammar of the given language on which it is based like C , C++ and gives row number and column number of the error.

Suppose we pass a statement through lexical analyzer –

a = b + c ; It will generate token sequence like this:

id=id+id; Where each id reference to it's variable in the symbol table referencing all details
For example, consider the program

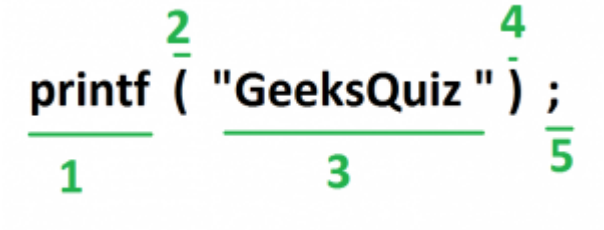
```
int main()
{
  // 2 variables
  int a, b;
  a = 10;
 return 0;
}
```

All the valid tokens are:

```
'int'  'main'  '('  ')'  '{'  'int'  'a' ','  'b'  ';'
 'a'  '='  '10'  ';' 'return'  '0'  ';'  '}'
```

Above are the valid tokens.
You can observe that we have omitted comments.

As another example, consider below printf statement.



There are 5 valid token in this printf statement.

Exercise 1:
Count number of tokens :

```
int main()
{
  int a = 10, b = 20;
  printf("sum is :%d",a+b);
  return 0;
}
Answer: Total number of token: 27.
```

Exercise 2:
Count number of tokens :

int max(int i);

- Lexical analyzer first read **int** and finds it to be valid and accepts as token

- **max** is read by it and found to be valid function name after reading **(**
- **int** is also a token , then again **i** as another token and finally **;**

```
 Answer:  Total number of tokens 7:

int, max, ( ,int, i, ), ;
```

Lexeme pg. 111

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Token pg. 111

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.

Pattern pg. 111

A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is more complex structure that is matched by many strings.

a) Tokens are symbolic names for the entities that make up the text of the program; e.g. if for the keyword if, and id for any identifier. These make up the output of the lexical analyser. 5

(b) A pattern is a rule that specifies when a sequence of characters from the input constitutes a token; e.g the sequence i, f for the token if , and any sequence of alphanumerics starting with a letter for the token id.

(c) A lexeme is a sequence of characters from the input that match a pattern (and hence constitute an instance of a token); for example if matches the pattern for if , and foo123bar matches the pattern for id.

4.12) explain the working principle of lexical analyser

A lexical analyzer (also known as lexer), a pattern recognition engine takes a string of individual letters as its input and divides it into tokens .The lexical analyzer breaks these sentences into a series of tokens, by removing any whitespace or comments in the given code. Some terms related to lexical phase include: 1) Lex: Lex is a program generator designed for lexical processing of character input streams. 2) Tokens: A token or lexical token is a structure representing a lexeme that explicitly indicates its categorization for the purpose of parsing that describes the class or category of input string. 3) Lexeme: Sequence of characters in the program that are matching with the pattern of token.

Lexical analysis is the very first phase in the compiler designing. It takes the modified source code which is written in the form of sentences. In other words, it helps you to converts a sequence of characters into a sequence of tokens. The lexical analysis breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.

Programs that perform lexical analysis are called lexical analyzers or lexers. A lexer contains tokenizer or scanner. If the lexical analyzer detects that the token is invalid, it generates an error. It reads character streams from the source code, checks for legal tokens, and pass the data to the syntax analyzer when it demands.

Roles of the Lexical analyzer

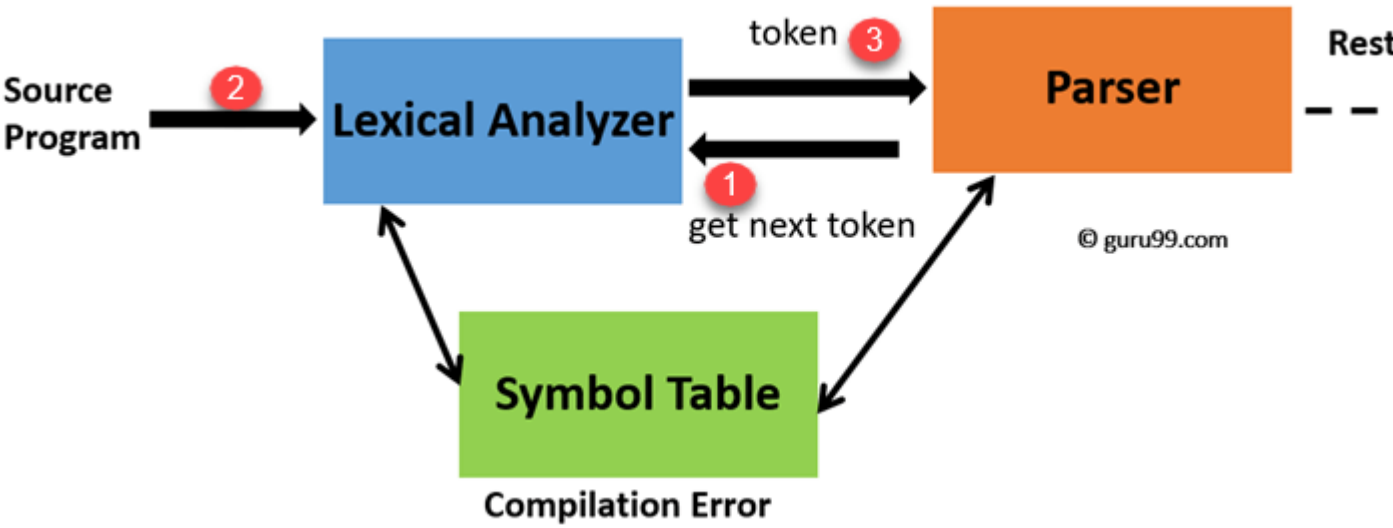
Lexical analyzer performs below given tasks:

- Helps to identify token into the symbol table
- Removes white spaces and comments from the source program
- Correlates error messages with the source program
- Helps you to expands the macros if it is found in the source program
- Read input characters from the source program

Lexical Analyzer Architecture: How tokens are recognized

The main task of lexical analysis is to read input characters in the code and produce tokens.

Lexical analyzer scans the entire source code of the program. It identifies each token one by one. Scanners are usually implemented to produce tokens only when requested by a parser. Here is how this works-



1. "Get next token" is a command which is sent from the parser to the lexical analyzer.
2. On receiving this command, the lexical analyzer scans the input until it finds the next token.

3. It returns the token to Parser.

Lexical Analyzer skips whitespaces and comments while creating these tokens. If any error is present, then Lexical analyzer will correlate that error with the source file and line number.

4.13) Describe the three major families of language.

Definition - What does Assembly Language mean?

An assembly language is a low-level programming language for microprocessors and other programmable devices. It is not just a single language, but rather a group of languages. An assembly language implements a symbolic representation of the machine code needed to program a given CPU architecture.

Assembly language is also known as assembly code. The term is often also used synonymously with 2GL.

Techopedia explains Assembly Language

An assembly language is the most basic programming language available for any processor. With assembly language, a programmer works only with operations that are implemented directly on the physical CPU.

Assembly languages generally lack high-level conveniences such as variables and functions, and they are not portable between various families of processors. They have the same structures and set of commands as machine language, but allow a programmer to use names instead of numbers. This language is still useful for programmers when speed is necessary or when they need to carry out an operation that is not possible in high-level languages.

Definition - What does High-Level Language (HLL) mean?

A high-level language is any programming language that enables development of a program in a much more user-friendly programming context and is generally independent of the computer's hardware architecture.

A high-level language has a higher level of abstraction from the computer, and focuses more on the programming logic rather than the underlying hardware components such as memory addressing and register utilization.

Techopedia explains High-Level Language (HLL)

High-level languages are designed to be used by the human operator or the programmer. They are referred to as "closer to humans." In other words, their programming style and context is easier to learn and implement than low-level languages, and the entire code generally focuses on the specific program to be created.

A high-level language does not require addressing hardware constraints when developing a program. However, every single program written in a high-level language must be interpreted into machine language before being executed by the computer.

BASIC, C/C++ and Java are popular examples of high-level languages.

Machine language is the basic low-level programming language designed to be recognized by a computer. Actually the language is written in a binary code of 0s and 1s that represent electric impulses or off and on electrical states respectively. A group of such digits is called an instruction and it is translated into a command that the central processing unit or CPU understands.

More specifically, instructions are organized in patterns of 0s and 1s in various lengths such as 16, 24, 32, and 64 digits or bits, representing specific tasks such as storing or transferring data.

4. 20) Explain the difference between Machine level and assembly level language.

Basis of Difference	Machine Language	Assembly Language
Level of programming language	Machine language ranks as the lowest level programming language. In this language, instructions are executed directly via the Central Processing Unit.	Assembly language refers to a low-level programming language that needs an assembler for converting the instructions to machine or object codes.
Ease of comprehension	Machine language cannot be deciphered by humans and can be comprehended only by computers.	Assembly language can be understood, used and applied by humans.
Nature of syntax	Machine languages comprise of binary digits 0s and 1s.	Assembly languages have a syntax that is similar to the English language, therefore, they can be understood by programmers and users alike.
Dependency	Machine languages are platform -dependent and their features vary accordingly.	Assembly language comprises of standard instruction sets.

Basis of Difference	Machine Language	Assembly Language
Areas of application	Machine language serves as a machine code only.	Assembly languages are used for real-time systems and microprocessor-based applications/ devices.
Usage of mnemonics	Machine language uses sequences of bits for giving commands. One depicts the true or on state; on the other hand, zero depicts the false or off state. The conversion of high-level programming language to machine language is dependent on the CPU. “Mnemonics” are not required in machine language.	Assembly language does not require users to remember op-codes. It uses “mnemonics” names and symbols rather than raw sequences of bits. The codes in assembly languages are slightly more readable and can be mapped to machine code by humans.
Generation of programming language	Machine languages are first generation programming languages.	Assembly languages are second generation programming languages.
Modification	Machine language do not support any type of modification.	An assembly programming language can be modified easily.
Risk of errors	The risk of errors existing in the syntax of machine language is high.	The risk of errors existing in assembly language is comparatively low.
Memorization	Binary codes cannot be memorized.	It is possible to memorize the commands given in assembly languages.
Compiler	No compiler is necessary for executing commands.	A compiler, also known as an assembler, is needed for the proper execution of assembly language commands.

4.14 Discuss about regular expressions

The Structure of a Regular Expression

There are three important parts to a regular expression. **Anchors** are used to specify the position of the pattern in relation to a line of text. **Character Sets** match one or more characters in a single position. **Modifiers** specify how many times the previous character set is repeated.

A **regular expression**, **regex** or **regexp**[1] (sometimes called a **rational expression**)[2][3] is a sequence of characters that define a *search pattern*. Usually such patterns are used by string searching algorithms for "find" or "find and replace" operations on strings, or for input validation. It is a technique developed in theoretical computer science and formal language theory. Regular expressions are used in search engines, search and replace dialogs of word processors and text editors, in text processing utilities such as sed and AWK and in lexical analysis.
Many programming languages provide regex capabilities either built-in or via libraries.

A regular expression, often called a **pattern**, is an expression used to specify a set of strings required for a particular purpose. A simple way to specify a finite set of strings is to list its elements or members. However, there are often more concise ways to specify the desired set of strings. For example, the set containing the three strings "Handel", "Händel", and "Haendel" can be specified by the **pattern** `H(ä|ae?)ndel`; we say that this pattern **matches** each of the three strings. In most formalisms, if there exists at least one regular expression that matches a particular set then there exists an infinite number of other regular expressions that also match it—the specification is not unique. Most formalisms provide the following operations to construct regular expressions.

Boolean "or"

A vertical bar separates alternatives. For example, `gray|grey` can match "gray" or "grey".

Grouping

Parentheses are used to define the scope and precedence of the operators (among other uses). For example, `gray|grey` and `gr(a|e)y` are equivalent patterns which both describe the set of "gray" or "grey".

Quantification

A quantifier after a token (such as a character) or group specifies how often that a preceding element is allowed to occur. The most common quantifiers are the question mark `?`, the asterisk `*` (derived from the Kleene star), and the plus sign `+` (Kleene plus).

`?`	The question mark indicates *zero or one* occurrences of the preceding element. For example, `colou?r` matches both "color" and "colour".
`*`	The asterisk indicates *zero or more* occurrences of the preceding element. For example, `ab*c` matches "ac", "abc", "abbc", "abbbc", and so on.
`+`	The plus sign indicates *one or more* occurrences of the preceding element. For example, `ab+c` matches "abc", "abbc", "abbbc", and so on, but not "ac".
`{n}`[19]	The preceding item is matched exactly *n* times.
`{min,}`[19]	The preceding item is matched *min* or more times.
`{min,max}`[19]	The preceding item is matched at least *min* times, but not more than *max* times.

Wildcard

The wildcard `.` matches any character. For example, `a.b` matches any string that contains an "a", then any other character and then a "b", `a.*b` matches any string that contains an "a" and a "b" at some later point.

These constructions can be combined to form arbitrarily complex expressions, much like one can construct arithmetical expressions from numbers and the operations **+**, **−**, **×**, and **÷**. For example, `H(ae?|ä)ndel` and `H(a|ae|ä)ndel` are both valid patterns which match the same strings as the earlier example, `H(ä|ae?)ndel`.

The precise syntax for regular expressions varies among tools and with context; more detail is given in the *Syntax* section.

A **regex processor** translates a regular expression in the above syntax into an internal representation which can be executed and matched against a string representing the text being

searched in. One possible approach is the Thompson's construction algorithm to construct a nondeterministic finite automaton (NFA), which is then made deterministic and the resulting deterministic finite automaton (DFA) is run on the target text string to recognize substrings that match the regular expression.

4.15) Give the parse tree for the following statement: A=(B+C)*(D/E)

4.16) Difference between top down and bottom up parsing

There are 2 types of Parsing Technique present in parsing, first one is Top-down parsing and second one is Bottom-up parsing.
Top-down Parsing is a parsing technique that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar
while **Bottom-up Parsing** is a parsing technique that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
There are some differences present to differentiate these two parsing techniques, which are given below:

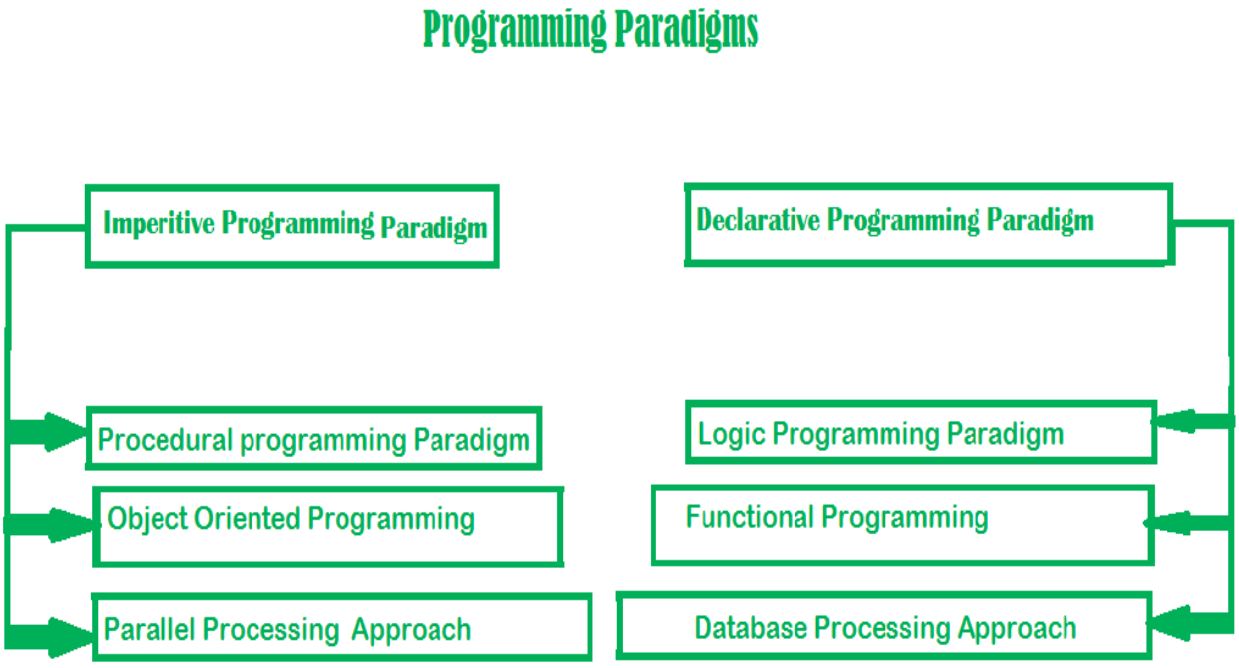
S.NO	TOP DOWN PARSING	BOTTOM UP PARSING
1.	It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
2.	Top-down parsing attempts to find the left most derivations for an input string.	Bottom-up parsing can be defined as an attempts to reduce the input string to start symbol of a grammar.
3.	In this parsing technique we start parsing from top (start symbol of parse tree) to down (the leaf node of parse tree) in top-down manner.	In this parsing technique we start parsing from bottom (leaf node of parse tree) to up (the start symbol of parse tree) in bottom-up manner.
4.	This parsing technique uses Left Most Derivation.	This parsing technique uses Right Most Derivation.

5.	It's main decision is to select what production rule to use in order to construct the string.	It's main decision is to select when to use a production rule to reduce the string to get the starting symbol.

1. The top-down parsing starts with the root while bottom-up begin the generation of the parse tree from the leaves.
2. Top-down parsing can be done in two ways with backtracking and without backtracking where leftmost derivation is used. On the other hand, the bottom-up parsing uses a shift-reduce method where the symbol is pushed and then popped from the queue. It employs rightmost derivation.
3. Bottom-up parsing is more powerful than the top-down parsing.
4. It is difficult to produce a bottom-up parser. As against, a top-down parser can be easily structured and formed.

4.17) Main features of programming paradigm

Paradigm can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach. There are lots for programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming language there are lots of paradigms to fulfil each and every demand. They are discussed below:



1. Imperative programming paradigm:
It is one of the oldest programming paradigm. It features close relation relation to machine architecture. It is based on Von Neumann architecture. It works by changing the program state through assignment statements. It performs step by step task by changing state. The main focus is on how to achieve the goal. The paradigm consist of several statements and after execution of all the result is stored.

Advantage:
1. Very simple to implement
2. It contains loops, variables etc.

Disadvantage:
1. Complex problem cannot be solved
2. Less efficient and less productive
3. Parallel programming is not possible

```
Examples of Imperative programming paradigm:

C : developed by Dennis Ritchie and Ken Thompson
Fortan : developed by John Backus for IBM
Basic : developed by John G Kemeny and Thomas E Kurtz
```

Imperative programming is divided into three broad categories: Procedural, OOP and parallel processing. These paradigms are as follows:

1. **Procedural programming paradigm –**
   This paradigm emphasizes on procedure in terms of under lying machine model. There is no difference in between procedural and imperative approach. It has the ability to reuse the code and it was boon at that time when it was in use because of its reusability.

```
Examples of Procedural programming paradigm:

C : developed by Dennis Ritchie and Ken Thompson
C++ : developed by Bjarne Stroustrup
Java : developed by James Gosling at Sun Microsystems
ColdFusion : developed by J J Allaire
Pascal : developed by Niklaus Wirth
```

8. **Object oriented programming –**
   The program is written as a collection of classes and object which are meant for communication. The smallest and basic entity is object and all kind of computation is performed on the objects only. More emphasis is on data rather procedure. It can handle almost all kind of real life problems which are today in scenario.
   Advantages:
   - Data security
   - Inheritance
   - Code reusability
   - Flexible and abstraction is also present

```
Examples of Object Oriented programming paradigm:

Simula : first OOP language
Java : developed by James Gosling at Sun Microsystems
```

```
C++ : developed by Bjarne Stroustrup
Objective-C : designed by Brad Cox
Visual Basic .NET : developed by Microsoft
Python : developed by Guido van Rossum
Ruby : developed by Yukihiro Matsumoto
Smalltalk : developed by Alan Kay, Dan Ingalls, Adele Goldberg
```

9. **Parallel processing approach –**
   Parallel processing is the processing of program instructions by dividing them among multiple processors. A parallel processing system posses many numbers of processor with the objective of running a program in less time by dividing them. This approach seems to be like divide and conquer. Examples are NESL (one of the oldest one) and C/C++ also supports because of some library function.

2. Declarative programming paradigm:

It is divided as Logic, Functional, Database. In computer science the *declarative programming* is a style of building programs that expresses logic of computation without talking about its control flow. It often considers programs as theories of some logic.It may simplify writing parallel programs. The focus is on what needs to be done rather how it should be done basically emphasize on what code code is actually doing. It just declare the result we want rather how it has be produced. This is the only difference between imperative (how to do) and declarative (what to do) programming paradigms. Getting into deeper we would see logic, functional and database.

1. **Logic programming paradigms –**
   It can be termed as abstract model of computation. It would solve logical problems like puzzles, series etc. In logic programming we have a knowledge base which we know before and along with the question and knowledge base which is given to machine, it produces result. In normal programming languages, such concept of knowledge base is not available but while using the concept of artificial intelligence, machine learning we have some models like Perception model which is using the same mechanism.
   In logical programming the main emphasize is on knowledge base and the problem. The execution of the program is very much like proof of mathematical statement, e.g., Prolog
2. **Functional programming paradigms –**
   The functional programming paradigms has its roots in mathematics and it is language independent. The key principal of this paradigms is the execution of series of mathematical functions. The central model for the abstraction is the function which are meant for some specific computation and not the data structure. Data are loosely coupled to functions.The function hide their implementation. Function can be replaced with their values without changing the meaning of the program. Some of the languages like perl, javascript mostly uses this paradigm.

```
3. Examples of Functional programming paradigm:
4.
5. JavaScript : developed by Brendan Eich
6. Haskwell : developed by Lennart Augustsson, Dave Barton
7. Scala : developed by Martin Odersky
8. Erlang : developed by Joe Armstrong, Robert Virding
9. Lisp : developed by John Mccarthy
```

10. **ML** : developed by Robin Milner
Clojure : developed by Rich Hickey

The next kind of approach is of Database.

3. Database/Data driven programming approach –
This programming methodology is based on data and its movement. Program statements are defined by data rather than hard-coding a series of steps. A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions. There are several programming languages that are developed mostly for database application. For example SQL. It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs. So it has its own wide application.

4.18) Features of logic programming

Logic programming is a computer programming paradigm in which program statements express facts and rules about problems within a system of formal logic. Rules are written as logical clauses with a head and a body; for instance, "H is true if B1, B2, and B3 are true." Facts are expressed similar to rules, but without a body; for instance, "H is true."

Some logic programming languages, such as Datalog and ASP (Answer Set Programming), are purely declarative. They allow for statements about what the program should accomplish, with no explicit step-by-step instructions on how to do so. Others, such as Prolog, are a combination of declarative and imperative. They may alsoinclude procedural statements, such as "To solve H, solve B1, B2, and B3."

Logical

- ***Introduction:***
  The *Logical Paradigm* takes a declarative approach to problem-solving. Various logical assertions about a situation are made, establishing all known facts. Then queries are made. The role of the computer becomes maintaining data and logical deduction.
- ***Logical Paradigm Programming:***
  A logical program is divided into three sections:
  1. **a series of definitions/declarations that define the problem domain**
  2. **statements of relevant facts**
  3. **statement of goals in the form of a query**

Any deducible solution to a query is returned. The definitions and declarations are constructed entirely from relations. i.e. X is a member of Y or X is in the internal between a and b etc.

- ***Advantages:***
  The advantages of logic oriented programming are bifold:
  1. **The system solves the problem, so the programming steps themselves are kept to a minimum;**
  2. **Proving the validity of a given program is simple.**

4.19) Define syntax and semantics of a language - Check from stages of compiler design.

4.21) Discuss the evolution of Object Oriented Language with reference to languages evolved.- Same as 4.22, write the answer in the form of evolution and not difference.

4.22) Disadvantages of Procedural language in comparison with OOL.

- Perhaps the most serious limitation is the tendency for large procedural-based programs to turn into "spaghetti-code". **Spaghetti code** is code that has been modified so many times that the logical flow shown in the figures above becomes so convoluted that any new programmer coming onto the project needs a two month prep-course in order to even begin to understand the software innards.
- In order to meet the demands of the evil user, the programmer is forced to modify the code. This can mean introducing new sub loops, new eddies of flow control and new methods, libraries and variables altogether. There are no great tools for **abstraction and modularization** in procedural languages...thus, it is hard to add new functionality or change the work flow without going back and modifying all other parts of the program.
- This gets us to the second problem with procedural-based programming. Not only does procedural code have a tendency to be difficult to understand, as it evolves, it becomes even harder to understand, and thus, harder to modify.
- Since everything is tied to everything else, **nothing is independent**. If you change one bit of code in a procedural-based program, it is likely that you will break three other pieces in some other section that might be stored in some remote library file you'd forgotten all about.
- A final problem with spaghetification, is that the code you write today will not help you write the code you have to write tomorrow. Procedural-based code has a tenacious ability to resist being cut and pasted from one application to another. Thus, procedural programmers often find themselves reinventing the wheel on every new project.
- Procedural languages does not have **automatic memory management** as like in Java. Hence, it makes the programmer to concern more about the memory management of the program.

Advantages

POP (Procedure Oriented Programming)

- Provides an ability to reuse the same code at various places.
- Facilitates in tracking the program flow.
- Capable of constructing modules.

OOP (Object Oriented Programming)

- Objects help in task partitioning in the project.
- Secure programs can be built using data hiding.
- It can potentially map the objects.
- Enables the categorization of the objects into various classes.
- Object-oriented systems can be upgraded effortlessly.
- Redundant codes can be eliminated using inheritance.
- Codes can be extended using reusability.
- Greater modularity can be achieved.
- Data abstraction increases reliability.
- Flexible due to the dynamic binding concept.
- Decouples the essential specification from its implementation by using information hiding.

Disadvantages

POP (Procedure Oriented Programming

- Global data are vulnerable.
- Data can move freely within a program
- It is tough to verify the data position.
- Functions are action-oriented.
- Functions are not capable of relating to the elements of the problem.
- Real-world problems cannot be modelled.
- Parts of code are interdependent.
- One application code cannot be used in other application.
- Data is transferred by using the functions.

OOP (Object Oriented Programming)

- It requires more resources.
- Dynamic behaviour of objects requires RAM storage.
- Detection and debugging is harder in complex applications when the message passing is performed.
- Inheritance makes their classes tightly coupled, which affects the reusability of objects.

BASIS FOR COMPARISON	POP	OOP
Basic	Procedure/Structure oriented .	Object oriented.
Approach	Top-down.	Bottom-up.
Basis	Main focus is on "how to get the task done" i.e. on the procedure or structure of a program .	Main focus is on 'data security'. Hence, only objects are permitted to access the entities of a class.
Division	Large program is divided into units called functions.	Entire program is divided into objects.
Entity accessing mode	No access specifier observed.	Access specifier are "public", "private", "protected".
Overloading/Polymorphism	Neither it overload functions nor operators.	It overloads functions, constructors, and operators.
Inheritance	Their is no provision of inheritance.	Inheritance achieved in three modes public private and protected.
Data hiding & security	There is no proper way of hiding the data, so data is insecure	Data is hidden in three modes public, private, and protected. hence data security increases.
Data sharing	Global data is shared among the functions in the program.	Data is shared among the objects through the member functions.
Friend functions/classes	No concept of friend function.	Classes or function can become a friend of another class with the keyword "friend". Note: "friend" keyword is used only in c++
Virtual classes/ function	No concept of virtual classes .	Concept of virtual function appear during inheritance.
Example	C, VB, FORTRAN, Pascal	C++, JAVA, VB.NET, C#.NET.

4.24) Discuss the working principle of syntax analyser - Check from stages of compiler design.

4.25) Discuss the difference b/w working principle of a compiler and interpreter.

BASIS FOR COMPARISON	COMPILER	INTERPRETER
Input	It takes an entire program at a time.	It takes a single line of code or instruction at a time.
Output	It generates intermediate object code.	It does not produce any intermediate object code.
Working mechanism	The compilation is done before execution.	Compilation and execution take place simultaneously.
Speed	Comparatively faster	Slower
Memory	Memory requirement is more due to the creation of object code.	It requires less memory as it does not create intermediate object code.
Errors	Display all errors after compilation, all at the same time.	Displays error of each line one by one.
Error detection	Difficult	Easier comparatively
Pertaining Programming languages	C, C++, C#, Scala, typescript uses compiler.	PHP, Perl, Python, Ruby uses an interpreter.

4.23) **Name some object oriented language and their advantages**

C++ - **C++** (/ˌsiːˌplʌsˈplʌs/) is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes". The language has expanded significantly over time, and modern C++ has object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Oracle, and IBM, so it is available on many platforms.[6]

C++ was designed with a bias toward system programming and embedded, resource-constrained software and large systems, with performance, efficiency, and flexibility of use as its design highlights.[7] C++ has also been found useful in many other contexts, with key strengths being software infrastructure and resource-constrained applications,[7] including desktop applications, servers (e.g. e-commerce, Web search, or SQL servers), and performance-critical applications

Java - Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]). With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. For example: J2EE for Enterprise Applications, J2ME for Mobile Applications.

Java is −

- **Object Oriented** − In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent** − Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
- **Simple** − Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **Secure** − With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architecture-neutral** − Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable** − Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **Robust** − Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded** − With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** − Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- **High Performance** − With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed** − Java is designed for the distributed environment of the internet.
- **Dynamic** − Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Python – Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands. Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages. Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.

1. Presence of Third Party Modules:

The Python Package Index (PyPI) contains numerous third-party modules that make Python capable of interacting with most of the other languages and platforms.

2. Extensive Support Libraries:

Python provides a large standard library which includes areas like internet protocols, string operations, web services tools and operating system interfaces. Many high use programming tasks have already been scripted into the standard library which reduces length of code to be written significantly.

3. Open Source and Community Development:

Python language is developed under an OSI-approved open source license, which makes it free to use and distribute, including for commercial purposes.

Further, its development is driven by the community which collaborates for its code through hosting conferences and mailing lists, and provides for its numerous modules.

4. Learning Ease and Support Available:

Python offers excellent readability and uncluttered simple-to-learn syntax which helps beginners to utilize this programming language. The code style guidelines, PEP 8, provide a set of rules to facilitate the formatting of code. Additionally, the wide base of users and active developers has resulted in a rich internet resource bank to encourage development and the continued adoption of the language.

5. User-friendly Data Structures:

Python has built-in list and dictionary data structures which can be used to construct fast runtime data structures. Further, Python also

provides the option of dynamic high-level data typing which reduces the length of support code that is needed.

6. Productivity and Speed:

Python has clean object-oriented design, provides enhanced process control capabilities, and possesses strong integration and text processing capabilities and its own unit testing framework, all of which contribute to the increase in its speed and productivity. Python is considered a viable option for building complex multi-protocol network applications.

Ruby is a dynamic, object-oriented scripting language designed to make programming **faster** and more **productive**. It runs on the back end of a site, building the brains of your site that users don't see. Back-end scripting languages run a site's functionality, connect it with a database, and run on a server. Ruby excels in this phase of development with its full-stack framework, Ruby on Rails. Full stack means it's got everything built in that a site needs to get started. Ruby on Rails is definitely not minimalist—it's got it all.

RUBY IS…

- Short and simple code that's concise, readable, and consistent.
- Great for startups. **Scalability** is a big selling point with languages like Ruby and Python. For startups, Ruby enables you to focus on your differentiators—short-term, creative solutions you'd rather put energy into than time-consuming, back-end programming.
- **Used for high-traffic sites**. Like Java, Ruby is frequently used on web servers that deal with a large amount of traffic like Twitter and GitHub.
- **Powered by the Ruby on Rails framework.** In the context of the Rails software library, Ruby gets even easier to work with thanks to bundles of code streamlining more complicated projects.
- **Similar to, but different than, Python**. Both are high-level, back-end scripting languages that can be used interchangeably in a stack, but where Ruby differs is its language or syntax. It's pliable and flexible vs. Python's "one right way to program" syntax. In Ruby, there are multiple ways to do the same thing, and some may be faster than others.
- **Designed for programmer happiness**. Things like "comfort and convenience" make it a draw for programmers, and by taking the best of all other languages that came before it, it embraces many different styles of programming in one.
- **Updated regularly, but not too regularly**. Languages like JavaScript are always changing while languages like Perl go for longer periods without an update. Developers have to keep up with new versions.

- **An interpreted scripting language**. Ruby (and Python) doesn't require a compiler, although **compiled** languages like Scala or C/C++ tend to run faster. *Rule of thumb*: what you get in speed of development, you lose in runtime speed.

4.35) characteristics of an algorithm

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

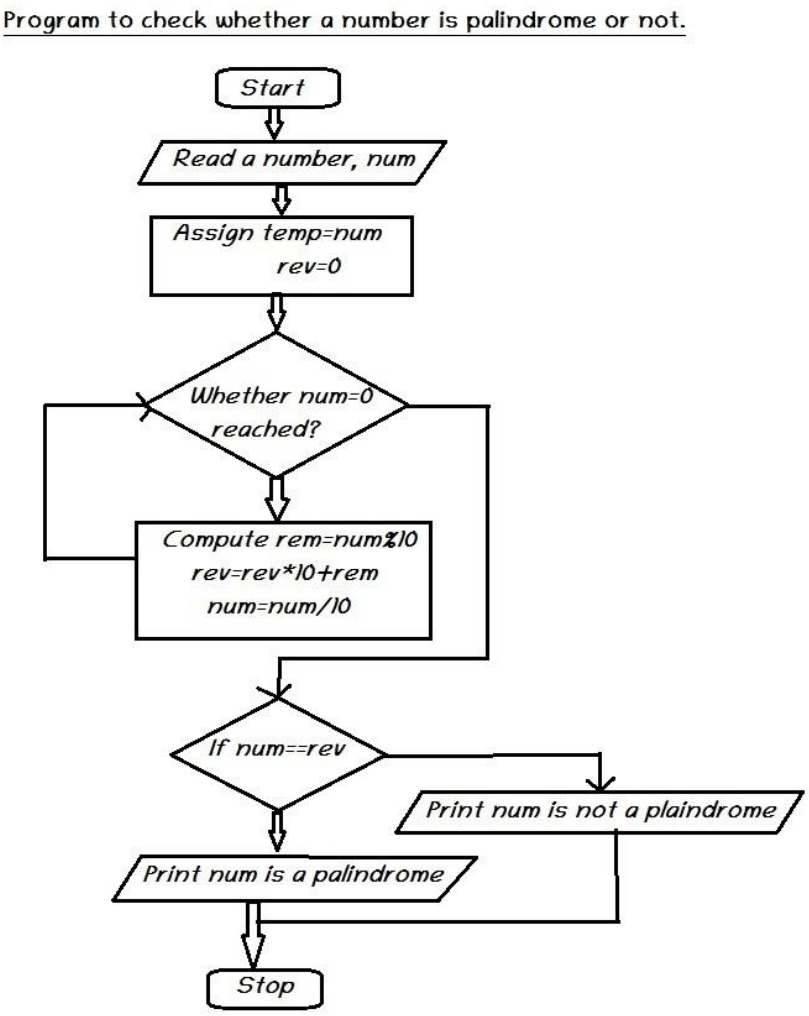
- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

4.36) General Algorithm for sum of digits in a given number

1. Get the number
2. Declare a variable to store the sum and set it to 0
3. Repeat the next two steps till the number is not 0
4. Get the rightmost digit of the number with help of remainder '%' operator by dividing it with 10 and add it to sum.
5. Divide the number by 10 with help of '/' operator
6. Print or return the sum OR

```
Step 1:  Input N
Step 2:  Sum = 0
Step 3:  While (N != 0)
                Rem = N % 10;
                 Sum = Sum + Rem;
                 N = N / 10;
Step 4:  Print Sum
```

4.37) Flowchart to check a number is palindrome or not.



4.38) Describe symbols used in a flowchat.

1. The Oval

An End or a Beginning

The oval, or **terminator,** is used to represent the start and end of a process. Use the Gliffy flowchart tool to drag and drop one of these bad boys and you've got yourself the beginning of a flowchart. Remember to use the same symbol again to show that your flowchart is complete.

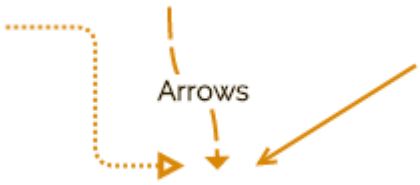
2. The Rectangle

A Step in the Flowcharting Process

The rectangle is your go-to symbol once you've started flowcharting. It represents any step in the process you're diagramming and is the workhorse of the flowchart diagram. Use rectangles to capture **process steps** like basic tasks or actions in your process.

3. The Arrow

Indicate Directional Flow



The **arrow** is used to guide the viewer along their flowcharting path. And while there are many different types of arrow tips to choose from, we recommend sticking with one or two for your entire flowchart. This keeps your diagram looking clean, but also allows you to emphasize certain steps in your process.

4. The Diamond

Indicate a Decision

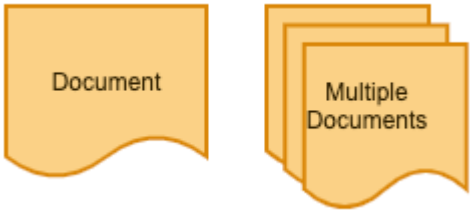
The diamond symbolizes that a **decision** is required to move forward. This could be a binary, this-or-that choice or a more complex decision with multiple choices. Make sure that you capture each possible choice within your diagram.

With those four basic symbols, you likely have everything you need to get started on your own flowchart! Give it a try with Gliffy or read on for more info on intermediate flowcharting symbols.

Intermediate & Advanced Flowchart Symbols

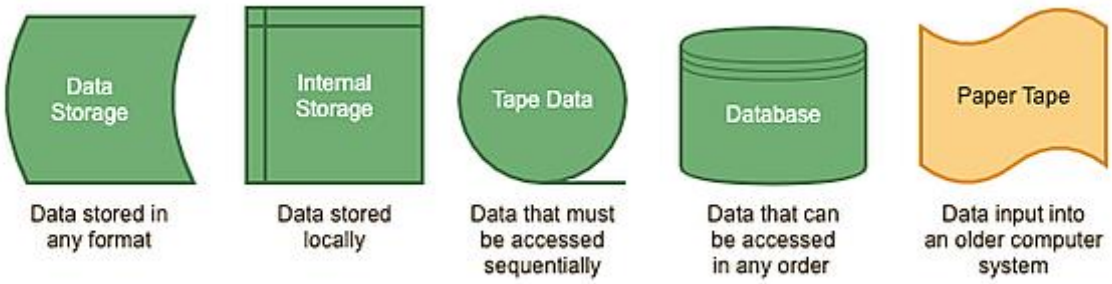
As you know, flowcharts are made up of a sequence of actions, data, services, and/or materials. They illustrate where data is being input and output, where information is being stored, what decisions need to be made, and which people need to be involved. In addition to the basics, these intermediate flowchart symbols will help you describe your process with even more detail.

Document Symbols



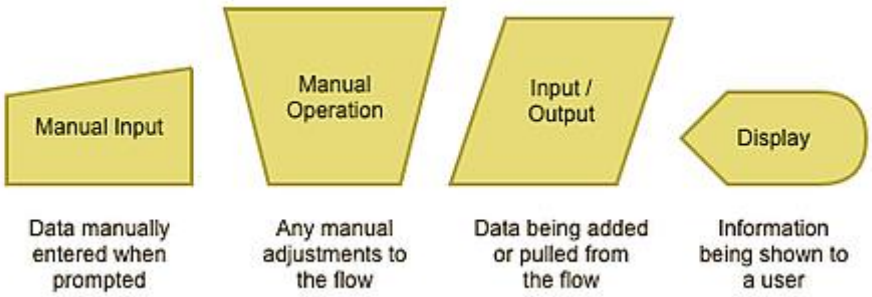
Single and multiple document icons show that there are additional points of reference involved in your flowchart. You might use these to indicate items like “create an invoice” or “review testing paperwork.”

Data Symbols



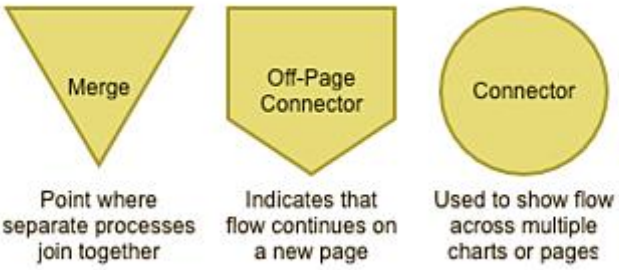
Data symbols clarify where the data your flowchart references is being stored. (You probably won't use the paper tape symbol, but it definitely came in handy back in the day.)

Input & Output



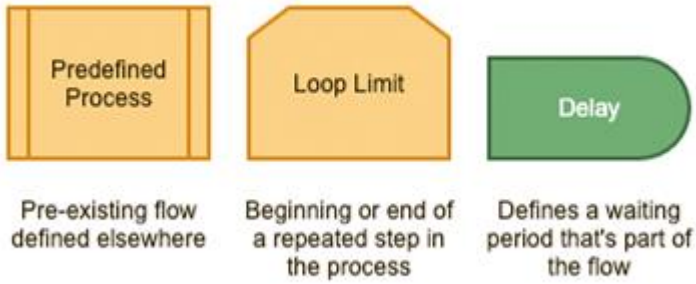
Input and output symbols show where and how data is coming in and out throughout your process.

Merging & Connecting

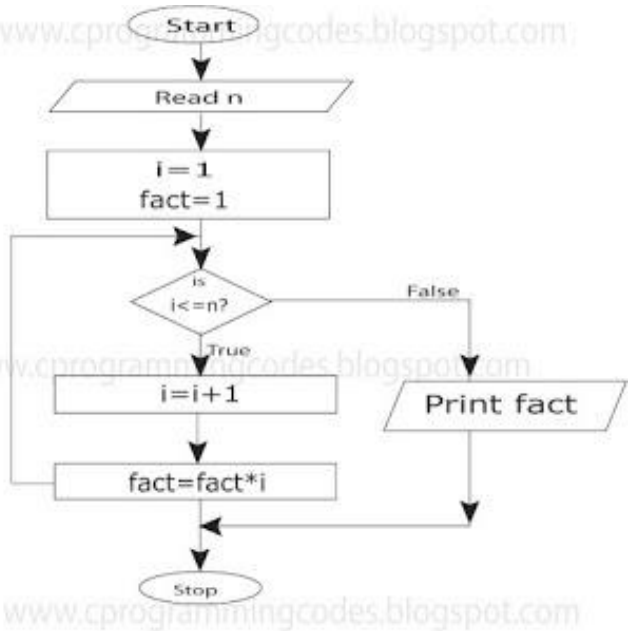


Agreed-upon merging and connector symbols make it easier to connect flowcharts that span multiple pages.

Additional Useful Shapes



4.39) Algorithm to find factorial of a number.




```
[algorithm to calculate
the factorial of a number]
step 1. Start
step 2. Read the number n
step 3. [Initialize]
        i=1, fact=1
step 4. Repeat step 4
through 6 until i=n
step 5. fact=fact*i
step 6. i=i+1
step 7. Print fact
step 8. Stop
[process finish of
calculate the factorial
value of a number]
```

4.40) Algorithm to calculate simple interest

4.41) Describe time complexity of an algorithm with example.

What is Time Complexity?

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**. It's an asymptotic notation to represent the time complexity. We will study about it in detail in the next tutorial.

Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution. Like in the example above, for the first code the loop will run `n` number of times, so the time complexity will be `n` atleast and as the value of `n` will increase the time taken will also increase. While for the second code, time complexity is constant, because it will never be dependent on the value of `n`, it will always give the result in 1 step.

And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)
{
    statement;
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)
{
    for(j=0; j < N;j++)
    {
    statement;
    }
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while(low <= high)
{
    mid = (low + high) / 2;
    if (target < list[mid])
        high = mid - 1;
    else if (target > list[mid])
        low = mid + 1;
    else break;
}
```

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)
{
    int pivot = partition(list, left, right);
    quicksort(list, left, pivot - 1);
    quicksort(list, pivot + 1, right);
}
```

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE: In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

Types of Notations for Time Complexity

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.
4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

Understanding Notations of Time Complexity with Example

O(expression) is the set of functions that grow slower than or at the same rate as expression. It indicates the maximum required by an algorithm for all input values. It represents the worst case of an algorithm's time complexity.

Omega(expression) is the set of functions that grow faster than or at the same rate as expression. It indicates the minimum time required by an algorithm for all input values. It represents the best case of an algorithm's time complexity.

Theta(expression) consist of all the functions that lie in both O(expression) and Omega(expression). It indicates the average bound of an algorithm. It represents the average case of an algorithm's time complexity.

Suppose you've calculated that an algorithm takes f(n) operations, where,

```
f(n) = 3*n^2 + 2*n + 4.   // n^2 means square of n
```

Since this polynomial grows at the same rate as **n^2**, then you could say that the function **f** lies in the set **Theta(n^2)**. (It also lies in the sets **O(n^2)** and **Omega(n^2)** for the same reason.)

The simplest explanation is, because **Theta** denotes *the same as* the expression. Hence, as **f(n)** grows by a factor of **n^2**, the time complexity can be best represented as **Theta(n^2)**.

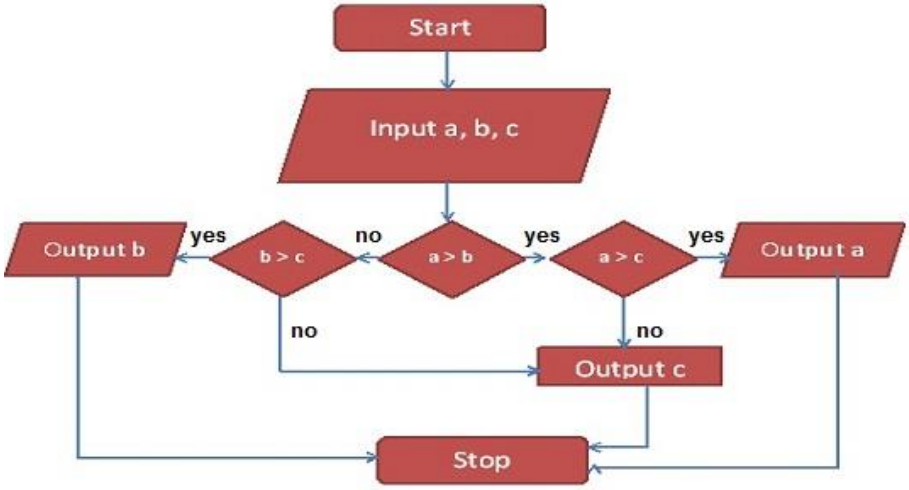
Imagine a classroom of 100 students in which you gave your pen to one person. Now, you want that pen. Here are some ways to find the pen and what the O order is.

O(n^2): You go and ask the first person of the class, if he has the pen. Also, you ask this person about other 99 people in the classroom if they have that pen and so on, This is what we call O(n^2).

O(n): Going and asking each student individually is O(N).
O(log n): Now I divide the class into two groups, then ask: “Is it on the left side, or the right side of the classroom?” Then I take that group and divide it into two and ask again, and so on. Repeat the process till you are left with one student who has your pen. This is what you mean by O(log n).
I might need to do the O(n^2) search if only one student knows on which student the pen is hidden. I’d use the O(n) if one student had the pen and only they knew it. I’d use the O(log n) search if all the students knew, but would only tell me if I guessed the right side.
4.42) Flowchart to find largest among three numbers

- Algorithm : a
    - Step 1 : Start
    - Start 2 : Input a, b, c
    - Start 3 : if a > b goto step 4, otherwise goto step 5
    - Start 4 : if a > c goto step 6, otherwise goto step 8
    - Start 5 : if b > c goto step 7, otherwise goto step 8
    - Start 6 : Output "a is the largest", goto step 9

- Start 7 : Output "b is the largest", goto step 9
- Start 8 : Output " c is the largest", goto step 9
- Start 9 : Stop



4.44) space complexity of an algorithm using examples.

Space Complexity of Algorithms

Auxiliary Space is the extra space or temporary space used by an algorithm.
Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.
For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criteria than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort and Heap Sort use O(1) auxiliary space. Space complexity of all these sorting algorithms is O(n) though.

But often, people confuse Space complexity with Auxiliary space. Auxiliary space is just a temporary or extra space and it is not the same as space complexity. In simpler terms,

Space Complexity = Auxiliary space + Space use by input values

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

Space complexity *is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.*

Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Space Complexity = **Auxiliary Space + Input space**

Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

   It's the amount of memory used to save the compiled version of instructions.

2. **Environmental Stack**

   Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

   For example, If a function `A()` calls function `B()` inside it, then all th variables of the function `A()` will get stored on the system stack temporarily, while the function `B()` is called and executed inside the funciton `A()`.

3. **Data Space**

   Amount of space used by the variables and constants.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space** and we neglect the **Instruction Space** and **Environmental Stack**.

Calculating the Space Complexity

For calculating the space complexity, we need to know the value of memory used by different type of datatype variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Type	**Size**
bool, char, unsigned char, signed char, __int8	1 byte

__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

```
#include<stdio.h>
int main()
{
    int a = 5, b = 5, c;
    c = a + b;
    printf("%d", c);
}
```

Output:

Explanation: Do not misunderstand space complexity to be 1364 Kilobytes as shown in the output image. The method to calculate the actual space complexity is shown below. In the above program, 3 integer variables are used. The size of the integer data type is 2 or 4 bytes which depends on the compiler. Now, let's assume the size as 4 bytes. So, the total space occupied by the above-given program is 4 * 3 = 12 bytes. Since no additional variables are used, no extra space is required. Hence, **space complexity for the above-given program is O(1), or constant.**

```
#include <stdio.h>
int main()
{
    int n, i, sum = 0;
    scanf("%d", &n);
    int arr[n];
    for(i = 0; i < n; i++)
    {
```

```
        scanf("%d", &arr[i]);
        sum = sum + arr[i];
    }
    printf("%d", sum);
}
```

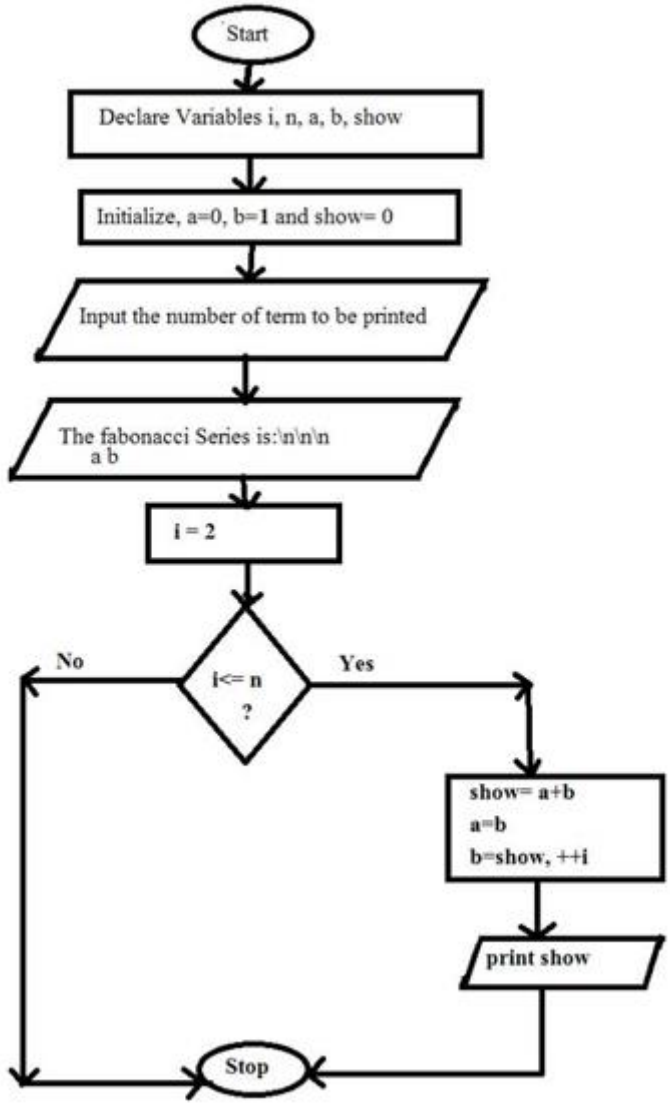
Output:

Explanation:

In the above-given code, the array consists of ‘n’ integer elements. So, the space occupied by the array is 4 * n. Also we have integer variables such as n, i and sum. Assuming 4 bytes for each variable, the total space occupied by the program is 4n + 12 bytes. Since the highest order of n in the equation 4n + 12 is n, so **the space complexity is O(n) or linear.**

Big O Notation	Space Complexity details
O(1)	Constant Space Complexity occurs when the program doesn’t contain any loops, recursive functions or call to any other functions.
O(n)	Linear space complexity occurs when the program contains any loops.

4.45) Draw a flowchart to find Fibonacci series



- Start
- Declare variables i,a,b, show
- Initialize the variables, a=0, b=1, and show =0
- Enter the number of terms of Fibonacci series to be printed
- Print First two terms of series
- Use loop for the following steps

-> show=a+b

-> a=b

-> b=show

-> increase value of i each time by 1

-> print the value of show

- End

4.46) difference between high and low level language

Low level language	High level language
They are faster than high level language.	They are comparatively slower.
Low level languages are memory efficient.	High level languages are not memory efficient.
Low level languages are difficult to learn.	High level languages are easy to learn.
Programming in low level requires additional knowledge of the computer architecture.	Programming in high level do not require any additional knowledge of the computer architecture.
They are machine dependent and are not portable.	They are machine independent and portable.
They provide less or no abstraction from the hardware.	They provide high abstraction from the hardware.
They are more error prone.	They are less error prone.
Debugging and maintenance is difficult.	Debugging and maintenance is comparatively easier.

They are generally used for developing system software's (Operating systems) and embedded applications.	They are used to develop a variety of applications such as – desktop applications, websites, mobile software's etc.

4.47) Difference between assembler and complier

BASIS FOR COMPARISON	COMPILER	ASSEMBLER
Basic	Generates the assembly language code or directly the executable code.	Generates the relocatable machine code.
Input	Preprocessed source code.	Assembly language code.
Phases/ Passes	The compilation phases are lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generation, code optimization, code generation.	Assembler makes two passes over the given input.
Output	The assembly code generated by the compiler is a mnemonic version of machine code.	The relocatable machine code generated by an assembler is represented by binary code.

4.48) Discuss the process of compiling and running a c program.

Pre-processing

This is the first phase through which source code is passed. This phase include:

- Removal of Comments
- Expansion of Macros
- Expansion of the included files.
- Conditional compilation

The preprocessed output is stored in the **filename.i**. Let's see what's inside filename.i: using **$vi filename.i**

In the above output, source file is filled with lots and lots of info, but at the end our code is preserved.

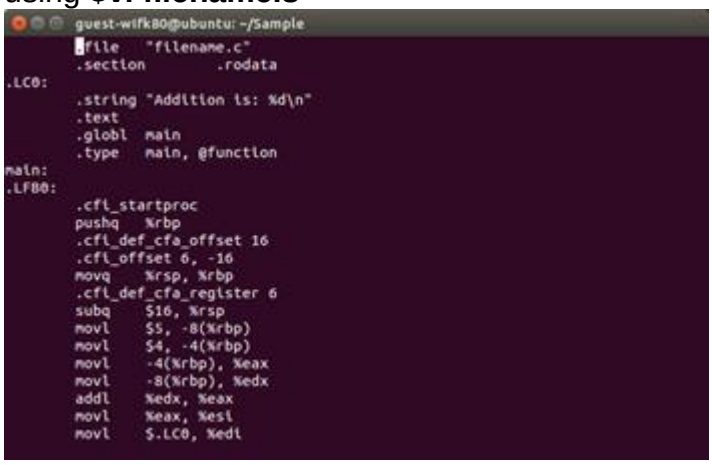
Analysis:

- printf contains now a + b rather than add(a, b) that's because macros have expanded.

- Comments are stripped off.
- **#include<stdio.h>** is missing instead we see lots of code. So header files has been expanded and included in our source file.

Compiling

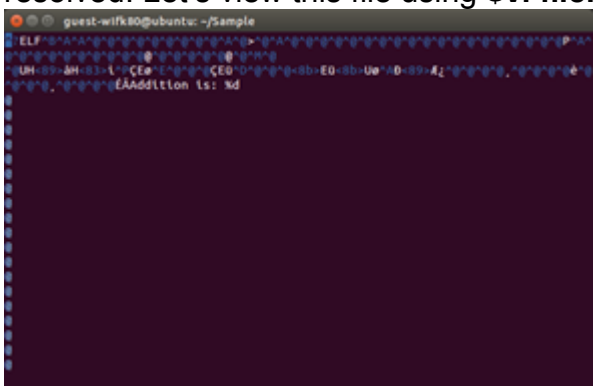
The next step is to compile filename.i and produce an; intermediate compiled output file **filename.s**. This file is in assembly level instructions. Let's see through this file using **$vi filename.s**



The snapshot shows that it is in assembly language, which assembler can understand.

Assembly

In this phase the filename.s is taken as input and turned into **filename.o** by assembler. This file contain machine level instructions. At this phase, only existing code is converted into machine language, the function calls like printf() are not resolved. Let's view this file using **$vi filename.o**



Linking

This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends. For example, there is a code which is required for setting up the environment like passing command line arguments. This task can be easily verified by using **$size filename.o** and **$size filename**. Through these commands, we know that how output file increases from an object file to an executable file. This is because of the extra code that linker adds with our program.

10.28) Phases of Compiler – Check first answer.

5.49) features of structured programming

Structured Programming Approach, as the word suggests, can be defined as a programming approach in which the program is made as a single structure. It means that the code will execute the instruction by instruction one after the other. It doesn't support the possibility of jumping from one instruction to some other with the help of any statement like GOTO, etc. Therefore, the instructions in this approach will be executed in a serial and structured manner. In structured programming design, programs are broken into different functions these functions are also known as **modules**, **subprogram**, **subroutines** and **procedures**.

Each function is design to do a specific task with its own data and logic. Information can be passed from one function to another function through parameters. A function can have local data that cannot be accessed outside the function's scope. The result of this process is that all the other different functions are synthesized in an another function. This function is known as main function. Many of the high level languages supported structure programming.

Structured programming minimized the chances of the function affecting another. It supported to write clearer programs. It made global variables to disappear and replaced by the local variables. Due to this change one can save the memory allocation space occupied by the global variable

The languages that support Structured programming approach are:C,C++,Java

What are the main features of Structural Programming language?

1. Division of Complex problems into small procedures and functions.
2. No presence of GOTO Statement
3. The main statement include – If-then-else, Call and Case statements.
4. Large set of operators like arithmetic, relational, logical, bit manipulation, shift and part word operators.
5. Inclusion of facilities for implementing entry points and external references in program.
6. Easier to read and understand
7. User Friendly
8. Easier to Maintain
9. Mainly problem based instead of being machine based
10. Development is easier as it requires less effort and time
11. Easier to Debug
12. Machine-Independent, mostly.

5.64) how constants are defined in c

There are many different ways to make the variable as constant

1. **Using const keyword:** The const keyword specifies that a variable or object value is constant and can't be modified at the compilation time.

```
// C program to demonstrate const specifier
#include <stdio.h>
int main()
{
    const int num = 1;

    num = 5; // Modifying the value
    return 0;
}
```

```
It will throw as error like:

error: assignment of read-only variable 'num'
```

2. **Using enum keyword:** Enumeration (or enum) is a user defined data type in C and C++. It is mainly used to assign names to integral constants, that make a program easy to read and maintain.

```
// In C and C++ internally the default
// type of 'var' is int
enum VARS { var = 42 };

// In C++ 11 (can have any integral type):
enum : type { var = 42; }

// where mytype = int, char, long etc.
// but it can't be float, double or
// user defined data type.
```

Note: The data types of **enum** are of course limited as we can see in above example.

3. **Using constexpr keyword:** Using constexpr in C++(not in C) can be used to declare variable as a guaranteed constant. But it would fail to compile if its initializer isn't a constant expression.

```
#include <iostream>

int main()
{
    int var = 5;
    constexpr int k = var;
    std::cout << k;
    return 0;
}
```

Above program will throw an error i.e.,

```
error: the value of 'var' is not usable in a constant expression
```

because the variable 'var' in not constant expression. Hence in order to make it as constant, we have to declare the variable 'var' with **const** keyword.

4. **Using Macros:** We can also use Macros to define constant, but there is a catch,

```
#define var 5
```

5.65) what is type casting with example

Typecasting is converting one data type into another one. It is also called as data conversion or type conversion.

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. This type of type conversion can be seen in the following example.

```
#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);
}
```

Output

```
10
10
```

In implicit type conversion, the data type is converted automatically. There are some scenarios in which we may have to force type conversion. Suppose we have a variable div that stores the division of two operands which are declared as an int data type.

```
 int result, var1=10, var2=3;
result=var1/var2;
```

In this case, after the division performed on variables var1 and var2 the result stored in the variable "result" will be in an integer format. Whenever this happens, the value stored in the variable "result" loses its meaning because it does not consider the fraction part which is normally obtained in the division of two numbers.

To force the type conversion in such situations, we use explicit type casting.

Let us write a program to demonstrate implementation of explicit type-casting in 'C'.

```
#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;
}
```

Output:

```
Value of a is 1.200000
Value of b is 2
```

5.66) Discuss storage classes in C

STORAGE CLASSES : -

What is a Storage Class?

Storage class in C decides the part of storage to allocate memory for a variable, it also determines the scope of a variable. All variables defined in a C program get some physical location in memory where variable's value is stored. Memory and CPU registers are types of memory locations where a variable's value can be stored. The storage class of a variable in C determines the life time of the variable if this is 'global' or 'local'. Along with the life time of a variable, storage class also determines variable's storage location (memory or registers), the scope (visibility level) of the variable, and the initial value of the variable. There are four storage classes in C those are *automatic*, *register*, *static*, and *external*.

Storage Class Specifiers

There are four storage class specifiers in C as follows, `typedef` specifier does not reserve storage and is called a storage class specifier only for syntactic convenience. It is not a storage class specifier in the common meaning.

- auto
- register
- extern
- static
- typedef

These specifiers tell the compiler how to store the subsequent variable. The general form of a variable declaration that uses a storage class is shown here:

```
storage_class_specifier data_type variable_name;
```

At most one storage class specifier may be given in a declaration. If no storage class specifier is specified then following rules are used:

1. Variables declared inside a function are taken to be `auto`.
2. Functions declared within a function are taken to be `extern`.
3. Variables and functions declared outside a function are taken to be `static`, with *external linkage*.

Variables and functions having *external linkage* are available to all files that constitute a program. File scope variables and functions declared as `static` (described shortly) have *internal linkage*. These are known only within the file in which they are declared. Local variables have no linkage and are therefore known only within their own block.

Types of Storage Classes

There are four storage classes in C they are as follows:

1. Automatic Storage Class
2. Register Storage Class
3. Static Storage Class
4. External Storage Class

Now, let us discuss these storage classes one by one.

1. Automatic Storage Class

A variable defined within a function or block with `auto` specifier belongs to automatic storage class. All variables defined within a function or block by default belong to automatic storage class if no storage class is mentioned. Variables having automatic storage class are local to the block which they are defined in, and get destroyed on exit from the block.

The following C program demonstrates the visibility level of `auto` variables.

```
#include <stdio.h>
int main( )
{
  auto int i = 1;
  {
    auto int i = 2;
    {
      auto int i = 3;
      printf ( "\n%d ", i);
    }
    printf ( "%d ", i);
  }
  printf( "%d\n", i);
}


OUTPUT
======
3 2 1
```

In above example program you see three definitions for variable `i`. Here, you may be thinking if there could be more than one variable with the same name. Yes, there could be if these variables are defined in different blocks. So, there will be no error here and the program will compile and execute successfully. The `printf` in the inner most block will print 3 and the variable `i` defined in the inner most block gets destroyed as soon as control exits from the block. Now control comes to the second outer block and prints 2 then comes to the outer block and prints 1. Here, note that automatic variables must always be initialized properly, otherwise you are likely to get unexpected results because automatic variables are not given any initial value by the compiler.

2. Register Storage Class

The `register` specifier declares a variable of register storage class. Variables belonging to register storage class are local to the block which they are defined in, and get destroyed on exit from the block. A `register` declaration is equivalent to an `auto` declaration, but hints that the declared variable will be accessed frequently; therefore they are placed in CPU registers, not in memory. Only a few variables are actually placed into registers, and only certain types are eligible; the restrictions are implementation-dependent. However, if a variable is declared register, the unary `&` (address of) operator may not be applied to it, explicitly or implicitly. Register variables are also given no initial value by the compiler.

The following piece of code is trying to get the address of variable `i` into pointer variable `p` but it won't succeed because `i` is declared `register`; therefore following piece of code won't compile and exit with error *"error: address of register variable requested"*.

```
#include <stdio.h>

int main()
{
  register int i = 10;
  int *p = &i; //error: address of register variable requested

  printf("Value of i: %d", *p);
  printf("Address of i: %u", p);

}
```

3. Static Storage Class

The `static` specifier gives the declared variable static storage class. Static variables can be used within function or file.Unlike global variables, static variables are not visible outside their function or file, but they maintain their values between calls. The `static` specifier has different effects upon local and global variables. See the following flavours of `static` specifier.

- When `static` specifier is applied to a local variable inside a function or block, the compiler creates permanent storage for it, much as it creates storage for a global variable but static local variable remains visible only to the function or block in which it is defined. In simple terms, a static local variable is a local variable that retains its value between function calls. For example, the following program code defines `static` variable `i` at two places in two blocks inside function `staticDemo()`. Function `staticDemo()` is called twice within from `main` function. During second call

static variables retain their old values and they are not initialized again in second call of `staticDemo()`.

```
#include <stdio.h>

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

int main()
{
  staticDemo();
  staticDemo();
}
OUTPUT
======
1 0
2 1
```

- When `static` specifier is applied to a global variable or a function then compiler makes that variable or function known only to the file in which it is defined. A static global variable has *internal linkage* that means even though the variable is global; routines in other files have no knowledge of it and cannot access and alter its contents directly. The following C program defines one static global variable `gInt` and a static function `staticDemo()`, for the variable and function are defined static they cannot be used outside the file (translation unit) `staticdemo.c`..

```
/* staticdemo.c */
#include <stdio.h>
static int gInt = 1;
static void staticDemo()
{
  static int i;
  printf("%d ", i);
  i++;
  printf("%d\n", gInt);
  gInt++;
}
```

```
int main()
{
  staticDemo();
  staticDemo();
}
OUTPUT
======
0 1
1 2
```

Static variables have default initial value zero and initialized only once in their lifetime.

4. External Storage Class

The `extern` specifier gives the declared variable external storage class. The principal use of `extern` is to specify that a variable is declared with *external linkage* elsewhere in the program. To understand why this is important, it is necessary to understand the difference between a declaration and a definition. A declaration declares the name and type of a variable or function. A definition causes storage to be allocated for the variable or the body of the function to be defined. The same variable or function may have many declarations, but there can be only one definition for that variable or function.

When `extern` specifier is used with a variable declaration then no storage is allocated to that variable and it is assumed that the variable has already been defined elsewhere in the program. When we use `extern` specifier the variable cannot be initialized because with `extern` specifier variable is declared, not defined.

In the following sample C program if you remove `extern int x;` you will get an error *"Undeclared identifier 'x'"* because variable `x` is defined later than it has been used in `printf`. In this example, the `extern` specifier tells the compiler that variable `x` has already been defined and it is declared here for compiler's information.

```
#include <stdio.h>

extern int x;

int main()
{
  printf("x: %d\n", x);
}

int x = 10;
```

Also, if you change the statement `extern int x;` to `extern int x = 50;` you will again get an error *"Redefinition of 'x'"* because with `extern` specifier the variable cannot be initialized, if it is defined elsewhere. If not then `extern` declaration becomes a definition.

Note that `extern` can also be applied to a function declaration, but doing so is redundant because all function declarations are implicitly `extern`.

5.67) rules to construct variable in c

Rules for constructing variable names
1. First character in a variable name must be an alphabet or an underscore(_).
2. Variable name can have alphabet, digits and underscore.
3. No commas, space allowed in variable name.
4. No other special symbols other than underscore is allowed.
5. C variables are case sensitive. Ex: name and Name are two different variables.
6. You can not use keywords / reserve words as variable name in C.

5.68) Data types in C with examples

11.79) Discuss about data type in c and derived data types.

A data-type in C programming is a set of values and is determined to act on those values. C provides various types of data-types which allow the programmer to select the appropriate type for the variable to set its value.

The data-type in a programming language is the collection of data with values having fixed meaning as well as characteristics. Some of them are an integer, floating point, character, etc. Usually, programming languages specify the range values for given data-type.

C Data Types are used to:

- Identify the type of a variable when it declared.
- Identify the type of the return value of a function.
- Identify the type of a parameter expected by a function.
    - Primary(Built-in) Data Types:
      void, *int*, *char*, *double* and *float*.
    - Derived Data Types:
      Array, *References*, and *Pointers*.
    - User Defined Data Types:
      Structure, *Union*, and *Enumeration*.
    - **char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
    - **int:** As the name suggests, an int variable is used to store an integer.
    - **float:** It is used to store decimal numbers (numbers with floating point value) with single precision.

- **double:** It is used to store decimal numbers (numbers with floating point value) with double precision.

Every C compiler supports five 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.4, long int 4, long long int 8
char	Used to denote a character type.1
float, double	Used to denote a floating point type. Float 4, double 8, long double 12
int *, float *, char *	Used to denote a pointer type.

Three more data types have been added in C99:

- _Bool
- _Complex
- _Imaginary

Declaration of Primary Data Types with Variable Names

After taking suitable variable names, they need to be assigned with a data type. This is how the data types are used along with variables:

Example:

```
int    age;

char   letter;

float  height, width;
```

Derived Data Types

C supports three derived data types:

Data Types	Description
Arrays	Arrays are sequences of data items having homogeneous values. They have adjacent memory locations to store values.
References	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.

FUNDAMENTAL DATA TYPES DERIVED DATA TYPESFundamental data type is also called primitive data type. These are the basic data types.Derived data type is the aggregation of fundamental data type.character, integer, float, and void are fundamental data types.Pointers, arrays, structures and unions are derived data types.Character is used for characters. It can be classified as char, Signed char, Unsigned char.Pointers are used for storing address of variables.Integer is used for integers(not having decimal digits). It can be classified as signed and unsigned. Further classified as int, short int and long int.Array is used to contain similar type of data.float is used for decimal numbers. These are classified as float, double and long double.structure is used to group items of possibly different types into a single type.void is used where there is no return value required.It is like structure but all members in union share the same memory location

Derived Types

There are five derived types in C:

- Function types
- Pointer types
- Array types
- Structure types
- Union types

The following sections describe these derived types.

A derived type is formed by using one or more basic types in combination. Using derived types, an infinite variety of new types can be formed. The array and structure types are collectively called the *aggregate* types. Note that the aggregate types do not include union types, but a union may contain an aggregate member.

3.4.1 Function Type

A function type describes a function that returns a value of a specified type. If the function returns no value, it should be declared as "function returning `void` " as follows:

```
void function1 ();
```

In the following example, the data type for the function is "function returning `int` ":

```
int uppercase(int lc)
{
  int uc = lc + 0X20;
  return uc;
}
```

Chapter 4 discusses declarations in general. Chapter 5 covers functions specifically, including their declarations, parameters, and argument passing.

3.4.2 Pointer Type

A pointer type describes a value that represents the address of an object of a stated type. A pointer is stored as an integral value that references the address of the target object. Pointer types are derived from other types, called the *referenced type* of the pointer. For example:

```
int *p;           /*  p is a pointer to an int type                */
double *q();      /*  q is a function returning a pointer to an
                      object of type double                        */
int (*r)[5];      /*  r is a pointer to an array of five elements  */
                  /*  (r holds the address to the first element of
                      the array)                                   */
const char s[6]; /*  s is a const-qualified array of 6 elements   */
```

The pointer itself can have any storage class, but the object addressed by the pointer cannot have the `register` storage class or be a bit field. Pointers to qualified or unqualified versions of compatible types have the same representation and alignment requirements as the target type. Pointers to other types need not have the same representation or alignment requirements.

The construction `void *` designates a generic "pointer to `void` " type. The `void *` construction can be used to point to an object of any type, and it is most useful when a pointer is needed to point to the address of objects with different or unknown types (such as in a function prototype). A pointer to `void` can also be converted to or from a pointer of any other type, and has the same representation and alignment requirements as a pointer to a character type.

A pointer to the address 0 (zero) is called a *null pointer*. Null pointers are often used to indicate that no more members of a list exist (for example, when using pointers to show the next member of the list). Dereferencing a null pointer with the * or subscripting operators leads to unpredictable and usually very unfavorable results.

See Chapter 4 for details on the syntax of pointer declarations.

3.4.3 Array Type

An array type can be formed from any valid completed type. Completion of an array type requires that the number and type of array members be explicitly or implicitly specified. The member types can be completed in the same or a different compilation unit. Arrays cannot be of `void` or function type, since the `void` type cannot be completed and function types are not object types requiring storage.

Typically, arrays are used to perform operations on some homogeneous set of values. The size of the array type is determined by the data type of the array and the number of elements in the array. Each element in an array has the same type. For example, the following definition creates an array of four characters:

```
char x[] = "Hi!"   /*  Declaring an array x   */;
```

Each of the elements has the size of a `char` object, 8 bits. The size of the array is determined by its initialization; in the previous example, the array has three explicit elements plus one null character. Four elements of 8 bits each results in an array with a size of 32 bits.

An array is allocated contiguously in memory, and cannot be empty (that is, have no members). An array can have only one dimension. To create an array of "two dimensions," declare an array of arrays, and so on.

It is possible to declare an array of unknown size; this sort of declaration is called an *incomplete array declaration*, because the size is not specified. The following example shows an incomplete declaration:

```
int x[];
```

The size of an array declared in this manner must be specified elsewhere in the program. (See Section 4.7 for more information on declaring incomplete arrays and initializing arrays.)

Character strings (string literals) are stored in the form of an array of `char` or `wchar_t` type, and are terminated by the null character (\0).

An array in C has only one dimension. An array of arrays can be declared, however, to create a *multidimensional array*. The elements of these arrays are stored in increasing addresses so that the rightmost subscript varies most rapidly. This is called *row- major order*, and is analogous to a car's odometer. For example, in an array of two arrays declared as `int a[2][3];` the elements are stored in this order:

```
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]
```

3.4.4 Structure Type

A structure type is a sequentially allocated nonempty set of objects, called *members*. Structures let you group heterogeneous data. They are much like records in Pascal. Unlike arrays, the elements of a structure need not be of the same data type. Also, elements of a structure are accessed by name, not by subscript. The following example declares a structure `employee` , with two structure variables (`ed` and `mary`) of the structure type `employee` :

```
struct employee { char name[30]; int age; int empnumber; };
struct employee ed, mary;
```

Structure members can have any type except an incomplete type, such as the `void` type or a function type. Structures can contain pointers to objects of their own type, but they cannot contain an object of their own type as a member; such an object would have an incomplete type. For example:

```
struct employee {
  char name[30];
  struct employee div1;       /*  This is invalid. */
  int *f();
};
```

The following example, however, is valid:

```
struct employee {
  char name[30];
  struct employee *div1;/*  Member can contain pointer to employee
                            structure.                              */
  int (*f)();           /*  Pointer to a function returning int     */
};
```

The name of a declared structure member must be unique within the structure, but it can be used in another nested or unnested structure or name spaces to refer to a different object. For example:

```
struct {
  int a;
  struct {
    int a;  /* This 'a' refers to a different object
               than the previous 'a'                */
  };
};
```

Chapter 4 contains more examples on structures and their declarations.

The compiler assigns storage for structure members in the order of member declaration, with increasing memory addresses for subsequent members. The first member always begins at the starting address of the structure itself. Subsequent members are aligned per the *alignment unit*, which may differ depending on the

member sizes in the structure. A structure may contain padding (unused bits) so that members of an array of such structures are properly aligned, and the size of the structure is the amount of storage necessary for all members plus any padded space needed to meet alignment requirements. See your system's DEC C documentation for platform-specific information about structure alignment and representation.

A pragma is available to change the alignment of a structure on one platform to match that of structures on other platforms. See Section B.29 for more information on this pragma.

3.4.5 Union Type

A union type can store objects of different types at the same location in memory. The different union members can occupy the same location at different times in the program. The declaration of a union includes all members of the union, and lists the possible object types the union can hold. The union can hold any one member at a time-subsequent assignments of other members to the union overwrite the existing object in the same storage area.

Unions can be named with any valid identifier. An empty union cannot be declared, nor can a union contain an instance of itself. A member of a union cannot have a `void` , function, or incomplete type. Unions can contain pointers to unions of their type.

Another way to look at a union is as a single object that can represent objects of different types at different times. Unions let you use objects whose type and size can change as the program progresses, without using machine-dependent constructions. Some other languages call this concept a *variant record*.

The syntax for defining unions is very similar to that for structures. Each union type definition creates a unique type. Names of union members must be unique within the union, but they can be duplicated in other nested or unnested unions or name spaces. For example:

```
union {
  int a;
  union {
    int a;  /* This 'a' refers to a different object
               than the previous 'a'                */
  };
};
```

The size of a union is the amount of storage necessary for its largest member, plus any padding needed to meet alignment requirements.

Once a union is defined, a value can be assigned to any of the objects declared in the union declaration. For example:

```
union name {
  double dvalue;
  struct x { int value1; int value2; };
  float fvalue;
} alberta;
alberta.dvalue = 3.141596; /* Assigns the value of pi to the union object
*/
```

Here, `alberta` can hold a `double` , `struct` , or `float` value. The programmer has responsibility for tracking the current type of object contained in the union. An assignment expression can be used to change the type of value held in the union.

Undefined behavior results when a union is used to store a value of one type, and then the value is accessed through another type.

User Defined Data Types

C allows the feature called *type definition* which allows programmers to define their identifier that would represent an existing data type. There are three such 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.

5.69) difference between local and global variables

BASIS FOR COMPARISON	LOCAL VARIABLE	GLOBAL VARIABLE
Declaration	Variables are declared inside a function.	Variables are declared outside any function.

BASIS FOR COMPARISON	LOCAL VARIABLE	GLOBAL VARIABLE
Scope	Within a function, inside which they are declared.	Throughout the program.
Value	Uninitialized local variable result in storage of the garbage value.	Uninitialized global variable stores zero by default.
Access	Accessed only by the statements, inside a function in which they are declared.	Accessed by any statement in the entire program.
Data sharing	Not provided	Facilitated
Life	Created when the function block is entered and destroyed upon exit.	Remain in existence for the entire time your program is executing.
Storage	Local variables are stored on the stack unless specified.	Stored on a fixed location decided by a compiler.
Parameter passing	Necessarily required	Not required for global variables.
Changes in a variable value	Any modification implied in a local variable does not affect the other functions of the program.	The changes applied in the global variable of a function reflect the changes in the whole program.

5.70) Compare automatic and external variable in C

5.73) Difference between register and automatic variable.

Feature	Automatic Variable	Register Variable	Static Variable	External Variable	
1	**Keyword Used**	auto	register	static	extern
2	**Storage**	Memory	CPU registers	Memory	Memory
3	**Default initial value**	Garbage Value	Garbage Value	Zero Value	Zero Value
4	**Scope**	Local to the block in which the variable is defined	Local to the block in which the variable is defined	Local to the block in which the variable is defined	Global
5	**Life**	Till the control remains within the block in which the variable is defined	Till the control remains within the block in which the variable is defined	Value of the variable persists between different function calls	As long as the program's execution doesn't come to an end

6	**Use**	General purpose use. Most widely used compared to other storage classes	Used extensively for loop counters	Used extensively for recursive functions	Used in case of variables which are being used by almost all the functions in a program

```
5.71)  C  program  to  convert  from  Celsius  to  Fahrenheit
int main()
{
    float celsius, fahrenheit;

    printf("Please Enter temperature in Celsius: \n");
    scanf("%f", &celsius);

    // Convert the temperature from celsius to fahrenheit
    fahrenheit = ((celsius * 9)/5) + 32;
    // fahrenheit = ((9/5) * celsius) + 32;
    // fahrenheit = ((1.8 * celsius) + 32;

    printf("\n %.2f Celsius = %.2f Fahrenheit", celsius, fahrenheit);

    return 0;
}
```

5.74) print a string on console

C program

```c
#include <stdio.h>

int main()
{
    char array[100];

    printf("Enter a string\n");
    scanf("%s", array);

    printf("Your string: %s\n", array);
    return 0;
}
```

Output:
Enter a string
We love C.
Your string: We

Only "We" is printed because function scanf can only be used to input strings without any spaces, to input strings containing spaces use gets function.

Input string containing spaces

```
#include <stdio.h>

int main()
{
  char z[100];

  printf("Enter a string\n");
  gets(z);

  printf("The string: %s\n", z);
  return 0;
}
```

5.75) read user input from keyboard

Make the changes, then compile and run the program to make sure it works. Note that scanf uses the same sort of format string as printf (type **man scanf** for more info). Also note the & in front of a and b. This is the **address operator** in C: It returns the address of the variable (this will not make sense until we discuss pointers). You must use the & operator in scanf on any variable of type char, int, or float, as well as structure types (which we will get to shortly). If you leave out the & operator, you will get an error when you run the program. Try it so that you can see what that sort of run-time error looks like.

Let's look at some variations to understand printf completely. Here is the simplest printf statement:

```
printf("Hello");
```

This call to printf has a format string that tells printf to send the word "Hello" to standard out. Contrast it with this:

```
printf("Hello\n");
```

The difference between the two is that the second version sends the word "Hello" followed by a carriage return to standard out.

The following line shows how to **output the value of a variable using printf**.

```
printf("%d", b);
```
The **%d** is a placeholder that will be replaced by the value of the variable **b** when the printf statement is executed. Often, you will want to embed the value within some other words. One way to accomplish that is like this:

```
printf("The temperature is ");
printf("%d", b);
printf(" degrees\n");
```
An easier way is to say this:

```
printf("The temperature is %d degrees\n", b);
```
You can also use multiple %d placeholders in one printf statement:

```
printf("%d + %d = %d\n", a, b, c);
```
In the printf statement, it is extremely important that the number of **operators** in the format string corresponds exactly with the number and type of the variables following it. For example, if the format string contains three %d operators, then it must be followed by exactly three parameters and they must have the same types in the same order as those specified by the operators.

You can **print all of the normal C types with printf** by using different placeholders:

- **int** (integer values) uses **%d**
- **float** (floating point values) uses **%f**
- **char** (single character values) uses **%c**
- **character strings** (arrays of characters, discussed later) use **%s**

6.76) format specifiers of printf in c

The format specifier is used during input and output. It is a way to tell the compiler what type of data is in a variable during taking input using scanf() or printing using printf(). Some examples are %c, %d, %f, etc.

The format specifier in printf() and scanf() are mostly the same but there is some difference which we will see.

printf(char *format, arg1, arg2, …)

This function prints the character on standard output and returns the number of character printed the format is a string starting with **%** and ends with conversion character (like c, i, f, d, etc.).
Between both, there can be elements governing the printing format. Below is its description

1. A minus(-) sign tells left alignment.
2. A number after **%** specifies the minimum field width to be printed if the characters are less than the size of width the remaining space is filled with space and if it is greater than it printed as it is without truncation.
3. A period(.) symbol seperate field width with the precision.

Precision tells the maximum number of digits in integer, characters in string and number of digits after decimal part in floating value.

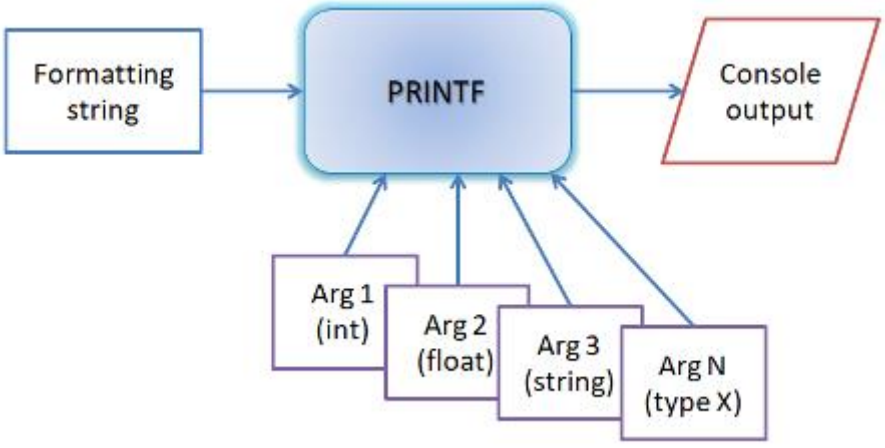
5.77) how scanf works in C

Name of printf and scanf

Print function in C abbreviated to printf() and scan function abbreviated to scanf. Printf and scanf are most frequently used in any programs written with C language. These functions implicitly work on console stdin/stdout files. There are other variations of these functions. fprintf/fscanf operates on file buffer and sscanf/sprintf can operates on string buffer. These functions are adapted to work on different I/O buffer but they all work on the same principle.

Printf working principle

printf or print function in C takes a formatting string and couple of optional variables as input and outputs strings to console while converting input variables to strings.

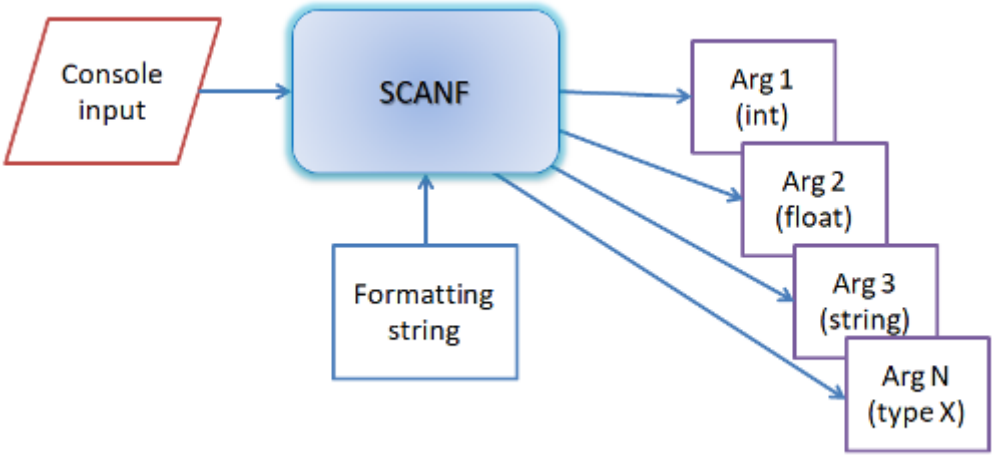


Printf and scanf takes multiple arguments and these functions are called variable length arguments function or vararg function. Take printf for consideration. User supply a string and input argumnets. Printf creates an internal buffer for constructing output string. Now printf iterates through each characters of user string and copies the character to the output string. Printf only stops at "%". "%" means there is an argument to convert. Arguments are in the form of char, int,

long, float, double or string. It converts it to string and appends to output buffer. If the argument is string then it does a string copy. Finally printf may reach at the end of user sting and it copies the entire buffer to the stdout file.

Scanf working principle

scanf or scan function in C takes a formatting string as input and couple of optional variables as output reference arguments. It converts the scanned string to variables and copies to the output variables.



Scanf is reverse process of printf. Scanf reads console input string. It iterates each characters of user provided string and stops at "%". Now scanf reads a line from stdin. User's input comes as a string. It converts string to char, int, long, float, double and sets the value of the pointer located at the argument. In care of string it simply copies the string to the output

The scanf function allows you to accept input from standard in, which for us is generally the keyboard. The scanf function can do a lot of different things, but can be unreliable because it doesn't handle human errors very well. But for simple programs it's good enough and easy to use.

The simplest application of **scanf** looks like this:

```
scanf("%d", &b);
```

The program will read in an integer value that the user enters on the keyboard (%d is for integers, as is printf, so b must be declared as an int) and place that value into b.

The scanf function uses the same placeholders as printf:

- **int** uses **%d**
- **float** uses **%f**
- **char** uses **%c**

- **character strings** (discussed later) use **%s**

You must put **&** in front of the variable used in scanf. The reason why will become clear once you learn about **pointers**. It is easy to forget the & sign, and when you forget it your program will almost always crash when you run it.

In general, it is best to use scanf as shown here -- to read a single value from the keyboard. Use multiple calls to scanf to read multiple values. In any real program, you will use the **gets** or **fgets** functions instead to read text a line at a time. Then you will "parse" the line to read its values. The reason that you do that is so you can detect errors in the input and handle them as you see fit.

6.93) Logical Operators

They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition under consideration. They are described below:

1. **Logical AND operator:** The **'&&'** operator returns true when both the conditions under consideration are satisfied. Otherwise it returns false. For example, **a && b** returns true when both a and b are true (i.e. non-zero).
2. **Logical OR operator:** The **'||'** operator returns true even if one (or both) of the conditions under consideration is satisfied. Otherwise it returns false. For example, **a || b** returns true if one of a or b or both are true (i.e. non-zero). Of course, it returns true when both a and b are true.
3. **Logical NOT operator:** The **'!'** operator returns true the condition in consideration is not satisfied. Otherwise it returns false. For example, **!a** returns true if a is false, i.e. when a=0.

6.98) Relational Operators

Relational operators are used for comparison of two values to understand the type of relationship a pair of number shares. For example, less than, greater than, equal to etc. Let's see them one by one

1. **Equal to operator:** Represented as **'=='**, the equal to operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise it returns false. For example, **5==5** will return true.
2. **Not equal to operator:** Represented as **'!='**, the not equal to operator checks whether the two given operands are equal or not. If not, it returns true. Otherwise it returns false. It is the exact boolean complement of the **'=='** operator. For example, **5!=5** will return false.

3. **Greater than operator:** Represented as **'>'**, the greater than operator checks whether the first operand is greater than the second operand or not. If so, it returns true. Otherwise it returns false. For example, **6>5** will return true.
4. Less than operator: Represented as **'<'**, the less than operator checks whether the first operand is lesser than the second operand. If so, it returns true. Otherwise it returns false. For example, **6<5** will return false.
5. **Greater than or equal to operator:** Represented as **'>='**, the greater than or equal to operator checks whether the first operand is greater than or equal to the second operand. If so, it returns true else it returns false. For example, **5>=5** will return true.
6. **Less than or equal to operator: Represented as '<='**, the less than or equal tooperator checks whether the first operand is less than or equal to the second operand. If so, it returns true else false. For example, **5<=5** will also return true.

6.94) Write a C program to swap between two numbers using bitwise operator.

```c
#include <stdio.h>

int main()
{
    int num1, num2;

    /* Input two numbers from user */
    printf("Enter any two numbers: ");
    scanf("%d%d", &num1, &num2);

    printf("Original value of num1 = %d\n", num1);
    printf("Original value of num2 = %d\n", num2);

    /* Swap two numbers */
    num1 ^= num2;
    num2 ^= num1;
    num1 ^= num2;

    printf("Num1 after swapping = %d\n", num1);
    printf("Num2 after swapping = %d\n", num2);

    return 0;
}
```

6.96) write a c program to find greatest no. between two numbers using ternary operator

```c
# include <stdio.h>

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

    printf("Enter three numbers : ") ;

```

```
9.     scanf("%d %d %d", &a, &b, &c) ;
10.
11.            big = a > b ? (a > c ? a : c) : (b > c ? b : c) ;
12.
13.            printf("\nThe biggest number is : %d", big) ;
14.        }
```

6.97) operator precedence in c

Operator	Operator/Description
Pre increment operator (++i)	value of i is incremented before assigning it to the variable i
Post increment operator (i++)	value of i is incremented after assigning it to the variable i
Pre decrement operator (–i)	value of i is decremented before assigning it to the variable i
Post decrement operator (i–)	value of i is decremented after assigning it to variable i

++ is really a pair of operators: pre-increment and post-increment. The former increments the value of a variable and returns the resulting value; the latter increments the value of the variable and returns the value prior to the increment. So, if a variable x initially contains the value 3, the following expressions yield different results:

```
y = ++x; // y==4, x==4
y = x++; // y==3, x==4
```

The effect on `x` is the same in both cases, but `y` gets a different value.

6.100) purpose of +=,<<=,>>=,&=,^= operators

+= , -=	Addition/subtraction assignment
*= , /=	Multiplication/division assignment
%= , &=	Modulus/bitwise AND assignment
^= , |=	Bitwise exclusive/inclusive OR assignment

<>=	Bitwise shift left/right assignment

Assignment operators are used to assigning value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of the variable on the left side otherwise the compiler will raise an error. Different types of assignment operators are shown below:

- **“=”**: This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.
  For example:

```
a = 10;
b = 20;
ch = 'y';
```

- **“+=”**: This operator is combination of ‘+’ and ‘=’ operators. This operator first adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.
  Example:

```
(a += b) can be written as (a = a + b)
```

  If initially value stored in a is 5. Then (a += 6) = 11.

- **“-=”**This operator is combination of ‘-‘ and ‘=’ operators. This operator first subtracts the current value of the variable on left from the value on the right and then assigns the result to the variable on the left.
  Example:

```
(a -= b) can be written as (a = a - b)
```

  If initially value stored in a is 8. Then (a -= 6) = 2.

- **“*=”**This operator is combination of ‘*’ and ‘=’ operators. This operator first multiplies the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.
  Example:

```
(a *= b) can be written as (a = a * b)
```

  If initially value stored in a is 5. Then (a *= 6) = 30.

- **“/=”**This operator is combination of ‘/’ and ‘=’ operators. This operator first divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.
  Example:

```
(a /= b) can be written as (a = a / b)
```

  If initially value stored in a is 6. Then (a /= 2) = 3.

6.101) odd and even using conditional operators

```
1. main()
2. {
3.    int n;
4.
```

```
    printf("Enter an integer\n");
    scanf("%d",&n);

    n%2 == 0 ? printf("Even number\n") : printf("Odd number\n");

            return 0;
        }

```

6.102) Error

6.105) Detect if two integers have opposite signs in c using xor operator

```
#include <stdio.h>
int main()
{
  int n1, n2; //declare two integers
  printf ("Enter two integer values: ");
  scanf ("%d%d", &n1, &n2);

  if ((n1 ^ n2) < 0)
  {
    printf ("Both numbers (%d, and %d) have opposite signs.\n", n1, n2);
  }
  else
  {
    printf ("Both numbers (%d, and %d) have same sign.\n", n1, n2);
  }

  return 0;
}
```

6.106) left and right shift operator

<< (left shift) Takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift. Or in other words left shifting an integer “x” with an integer “y” (x<<y) is equivalent to multiplying x with 2^y (2 raise to power y).

```
/* C++ Program to demonstrate use of left shift
   operator */
#include<stdio.h>
int main()
{
    // a = 5(00000101), b = 9(00001001)
    unsigned char a = 5, b = 9;

    // The result is 00001010
```

```
    printf("a<<1 = %d\n", a<<1);

    // The result is 00010010
    printf("b<<1 = %d\n", b<<1);
    return 0;
}
```

Output:

```
a<<1 = 10

b<<1 = 18
```

>> (right shift) Takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.Similarly right shifting (x>>y) is equivalent to dividing x with 2^y.

```
#include<stdio.h>
int main()
{
   int x = 19;
   printf ("x << 1 = %d\n", x << 1);
   printf ("x >> 1 = %d\n", x >> 1);
   return 0;
}
```

Output:

```
x << 1 = 38

x >> 1 = 9
```

6.107) What is the Difference Between Bitwise and Logical Operators?

Bitwise vs Logical Operators	
Bitwise operator is the type of operator provided by the programming language to perform computations.	Logical Operator is a type of operator provided by the programming language to perform logic-based operations.
Functionality	
Bitwise operators work on bits and perform bit by bit operations.	Logical operators are used to making a decision based on multiple conditions.
Themes	
Bitwise operators are &, |, ^, ~, <<, >>.	Logical operators are &&, ||, !

Evaluation	It evaluates both left and right side of the expression.	It only evaluates the left side of the expression.

Use	Use to check logical condition and also used to mask off certain bits such as parity bits.	Used only to check logical condition.

7.130) advantages of switch case and 7.132) difference between switch and nested if else

A switch statement is usually more efficient than a set of nested ifs. Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing.

1. **Check the Testing Expression:** An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.
2. **Switch better for Multi way branching:** When compiler compiles a switch statement, it will inspect each of the case constants and create a “jump table” that it will use for selecting the path of execution depending on the value of the expression. Therefore, if we need to select among a large group of values, a switch statement will run much faster than the equivalent logic coded using a sequence of if-elses. The compiler can do this because it knows that the case constants are all the same type and simply must be compared for equality with the switch expression, while in case of if expressions, the compiler has no such knowledge.
3. **if-else better for boolean values:** If-else conditional branches are great for variable conditions that result into a boolean, whereas switch statements are great for fixed data values.
4. **Speed:** A switch statement might prove to be faster than ifs provided number of cases are good. If there are only few cases, it might not effect the speed in any case. Prefer switch if the number of cases are more than 5 otherwise, you may use if-else too.
   If a switch contains more than five items, it’s implemented using a lookup table or a hash list. This means that all items get the same access time, compared to a list of if:s where the last item takes much more time to reach as it has to evaluate every previous condition first.
5. **Clarity in readability:** A switch looks much cleaner when you have to combine cases. Ifs are quite vulnerable to errors too. Missing an else statement can land you up in havoc. Adding/removing labels is also easier with a switch and makes your code significantly easier to change and maintain.

BASIS FOR COMPARISON	IF-ELSE	SWITCH
Basic	Which statement will be executed depend upon the output of the expression inside if statement.	Which statement will be executed is decided by user.

BASIS FOR COMPARISON	IF-ELSE	SWITCH
Expression	if-else statement uses multiple statement for multiple choices.	switch statement uses single expression for multiple choices.
Testing	if-else statement test for equality as well as for logical expression.	switch statement test only for equality.
Evaluation	if statement evaluates integer, character, pointer or floating-point type or boolean type.	switch statement evaluates only character or integer value.
Sequence of Execution	Either if statement will be executed or else statement is executed.	switch statement execute one case after another till a break statement is appeared or the end of switch statement is reached.
Default Execution	If the condition inside if statements is false, then by default the else statement is executed if created.	If the condition inside switch statements does not match with any of cases, for that instance the default statements is executed if created.
Editing	It is difficult to edit the if-else statement, if the nested if-else statement is used.	It is easy to edit switch cases as, they are recognized easily.

Definition of if-else

The if-else statements belong to selection statements in OOP. The general form of the if-else statements is as follow

```
1. if (expression){
2. statement(s)
3. }else{
4. statement(s)
5. }
```

where "if" and "else" are the keywords, and the statements can be a single statement or a block of statements. The expression evaluates to be "true" for any non-zero value and for zero it evaluates to be "false".
The expression in if statement can contain an integer, character, pointer, floating-point or it can be a boolean type. The else statement is optional in an if-else statement. If the expression returns true, the statements inside if statement is executed, and if it returns false the statements inside else statement are executed and, in case an else statement is a not created no action is performed, and the control of the program jump out of an if-else statement.

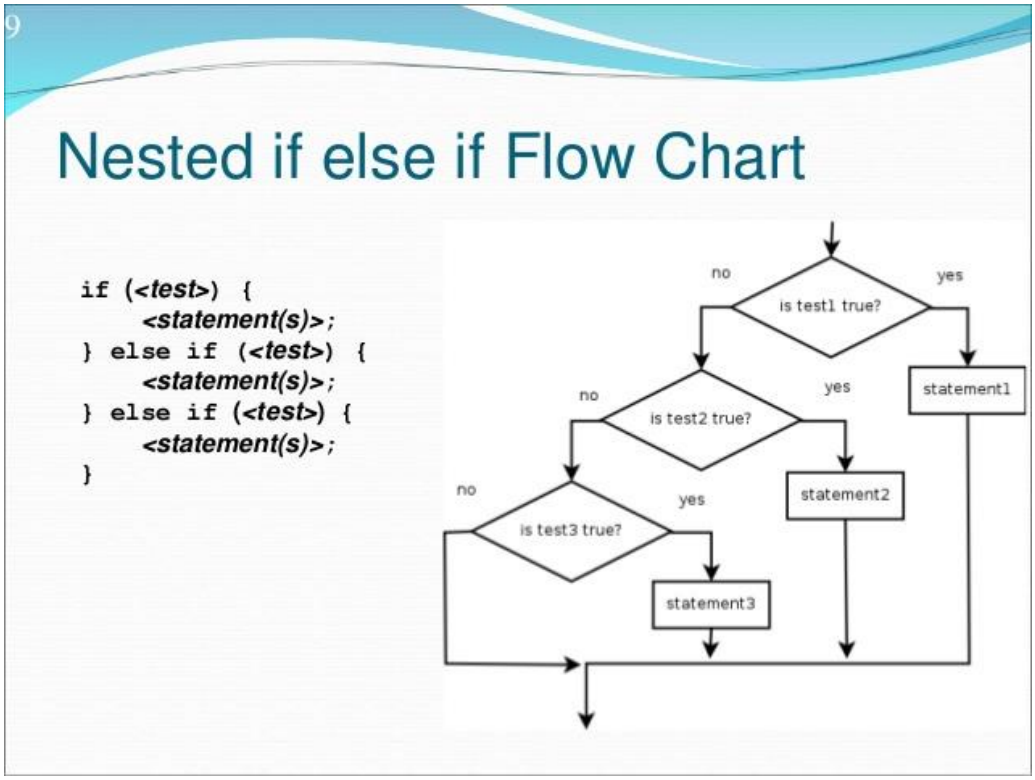
```
1. int i=45, j=34;
2. if (i==45 & j==34){
3. cout<< " i ="<<i;
```

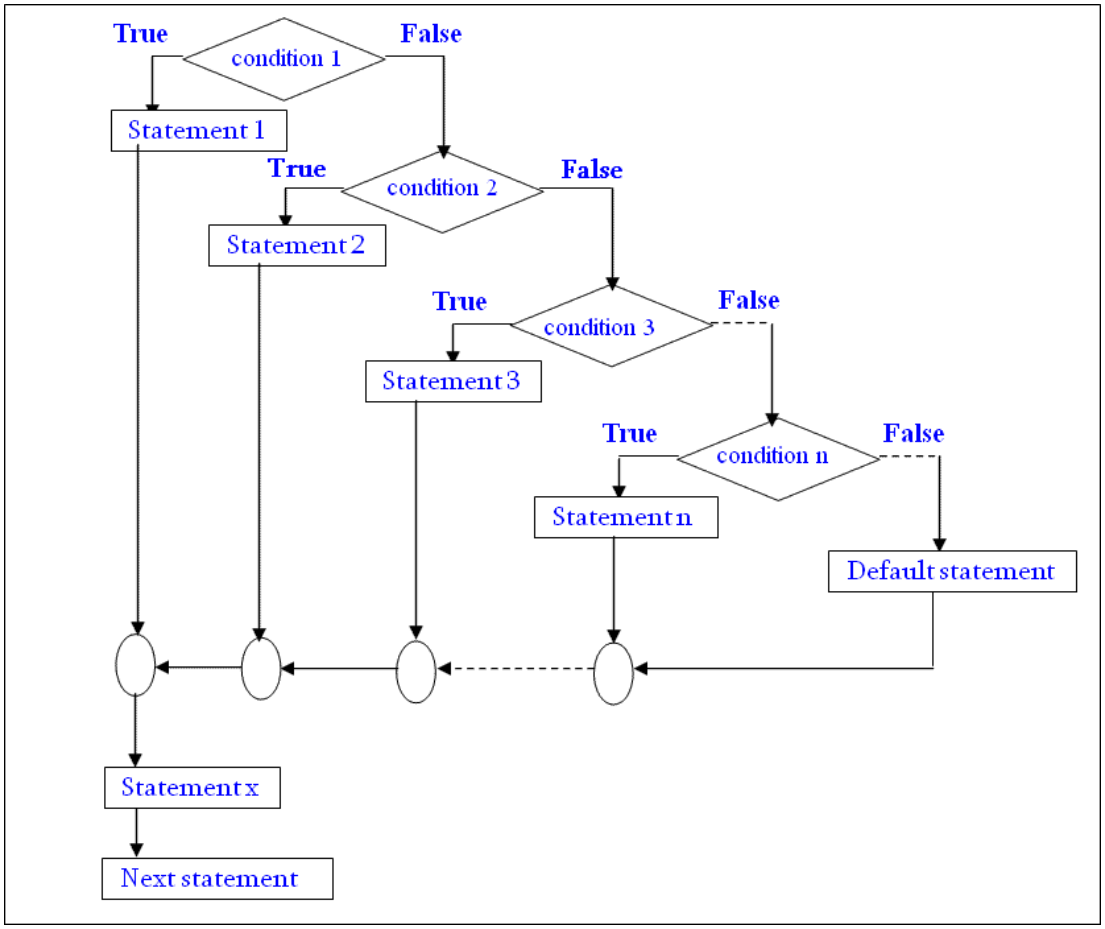
```
}else{
cout<< " j ="<<j;
}

//output
i=45
```

switch

```
switch( expression ){
case constant1:
statement(s);
break;
case constant2:
statement(s);
break;
case constant3:
statement(s);
break;
.
.
default
statement(s);
}
```

7.134) nested if else and if else ladder flowchart





7. 136) nested switch statement in c

Nested Switch Statements occurs when a switch statement is defined inside another switch statement. The first switch is referred to as an outer switch statement whereas the inside switch is referred to as an inner switch statement.

In this tutorial, we will learn about the syntax of a nested switch statement in C programming. In addition, we shall also write a real-life example to demonstrate the concept of nested switch statements.

What is a nested switch statement?

A nested switch statement is defined as having a switch statement within another switch statement

The syntax for Nested Switch Case:

The basic syntax used for creating a nested switch statement is as follows:

```
switch (a)
```

```
{
  printf("This a is part of outer switch" );
  case 1: // code to be executed if a = 1;
    break;
  case 2:
    switch(b)
    {
      case 1:
        // code to be executed if b = 1;
        printf("This b is part of inner switch" );
        break;
      case 2:
        // code to be executed if b = 2;
        printf("This b is part of inner switch" );
        break;
    }
    break;
    default:
      // code to be executed if
      // a doesn't match any cases
}
```

In the above syntax, we have declared two nested switch statements.

The first switch statement with variable “a” is termed as the **outer switch statement.** Whereas, the switch statement with the variable “b” is termed as the **inner switch statement**.

7.131) multi way branching in c

The C language programs follows a sequential form of execution of statements. Many times it is required to alter the flow of sequence of instructions. C language provides statements that can alter the flow of a sequence of instructions. These statements are called as control statements. To jump from one part of the program to another,these statements help. The control transfer may be unconditional or conditional. Branching Statemnt are of following categories:

1. **If Statement**
2. **The If else Statement**
3. **Compound Relational tests**
4. **Nested if Statement**
5. **Switch Statement**

If Statement

If statement is the simplest form of the control statement. It is very frequently used in allowing the flow of program execution and decision making.

The If structure has the following syntax

```
if(condition)

statement;
```

The command says that if the condition is true then perform the following statement or If the condition is fake the computer skips the statement and moves on to the next instruction in the program

---

The following program calculate the absolute value of an integer using if statement:

```
Calculate the absolute value of an integer */
# include < stdio.h > //Include the stdio.h file
void main ( ) // start of the program


{
int numbers; // Declare the variables
printf (“Type a number:”); // message to the user
```

```
scanf (“%d”, & number); // read the number from standard input
if (number < 0) // check whether the number is a negative
number
number = - number; // If it is negative then convert it into
positive.
Printf (“The absolute value is % d \n”, number); // print the value
}
```

The If else Statement

The if else is actually just an extension of the general format of if statement. If the result of the condition is true, then program statement 1 is executed else program statement 2 will be executed. The syntax of the If else statement is as follows:

```
If (condition)

Program statement 1;
Else
Program statement 2;
```

The following program find whether a number is negative or positive using if statement:

```
#include < stdio.h > //include the stdio.h header file in your program
void main ( ) // Start of the main
{
int num; // declare variable num as integer
printf (“Enter the number”); //message to the user
scanf (“%d”, &num); // read the input number from keyboard
if (num < 0) // check whether number is less than zero.
Printf (“The number is negative”) // If it is less than zero then it is negative.
Else // else statement.
Printf (“The number is positive”); //If it is more than zero then the given
number is positive.
}
```

Compound Relational tests

To perform compound relational tests,C language provides the necessary mechanisms. A compound relational test is simple one or more simple relational tests joined together by either the the logical OR operators or logical AND. These operators are represented by character pairs && // respectively. To

form complex expressions in C,the compound operators can be used. The syntax of the Compound Relational tests is as follows:

```
a> if (condition1 && condition2 && condition3)
b>if (condition1 // condition2 // condition3)
```

Nested if Statement

The if statement may itself contain another if statement is called as nested if statement. The syntax of the Nested if Statement is as follows

```
if (condition1)
if (condition2)
statement-1;
else
statement-2;
else
statement-3;
```

The following example print the given numbers along with the largest number using nested if statement.

```
#include < stdio.h > //includes the stdio.h file to your program
main ( ) //start of main function
{
int a,b,c,big; //declaration of variables
printf (“Enter three numbers”); //message to the user
scanf (“%d %d %d”, &a, &b, &c); //Read variables a,b,c,
if (a>b) // check whether a is greater than b if true then
if(a>c) // check whether a is greater than c
big = a ; // assign a to big
else big = c ; // assign c to big
else if (b>c) // if the condition (a>b) fails check whether b is
greater than c
big = b ; // assign b to big
else big = c ; // assign C to big
printf (“Largest of %d,%d&%d = %d”, a,b,c,big);
}
//print the given numbers along with the largest number.
```

Switch Statement

The **switch-case** statement is a multi-way decision making statement. Unlike the multiple decision statement that can be created using if-else, the **switch** statement evaluates the conditional expression and tests it against the numerous constant values.During execution,the branch corresponding to the value that the expression matches is taken.

The value of the expressions in a switch-case statement must have to be an ordinal type i.e. integer, char, short, long, etc.Double and Float are not allowed.

The syntax of switch statement is as follows :

```
switch( expression )
{
case constant-expression1: statements1;
[case constant-expression2: statements2;]
[case constant-expression3: statements3;]
[default : statements4;]
}
```

output:

```
#include
main()
{
int n=7;

switch(n) {
case 0:
printf("You typed zero.\n");
break;
case 3:
case 5:
case 7:
printf("n is a prime number\n");
break;
case 2: printf("n is a prime number\n");
case 4:
case 6:
case 8:
printf("n is an even number\n");
break;
case 1:
case 9:
printf("n is a perfect square\n");
break;
default:
printf("Only single-digit numbers are allowed\n");
break;
}
}
```

7.142) how if statement functions. Check from above.

7.139) why decision making is important in programming construct.

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C language handles decision-making by supporting the following statements, then write the points as above, if, else if, switch, nested.

7.138) properties of If statement. Check from above.

7.154) break and continue difference

BASIS FOR COMPARISON	BREAK	CONTINUE
Task	It terminates the execution of remaining iteration of the loop.	It terminates only the current iteration of the loop.
Control after break/continue	'break' resumes the control of the program to the end of loop enclosing that 'break'.	'continue' resumes the control of the program to the next iteration of that loop enclosing 'continue'.
Causes	It causes early termination of loop.	It causes early execution of the next iteration.
Continuation	'break' stops the continuation of loop.	'continue' do not stops the continuation of loop, it only stops the current iteration.
Other uses	'break' can be used with 'switch', 'label'.	'continue' can not be executed with 'switch' and 'labels'.

7.158) Difference between while,dowhile,for loops

Types of Loops

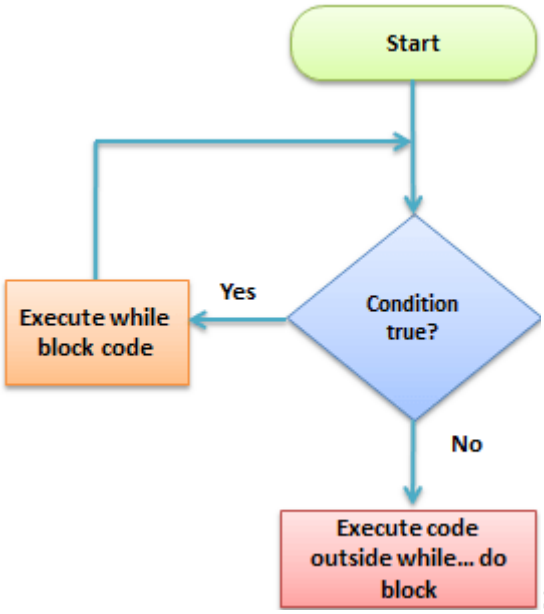
Depending upon the position of a control statement in a program, a loop is classified into two types:

1. Entry controlled loop

2. Exit controlled loop

In an **entry controlled loop,** a condition is checked before executing the body of a loop. It is also called as a pre-checking loop.

In an **exit controlled loop**, a condition is checked after executing the body of a loop. It is also called as a post-checking loop.



Sample Loop

The control conditions must be well defined and specified otherwise the loop will execute an infinite number of times. The loop that does not stop executing and processes the statements number of times is called as an **infinite loop**. An infinite loop is also called as an "**Endless loop**." Following are some characteristics of an infinite loop:

1. No termination condition is specified.

2. The specified conditions never meet.

The specified condition determines whether to execute the loop body or not.

'C' programming language provides us with three types of loop constructs:

1. The while loop

2. The do-while loop

3. The for loop

While Loop

A while loop is the most straightforward looping structure. The basic format of while loop is as follows:

```
while (condition) {
            statements;
}
```

It is an entry-controlled loop. In while loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only then the body of a loop is executed. After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false. Once the condition becomes false, the control goes out of the loop.

After exiting the loop, the control goes to the statements which are immediately after the loop. The body of a loop can contain more than one statement. If it contains only one statement, then the curly braces are not compulsory. It is a good practice though to use the curly braces even we have a single statement in the body.

In while loop, if the condition is not true, then the body of a loop will not be executed, not even once. It is different in do while loop which we will see shortly.

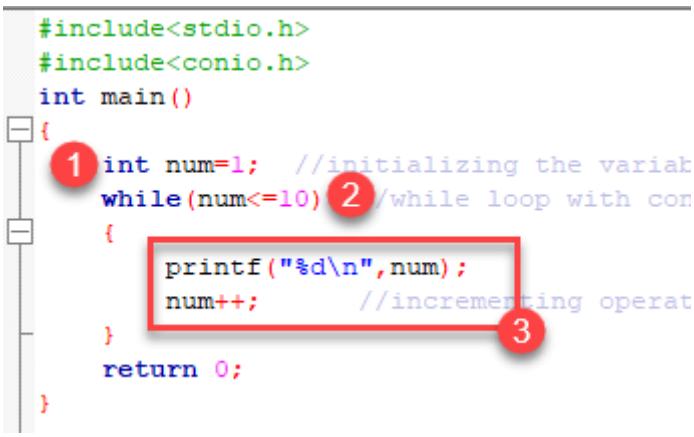
Following program illustrates a while loop:

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int num=1;       //initializing the variable
        while(num<=10)   //while loop with condition
        {
                printf("%d\n",num);
                num++;           //incrementing operation
        }
        return 0;
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

The above program illustrates the use of while loop. In the above program, we have printed series of numbers from 1 to 10 using a while loop.



1. We have initialized a variable called num with value 1. We are going to print from 1 to 10 hence the variable is initialized with value 1. If you want to print from 0, then assign the value 0 during initialization.
2. In a while loop, we have provided a condition (num<=10), which means the loop will execute the body until the value of num becomes 10. After that, the loop will be terminated, and control will fall outside the loop.
3. In the body of a loop, we have a print function to print our number and an increment operation to increment the value per execution of a loop. An initial value of num is 1, after the execution, it will become 2, and during the next execution, it will become 3. This process will continue until the value becomes 10 and then it will print the series on console and terminate the loop.

\n is used for formatting purposes which means the value will be printed on a new line.

Do-While loop

A do-while loop is similar to the while loop except that the condition is always executed after the body of a loop. It is also called an exit-controlled loop.

The basic format of while loop is as follows:

```
do {
  statements
} while (expression);
```

As we saw in a while loop, the body is executed if and only if the condition is true. In some cases, we have to execute a body of the loop at least once even if the condition is false. This type of operation can be achieved by using a do-while loop.

In the do-while loop, the body of a loop is always executed at least once. After the body is executed, then it checks the condition. If the condition is true, then it will again execute the body of a loop otherwise control is transferred out of the loop.

Similar to the while loop, once the control goes out of the loop the statements which are immediately after the loop is executed.

The critical difference between the while and do-while loop is that in while loop the while is written at the beginning. In do-while loop, the while condition is written at the end and terminates with a semi-colon (;)

The following program illustrates the working of a do-while loop:

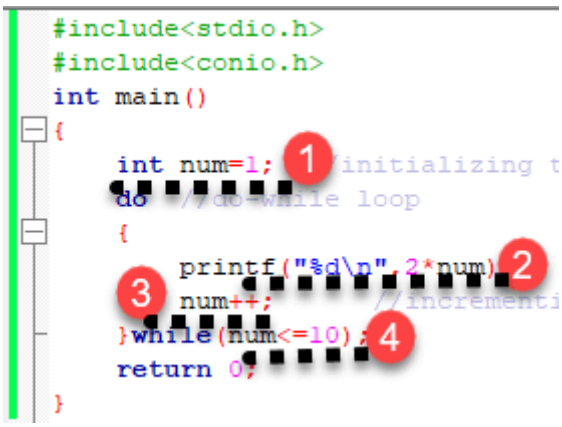
We are going to print a table of number 2 using do while loop.

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int num=1;       //initializing the variable
        do       //do-while loop
        {
                printf("%d\n",2*num);
                num++;           //incrementing operation
        }while(num<=10);
        return 0;
}
```

Output:

```
2
4
6
8
10
12
14
16
18
20
```

In the above example, we have printed multiplication table of 2 using a do-while loop. Let's see how the program was able to print the series.



1. First, we have initialized a variable 'num' with value 1. Then we have written a do-while loop.
2. In a loop, we have a print function that will print the series by multiplying the value of num with 2.
3. After each increment, the value of num will increase by 1, and it will be printed on the screen.
4. Initially, the value of num is 1. In a body of a loop, the print function will be executed in this way: 2*num where num=1, then 2*1=2 hence the value two will be printed. This will go on until the value of num becomes 10. After that loop will be terminated and a statement which is immediately after the loop will be executed. In this case return 0.

For loop

A for loop is a more efficient loop structure in 'C' programming. The general structure of for loop is as follows:

```
for (initial value; condition; incrementation or decrementation )
{
  statements;
}
```

- The initial value of the for loop is performed only once.
- The condition is a Boolean expression that tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.
- The incrementation/decrementation increases (or decreases) the counter by a set value.

Following program illustrates the use of a simple for loop:

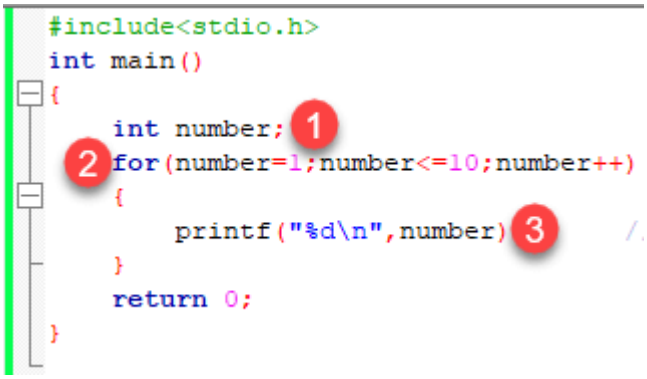
```
#include<stdio.h>
int main()
{
```

```
        int number;
        for(number=1;number<=10;number++)//for loop to print 1-10 numbers
        {
                printf("%d\n",number);           //to print the number
        }
        return 0;
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

The above program prints the number series from 1-10 using for loop.



1. We have declared a variable of an int data type to store values.
2. In for loop, in the initialization part, we have assigned value 1 to the variable number. In the condition part, we have specified our condition and then the increment part.
3. In the body of a loop, we have a print function to print the numbers on a new line in the console. We have the value one stored in number, after the first iteration the value will be incremented, and it will become 2. Now the variable number has the value 2. The condition will be rechecked and since the condition is true loop will be executed, and it will print two on the screen. This loop will keep on executing until the value of the variable becomes 10. After that, the loop will be terminated, and a series of 1-10 will be printed on the screen.

In C, the for loop can have multiple expressions separated by commas in each part.

For example:

```
for (x = 0, y = num; x < y; i++, y--) {
  statements;
}
```

Also, we can skip the initial value expression, condition and/or increment by adding a semicolon.

For example:

```
int i=0;
int max = 10;
for (; i < max; i++) {
  printf("%d\n", i);
}
```

Notice that loops can also be nested where there is an outer loop and an inner loop. For each iteration of the outer loop, the inner loop repeats its entire cycle.

Consider the following example, that uses nested for loops output a multiplication table:

```
#include <stdio.h>
int main() {
int i, j;
int table = 2;
int max = 5;
for (i = 1; i <= table; i++) { // outer loop
  for (j = 0; j <= max; j++) { // inner loop
    printf("%d x %d = %d\n", i, j, i*j);
  }
  printf("\n"); /* blank line between tables */
}}
```

Output:

```
1 x 0 = 0
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
```

```
2 x 0 = 0
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
```

The nesting of for loops can be done up-to any level. The nested loops should be adequately indented to make code readable. In some versions of 'C,' the nesting is limited up to 15 loops, but some provide more.

The nested loops are mostly used in array applications which we will see in further tutorials.

Break Statement

The break statement is used mainly in in the switch statement. It is also useful for immediately stopping a loop.

The break statement is used to exit an iteration or switch statement. It transfers control to the statement immediately following the iteration substatement or switch statement.

The break statement terminates only the most tightly enclosing loop or switch statement. In loops, break is used to terminate before the termination criteria evaluate to 0. In the switch statement, break is used to terminate sections of code — normally before a case label. The following example illustrates the use of the break statement in a for loop:

We consider the following program which introduces a break to exit a while loop:

```
#include <stdio.h>
int main() {
int num = 5;
while (num > 0) {
  if (num == 3)
    break;
  printf("%d\n", num);
  num--;
}}
```

Output:

```
5
4
```

Continue Statement

When you want to skip to the next iteration but remain in the loop, you should use the continue statement.

For example:

```
#include <stdio.h>
int main() {
int nb = 7;
while (nb > 0) {
  nb--;
  if (nb == 5)
    continue;
 printf("%d\n", nb);
}}
```

Output:

```
6
4
3
2
1
0
```

So, the value 5 is skipped.

break	continue
A `break` can appear in both `switch` and loop (`for`, `while`, `do`) statements.	A `continue` can appear only in loop (`for`, `while`, `do`) statements.
A `break` causes the `switch` or loop statements to terminate the moment it is executed. Loop or `switch` ends abruptly when break is encountered.	A `continue` doesn't terminate the loop, it causes the loop to go to the next iteration. All iterations of the loop are executed even if `continue` is encountered. The `continue` statement is used to skip statements in the loop that appear after the `continue`.
The `break` statement can be used in both `switch` and loop statements.	The `continue` statement can appear only in loops. You will get an error if this appears in switch statement.
When a `break` statement is encountered, it terminates the block and gets the control out of the `switch` or loop.	When a `continue` statement is encountered, it gets the control to the next iteration of the loop.

A `break` causes the innermost enclosing loop or `switch` to be exited immediately.	A `continue` inside a loop nested within a `switch` causes the next loop iteration.

7.137) write in brief about the use of break keyword in switch. Check above.

7.158) Differentiate between while,dowhile and for loops.

BASIS FOR COMPARISON	FOR	WHILE

Declaration	for(initialization; condition; iteration){
	//body of 'for' loop
	}
		while (condition) {
		statements; //body of loop
		}

Format	Initialization, condition checking, iteration statement are written at the top of the loop.	Only initialization and condition checking is done at the top of the loop.
Use	The 'for' loop used only when we already knew the number of iterations.	The 'while' loop used only when the number of iteration are not exactly known.
Condition	If the condition is not put up in 'for' loop, then loop iterates infinite times.	If the condition is not put up in 'while' loop, it provides compilation error.
Initialization	In 'for' loop the initialization once done is never repeated.	In while loop if initialization is done during condition checking, then initialization is done each time the loop iterate.
Iteration statement	In 'for' loop iteration statement is written at top, hence, executes only after all statements in loop are executed.	In 'while' loop, the iteration statement can be written anywhere in the loop.

BASIS FOR COMPARISON	WHILE	DO-WHILE

General Form	while (condition) {
	statements; //body of loop
	}
		do{
		.
		statements; // body of loop.
		.
		} while(Condition);

Controlling Condition	In 'while' loop the controlling condition appears at the start of the loop.	In 'do-while' loop the controlling condition appears at the end of the loop.

BASIS FOR COMPARISON	WHILE	DO-WHILE
Iterations	The iterations do not occur if, the condition at the first iteration, appears false.	The iteration occurs at least once even if the condition is false at the first iteration.
Alternate name	Entry-controlled loop	Exit-controlled loop
Semi-colon	Not used	Used at the end of the loop

8.186) Define array. Explain types of array in details.

Arrays:- When there is a need to use many variables then There is a big problem because we will Conflict with name of variables So that in this Situation where we wants to Operate on many numbers then we can use array . The Number of Variables also increases the complexity of the Program. So that we uses Arrays.

An array cannot have a mixture of different data types as its elements. Also, array elements cannot be functions; however, they may be pointers to functions. In computer memory, array elements are stored in a sequence of adjacent memory blocks. Since all the elements of an array are of same data type, the memory blocks allocated to elements of an array are also of same size. Each element of an array occupies one block of memory. The size of memory blocks allocated depends on the data type and it is same as for different data types.

Arrays are Set of Elements having same data type or we can Say that Arrays are Collection of Elements having same name and same data type But Always Remember Arrays are Always Start From its index value and the index of array is start From 0 to n-1.

Suppose we wants to Access 5th Element of array then we will use 4th Element Because Arrays are Start From 0 and arrays are always stored in Continuous Memory Locations The Number of Elements and Types of array are Identified by Subscript of array Elements. The Various types of Array those are provided by c as Follows:-

1. Single Dimensional Array
2. Two Dimensional Array
3. Three Dimensional array
4. Character Array or Strings.

A dimensional is used representing the elements of the array for example

int a[5]

The [] is used for dimensional or the sub-script of the array that is generally used for declaring the elements of the array For Accessing the Element from the array we can use the Subscript of the Array like this

a[3]=100

This will set the value of 4^{th} element of array

So there is only the single bracket then it called the Single Dimensional Array

This is also called as the Single Dimensional Array

2) Two Dimensional Array or the Matrix

The Two Dimensional array is used for representing the elements of the array in the form of the rows and columns and these are used for representing the Matrix A Two Dimensional Array uses the two subscripts for declaring the elements of the Array

Like this int a[3][3]

So This is the Example of the Two Dimensional Array In this first 3 represents the total number of Rows and the Second Elements Represents the Total number of Columns The Total Number of elements are judge by Multiplying the Numbers of Rows * Number of Columns in The Array in the above array the Total Number of elements are 9

3) Multidimensional or the Three Dimensional Array :The Multidimensional Array are used for Representing the Total Number of Tables of Matrix A Three dimensional Array is used when we wants to make the two or more tables of the Matrix Elements for Declaring the Array Elements we can use the way like this

int a[3][3][3]

In this first 3 represents the total number of Tables and the second 3 represents the total number of rows in the each table and the third 3 represents the total number of Columns in the Tables

So this makes the 3 Tables having the three rows and the three columns

The Main and very important thing about the array that the elements are stored always in the Contiguous in the memory of the Computer

4) Character Array of String: -Like an integer characters are also be in the Array The Array of Characters are called as the Strings They are Generally used for representing the Strings Always Remember that a String is Terminated with the \0 or Null Character

There are the built in string Operation those are provided by the C Language in the String.h Header file Like

1) strLen For Getting the Length or Total Numbers of Characters in String

2) strconcat This is Used for Joining the two Strings or This function is used for Concatenating the two Strings.

3) strRev This Function is used for obtaining the Reverse of the String

4) strcmp This Function is used for Comparing the Two Strings and it gives us the Result as follows after comparing the Two Strings

it Returns us + value

if String1 is Greater than String2

it Returns us the - Value

if String1 is Less than String2

it Returns us the 0

if string1 is Equals to String2

Like The array elements of Integer Types The Character array also are the Single Dimensional or The Two Dimensional Array

Single Dimensional Array The Single Dimensional array are used for creating the Number of characters like

char name[10]

in this we can use the 10 characters on the name variable Means we can give the name as 10 characters long

2) Two Dimensional array :-When we talk about the Two Dimensional of the Character array The first Subscript of the array if used for representing the Total Numbers of Strings and the second Subscript is used for defining the length of the array Characters in the String

like This

char name[5][10]

It declares the 5 names and each having the characters up to 10 So the First Subscript is used for defining the total number of the Strings and the Second is used for Length of the Characters

8. 190) Write in brief about a 2d array declaration and initialization. Check from above.

8.191) Discuss about 1D array declaration and initialization. Check from above.

8.189) advantages and disadvantages of array

Advantages

It is better and convenient way of storing the data of same datatype with same size.
It allows us to store known number of elements in it.
It allocates memory in contiguous memory locations for its elements. It does not allocate any extra space/ memory for its elements. Hence there is no memory overflow or shortage of memory in arrays.
Iterating the arrays using their index is faster compared to any other methods like linked list etc.
It allows to store the elements in any dimensional array - supports multidimensional array.

Disadvantages

It allows us to enter only fixed number of elements into it. We cannot alter the size of the array once array is declared. Hence if we need to insert more number of records than declared then it is not possible. We should know array size at the compile time itself.
Inserting and deleting the records from the array would be costly since we add / delete the elements from the array, we need to manage memory space too.
It does not verify the indexes while compiling the array. In case there is any indexes pointed which is more than the dimension specified, then we will get run time errors rather than identifying them at compile time.
Allocating more memory than the requirement leads to wastage of memory and less allocation of memory also leads to a problem.

8.214) call by value and call by reference difference

What is Call by Value method?

Call by value method copies the value of an argument into the formal parameter of that function. Therefore, changes made to the parameter of the main function do not affect the argument.

In this parameter passing method, values of actual parameters are copied to function's formal parameters, and the parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of the caller.

What is Call by Reference method?

Call by reference method copies the address of an argument into the formal parameter. In this method, the address is used to access the actual argument used in the function call. It means that changes made in the parameter alter the passing argument.

In this method, the memory allocation is the same as the actual parameters. All the operation in the function are performed on the value stored at the address of the actual parameter, and the modified value will be stored at the same address.

Example of a Call by Value method

```
void main() {
        int a = 10,
        void increment(int);
        Cout << "before function calling" << a;
        increment(a);
        Cout << "after function calling" << a;
        getch();

        void increment(int x) {
            int x = x + 1;
            Cout << "value is" << x;
        }
```

Output:

```
before function calling 10
value is 11
after function calling 1-0
```

Because variable declared 'a'in main() is different from variable 'x' in increment(). In this programme only variable names are similar, but their memory address are different and stored in different memory locations.

Example of a Call by Reference method

```
Public static void(string args[]) {
    int a = 10;
    System.out.println("Before call Value of a = ", a);
    Void increment();
    System.out.println("After call Value of a = ", a);
}

Void increment(int x) {
    int x = x + 1;
}
```

Output:

```
Before call Value of a =10
```

```
After call Value of a =11
```

Because variable declared 'a' in is referencing/ pointing to variable 'a' in main(). Here variable name is different, but both are pointing/referencing to same memory address locations.

Call by Value vs. Call by Reference

Parameters	Call by value	Call by reference
Definition	While calling a function, when you pass values by copying variables, it is known as "Call By Values."	While calling a function, in programming language instead of copying the values of variables, the address of the variables is used it is known as "Call By References.
Arguments	In this method, a copy of the variable is passed.	In this method, a variable itself is passed.
Effect	Changes made in a copy of variable never modify the value of variable outside the function.	Change in the variable also affects the value of the variable outside the function.
Alteration of value	Does not allow you to make any changes in the actual variables.	Allows you to make changes in the values of variables by using function calls.
Passing of variable	Values of variables are passed using a straightforward method.	Pointer variables are required to store the address of variables.
Value modification	Original value not modified.	The original value is modified.
Memory Location	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in the same memory location
Safety	Actual arguments remain safe as they cannot be modified accidentally.	Actual arguments are not Safe. They can be accidentally modified, so you need to handle arguments operations carefully.
Default	Default in many programming languages like C++.PHP. Visual Basic NET, and C#.	It is supported by most programming languages like JAVA, but not as default.

8.215) advantages of using functions

C programming makes use of *modularity* to remove the complexity of a program. The programmer divides the program into different *modules* or ***functions*** and accesses certain functions when needed. Every C program has at least one function. The most common function that we use in our day-to-day programming is the *main()* function. The compiler always executes the *main()* function first and then any other function(if it is called from the main method).

A **function** is basically a block of statements that performs a particular task. Suppose a task needs to be performed continuously on many data at different points of time, like one at the beginning of the program and one at the end of the program, so instead of writing the same piece of code twice, a person can simply write it in a function and call it twice. And after the execution of any function block, the control always comes back to the *main()* function.

ADVANTAGES OF USING A FUNCTION

Here are several advantages of using functions in your code:

- Use of functions enhances the *readability* of a program. A big code is always difficult to read. Breaking the code in smaller Functions keeps the program organized, easy to understand and makes it reusable.
- The C compiler follows *top-to-down* execution, so the control flow can be *easily managed* in case of functions. The control will always come back to the *main()* function.
- It reduces the complexity of a program and gives it a *modular structure*.
- In case we need to test only a particular part of the program we will have to run the whole program and figure out the errors which can be quite a complex process. Another advantage here is that functions can be *individually tested* which is more convenient than the above mentioned process.
- A function can be used to create our own *header file* which can be used in any number of programs i.e. the reusability.

With so many advantages, functions are a boon for any programmer. Let's learn more about these functions:

TYPES OF FUNCTIONS

Apart from the functions that programmers create according to their requirement, C compilers has some built-in functions that can be used anytime by the programmer. C Programming Language has two types of functions:

- ***Built-in Functions/library Functions***
- ***User-defined Functions***

LIBRARY FUNCTIONS

There functions are already defined in the C compilers. They are used for String handling, I/O operations, etc. These functions are defined in the **header file**. To use these functions we need to import the specific header files.

Eg:

The library function **<stdio.h>** includes these common functions(there are many other functions too):

- **printf()** shows the output in the user's format.
- **scanf()** used to take the user's input which can be a character, numeric value, string,etc.
- **getchar()** takes character input from the user.
- **gets()** reads a line.

The library function **<math.h>** includes these common functions(there are many other functions too):

- **pow()** finds the power of the given number.
- **sin()** finds the sine of the given number.
- **exp()** finds the exponent of the given number.
- **cos()** finds the cosine of the given number.
- **sqrt()** finds the square root of the number.

- **log()** finds the logarithmic value of the number.

Apart from **<stdio.h>** and **<math.h>** there are many other header files that contain library functions such as **<conio.h>** that contains **clrscr()** and **getch()**.

USER-DEFINED FUNCTIONS

User-defined functions are the ones created by the user. The user can program it to perform any desired function.It is like customizing the functions that we need in a program. A program can have more than one user-defined functions. All the user-defined functions need to be called(directly or indirectly) inside the *main()* function in order to be executed. User-defined functions can be added to the program in two ways. Either through **user-defined header files** or by adding a function block directly to the program. But the most important thing is to have a *main()* function. It makes it easier to code and call in other functions in its body.

Any function has 4 building blocks to be declared –

- Function name
- Function Parameters
- Return type
- Statements to be executed

8.216) Difference between library and user defined functions

The difference between the library and user-defined functions is that we do not need to write a code for a library function. It is already present inside the header file which we always include at the beginning of a program. You just have to type the name of a function and use it along with the proper syntax. Printf, scanf are the examples of a library function.

Whereas, a user-defined function is a type of function in which we have to write a body of a function and call the function whenever we require the function to perform some operation in our program.

A user-defined function is always written by the user, but later it can be a part of 'C' library. It is a major advantage of 'C' programming.

8.218,219) what are local and global variables give example. Refer to 5.69

```
public int add(){
int a =4;
int b=5;
return a+b;
}
```

Here, 'a' and 'b' are local variables

```
int a =4;
int b=5;
public int add(){
return a+b;
```

```
}
```

Here, 'a' and 'b' are global variables.

8.217) difference between recursion and iteration.

BASIS FOR COMPARISON	RECURSION	ITERATION
Basic	The statement in a body of function calls the function itself.	Allows the set of instructions to be repeatedly executed.
Format	In recursive function, only termination condition (base case) is specified.	Iteration includes initialization, condition, execution of statement within loop and update (increments and decrements) the control variable.
Termination	A conditional statement is included in the body of the function to force the function to return without recursion call being executed.	The iteration statement is repeatedly executed until a certain condition is reached.
Condition	If the function does not converge to some condition called (base case), it leads to infinite recursion.	If the control condition in the iteration statement never become false, it leads to infinite iteration.
Infinite Repetition	Infinite recursion can crash the system.	Infinite loop uses CPU cycles repeatedly.
Applied	Recursion is always applied to functions.	Iteration is applied to iteration statements or "loops".
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not uses stack.
Overhead	Recursion possesses the overhead of repeated function calls.	No overhead of repeated function call.
Speed	Slow in execution.	Fast in execution.
Size of Code	Recursion reduces the size of the code.	Iteration makes the code longer.

Definition of Recursion

C++ allows a function to call itself within its code. That means the definition of the function possesses a function call to itself. Sometimes it is also called "**circular definition**". The set of local variables and parameters used by the function are newly created each time the function calls itself and are stored at the top of the stack. But, each time when a

function calls itself, it does not create a new copy of that function. The recursive function does not significantly reduce the size of the code and does not even improve the memory utilization, but it does some when compared to the iteration.

To terminate the recursion, you must include a select statement in the definition of the function to force the function to return without giving a recursive call to itself. The absence of the select statement in the definition of a recursive function will let the function in infinite recursion once called.

Let us understand recursion with a function which will return the factorial of the number.

```
1. int factorial(int num){
2. int answer;
3. if (num==1) {
4. return 1;
5. }else{
6. answer = factorial(num-1) * num; //recursive calling
7. }
8. return (answer);
9. }
```

In above code, the statement in else part shows the recursion, as the statement calls the function factorial( ) in which it resides.

Definition of Iteration

Iteration is a process of executing the set of instructions repeatedly till the condition in iteration statement becomes false. The iteration statement includes the initialization, comparison, execution of the statements inside the iteration statement and finally the updating of the control variable. After the control variable is updated it is compared again, and the process repeats itself, till the condition in iteration statement turns out to be false. The iteration statements are “for” loop, “while” loop, “do-while” loop.

The iteration statement does not use a stack to store the variables. Hence, the execution of the iteration statement is faster as compared to recursive function. Even the iteration function do not have the overhead of repeated function calling which also make its execution faster than recursive function. The iteration is terminated when the control condition becomes false. The absence of control condition in iteration statement may result in an infinite loop, or it may cause a compilation error.

Let’s understand iteration regarding above example.

```
1.  int factorial(int num){
2.  int answer=1; //needs initialization because it may contain a garbage value before its initialization
3.  for(int t =1; t>num; t++) //iteration
4.  {
5.  answer=answer * (t);
6.  return (answer);
7.  }
8.  }
```

In above code, the function returns the factorial of the number using iteration statement.

1. **Time Complexity:** Finding the Time complexity of Recursion is more difficult than that of Iteration.
   - **Recursion**: Time complexity of recursion can be found by finding the value of the nth recursive call in terms of the previous calls. Thus, finding the destination case in terms of the base case, and solving in terms of the base case gives us an idea of the time complexity of recursive equations. Please see Solving Recurrences for more details.
   - **Iteration**: Time complexity of iteration can be found by finding the number of cycles being repeated inside the loop.
2. **Usage:** Usage of either of these techniques is a trade-off between time complexity and size of code. If time complexity is the point of focus, and number of recursive calls would be large, it is better to use iteration. However, if time complexity is not an issue and shortness of code is, recursion would be the way to go.
   - **Recursion**: Recursion involves calling the same function again, and hence, has a very small length of code. However, as we saw in the analysis, the time complexity of recursion can get to be exponential when there are a considerable number of recursive calls. Hence, usage of recursion is advantageous in shorter code, but higher time complexity.
   - **Iteration**: Iteration is repetition of a block of code. This involves a larger size of code, but the time complexity is generally lesser than it is for recursion.
3. **Overhead:** Recursion has a large amount of Overhead as compared to Iteration.
   - **Recursion**: Recursion has the overhead of repeated function calls, that is due to repetitive calling of the same function, the time complexity of the code increases manifold.
   - **Iteration**: Iteration does not involve any such overhead.
4. **Infinite Repetition:** Infinite Repetition in recursion can lead to CPU crash but in iteration, it will stop when memory is exhausted.
   - **Recursion**: In Recursion, Infinite recursive calls may occur due to some mistake in specifying the base condition, which on never becoming false, keeps calling the function, which may lead to system CPU crash.
   - **Iteration**: Infinite iteration due to mistake in iterator assignment or increment, or in the terminating condition, will lead to infinite loops, which may or may not lead to system errors, but will surely stop program execution any further.

8.222) purpose of main function

In C, program execution starts from the main() function. Every C program must contain a main() function. The main function may contain any number of statements. These statements are executed sequentially in the order which they are written.

The main function can in-turn call other functions. When main calls a function, it passes the execution control to that function. The function returns control to main when a return statement is executed or when end of function is reached.

In C, the function prototype of the 'main' is one of the following:

int main(); //main with no arguments
int main(int argc, char *argv[]); //main with arguments

The parameters argc and argv respectively give the number and value of the program's command-line arguments.

The function main() calls / invokes other functions within it. The execution of the program always starts with main() function.

The main() function is :

- The first function to start a program
- Returns int value to the environment which called the program
- It can be called recursively.
- It is a user defined function, except the name
- Like other functions, main(0 function can receive arguments. It has a) argument count and b) argument vector(string argument)

8.221) when is it necessary to declare prototype of a function?

Function prototype
A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.
A function prototype gives information to the compiler that the function may later be used in the program.

Syntax of function prototype

```
returnType functionName(type1 argument1, type2 argument2, ...);
```

In the above example, `int addNumbers(int a, int b);` is the function prototype which provides the following information to the compiler:
name of the function is `addNumbers()`
return type of the function is `int`
two arguments of type `int` are passed to the function
The function prototype is not needed if the user-defined function is defined before the `main()` function.

Calling a function
Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

```
functionName(argument1, argument2, ...);
```

In the above example, the function call is made using `addNumbers(n1, n2);` statement inside the `main()` function.

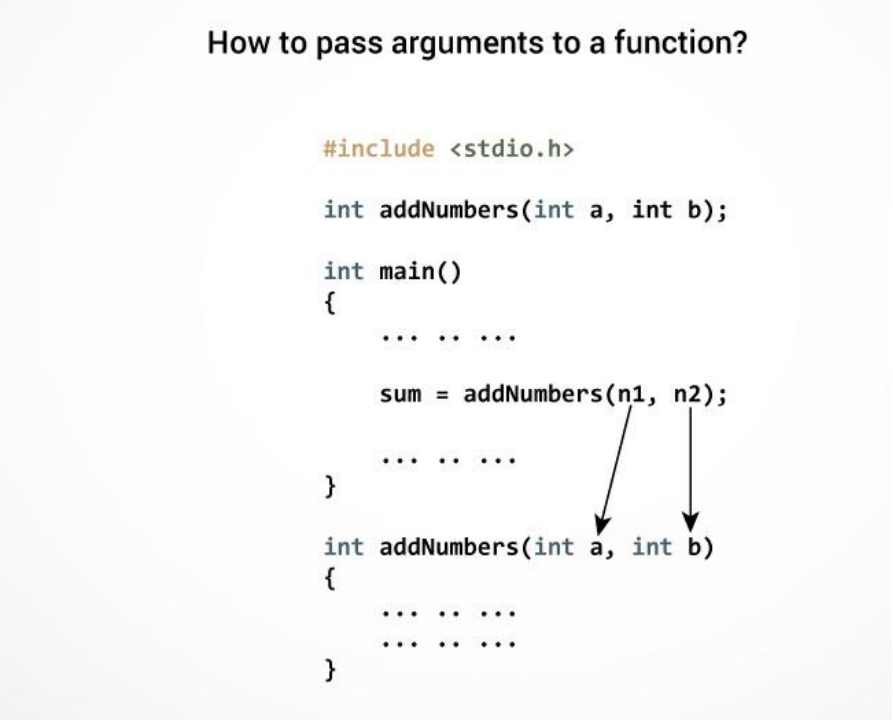
Function definition
Function definition contains the block of code to perform a specific task. In our example, adding two numbers and returning it.

Syntax of function definition

```
returnType functionName(type1 argument1, type2 argument2, ...)
{
    //body of the function
```

}
When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Passing arguments to a function
In programming, argument refers to the variable passed to the function. In the above example, two variables `n1` and `n2` are passed during the function call.
The parameters `a` and `b` accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.



The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.
If `n1` is of char type, `a` also should be of char type. If `n2` is of float type, variable `b` also should be of float type.
A function can also be called without passing an argument.

Return Statement
The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.
In the above example, the value of the `result` variable is returned to the main function.
The `sum` variable in the `main()` function is assigned this value.

What is the purpose of a function prototype?

The Function prototype serves the following purposes –

1) It tells the return type of the data that the function will return.
2) It tells the number of arguments passed to the function.
3) It tells the data types of the each of the passed arguments.
4) Also it tells the order in which the arguments are passed to the function.

Therefore essentially, function prototype specifies the input/output interlace to the function i.e. what to give to the function and what to expect from the function.

Prototype of a function is also called signature of the function.

All identifiers in C need to be declared before they are used. This is true for functions as well as variables. For functions the declaration needs to be before the first call of the function. A full declaration includes the return type and the number and type of the arguments. This is also called the function prototype.

Note: Older versions of the C language didn't have prototypes, the function declarations only specified the return type and did not list the argument types. Unless your stuck with an old compiler, function declarations should always be prototypes and this book uses the terms interchangeably.

Having the prototype available before the first use of the function allows the compiler to check that the correct number and type of arguments are used in the function call and that the returned value, if any, is being used reasonably.

The function definition itself can act as an implicit function declaration. This was used in the above example and was why `sum` was put before `main`. If the order was reversed the compiler would not recognize `sum` as a function. To correct this a prototype could be added before `main`

The prototype gives a lot of information about the function. One, it tells it the return type. Two, it tells it how many parameters there are, and what their types are. The actual names of the parameter values (`a` and `b` in our example) can be left in or out of the prototype. Generally, leaving them out leaves you with the flexibility of renaming variables at will.

Prototypes are often in a separate header file which can be included in other C source files that wish to use the functions. The header `stdio.h`, for example, contains prototypes for the functions `scanf` and `printf`. This is why our examples have been including this file, so that the compiler will let us call those functions. The actual code for these functions is kept in a library elsewhere on the system, the header file only says how to interface with the functions.

Function prototype declaration is necessary in order to provide information to the compiler about function, about return type, parameter list and function name etc.

Important Points :

Our program starts from main function. Each and every function is called directly or indirectly through main function

Like variable we also need to declare function before using it in program.

In C, declaration of function is called as prototype declaration

Function declaration is also called as function prototype

Points to remember

Below are some of the important notable things related to prototype declaration –

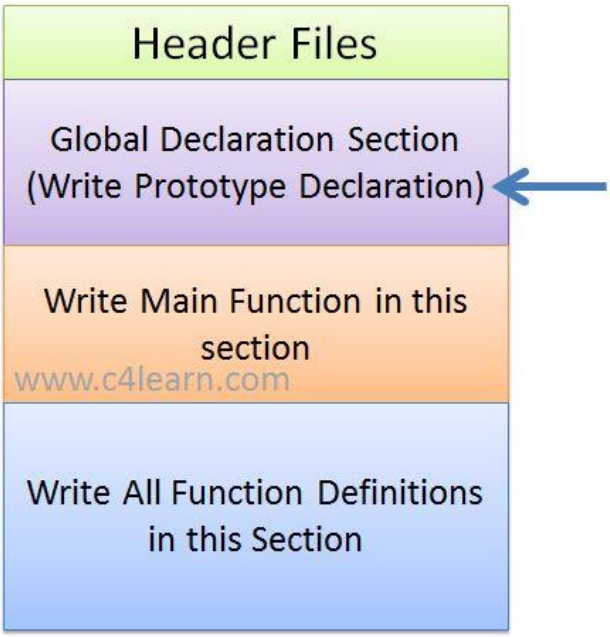
It tells name of function,return type of function and argument list related information to the compiler

Prototype declaration always ends with semicolon.

Parameter list is optional.

Default return type is integer.

Pictorial representation



Syntax

```
return_type function_name ( type arg1, type arg2..... );
```

prototype declaration comprised of three parts i.e name of the function,return type and parameter list

Examples of prototype declaration

Function with two integer arguments and integer as return type is represented using below syntax

```
int sum(int,int);
```

Function with integer argument and integer as return type is represented using below syntax

```
int square(int);
```

In the below example we have written function with no argument and no return type

```
void display(void);
```

In below example we have delclared function with no argument and integer as return type

```
int getValue(void);
```

Positioning function declaration

If function definition is written after main then and then only we write prototype declaration in global declaration section

If function definition is written above the main function then ,no need to write prototype declaration

Case 1 : Function definition written before main

```
#include<stdio.h>

void displayMessage() {
   printf("www.c4learn.com");
}

void main() {
   displayMessage();
}
```

Case 2 : Function definition written after main

```
#include<stdio.h>

//Prototype Declaration
```

```
void displayMessage();

void main() {
   displayMessage();
}

void displayMessage() {
   printf("www.c4learn.com");
}
```

Need of prototype declaration

Program Execution always starts from main , but during lexical analysis (1st Phase of Compiler) token generation starts from left to right and from top to bottom.
During code generation phase of compiler it may face issue of backward reference.

If we write prototype declaration then –

Prototype declaration tells compiler that we are going to define this function somewhere in the program.
Compiler will have prior information about function.
As compiler have prior information ,during function calling compiler looks forward in the program for the function definition.

If we don't write prototype declaration then –

Compiler don't have any reference of Function.
Compiler don't have prior information of that function.
Compiler gets confused and interpret it as unknown reference and throws error.

8.220) Types of user defined functions in C

Depending upon the presence of arguments and the return values, user defined functions can be classified into five categories.

1. **Function with no arguments and no return values**
2. **Function with no arguments and one return value**
3. **Function with arguments and no return values**
4. **Function with arguments and one return value**
5. **Function with multiple return values**

1: Function with no arguments and no return value

Function with no argument means the called function does not receive any data from calling function and **Function with no return value** means calling function does not receive any data from the called function. So there is no data transfer between calling and called function.

C program to calculate the area of square using the function with no arguments and no return values

```
/* program to calculate the area of square */
```

```
#include <stdio.h>

void area();   //function prototype

int main()

{

  area();    //function call

  return 0;

}

void area()

{

  int square_area,square_side;

  printf("Enter the side of square :");

  scanf("%d",&square_side);

  square_area = square_side * square_side;

  printf("Area of Square = %d",square_area);

}
```

Explanation

In the above program, `area( );` function calculates area and no arguments are passed to this function. The return type of this function is `void` and hence return nothing.

2: Function with no arguments and one return value

As said earlier function with no arguments means called function does not receive any data from calling function and function with one return value means one result will be sent back to the caller from the function.

C program to calculate the area of square using the function with no arguments and one return values

```
#include <stdio.h>

int area();   //function prototype with return type int

int main()

{

  int square_area;

  square_area = area();    //function call

  printf("Area of Square = %d",square_area);

  return 0;

}

int area()

{

  int square_area,square_side;

  printf("Enter the side of square :");

  scanf("%d",&square_side);

  square_area = square_side * square_side;

  return square_area;

}
```

Explanation

In this function `int area( );` no arguments are passed but it returns an integer value `square_area`.

3: Function with arguments and no return values

Here function will accept data from the calling function as there are arguments, however, since there is no return type nothing will be returned to the calling program. So it's a one-way type communication.

C program to calculate the area of square using the function with arguments and no return values

```
#include <stdio.h>

void area( int square_side);  //function prototype

int main()

{

  int square_side;

  printf("Enter the side of square :");

  scanf("%d",&square_side);

  area(square_side);   //function call

  return 0;

}

void area(int square_side)

{

  int square_area;

  square_area = square_side * square_side;

  printf("Area of Square = %d",square_area);

}
```

Explanation

In this function, the integer value entered by the user in *square_side* variable is passed to `area();` .The called function has `void` as a return type as a result, it does not return value.

This program calculates the area of a square and prints the result.

4: Function with arguments and one return value

Function with arguments and one return value means both the calling function and called function will receive data from each other. It's like a dual communication.

C program to calculate the area of square using the function with arguments and one return values

```
#include <stdio.h>

int area(int square_side);   //function prototype with return type int

int main()

{

  int square_area,square_side;

  printf("Enter the side of square :");

  scanf("%d",&square_side);

  square_area = area(square_side);    //function call

  printf("Area of Square = %d",square_area);

  return 0;

}

int area(int square_side)

{

  int square_area;

  square_area = square_side * square_side;

  return square_area;

}
```

5: Function with multiple return values

So far we have used functions that return only one value because function normally returns a single value. However, we can use functions which can return multiple values by using input parameters and output parameters. Those parameters which are used to receive data are called **input parameters** and the parameters used to send data are called **output parameters**. This is achieved by using **address operator(&)** and **indirection operator(*)**. Moreover, following example will clarify this concept.

C program to calculate the area and volume of square using the function with multiple return values

```
#include <stdio.h>

void area_volume(int l, int *a, int *v);   //function prototype

int main()

{

  int l,a,v;

  printf("Enter the side of square :");

  scanf("%d",&l);

  area_volume(l,&a,&v);    //function call

  printf("Area = %d\n Volume = %d",a,v);

  return 0;

}

void area_volume(int l, int *a, int *v)

{

 *a = l*l;

 *v = l*l*l;

}
```

Explanation

In the above program *l* is input argument, *a* and *v* are output arguments. In the function call, we pass actual value of *l* whereas addresses of *a* and *v* are passed.

8.223) what is recursion? What are the advantages?

8.224) Fibonacci series using 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();
}
```

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.

Recursion is a process in which a function calls itself as a subroutine. This allows the function to be repeated several times, since it calls itself during its execution. Functions that incorporate recursion are called recursive functions.

Recursion is often seen as an efficient method of programming since it requires the least amount of code to perform the necessary functions. However, recursion must be incorporated carefully, since it can lead to an infinite loop if no condition is met that will terminate the function.

Number Factorial

The following example calculates the factorial of a given number using a recursive function −

```
#include <stdio.h>

unsigned long long int factorial(unsigned int i) {

   if(i <= 1) {
      return 1;
   }
   return i * factorial(i - 1);
}

int  main() {
   int i = 12;
   printf("Factorial of %d is %d\n", i, factorial(i));
```

```
   return 0;
}
```

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

```
Factorial of 12 is 479001600
```

Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function −

```
#include <stdio.h>

int fibonacci(int i) {

   if(i == 0) {
      return 0;
   }

   if(i == 1) {
      return 1;
   }
   return fibonacci(i-1) + fibonacci(i-2);
}

int  main() {

   int i;

   for (i = 0; i < 10; i++) {
      printf("%d\t\n", fibonacci(i));
   }

   return 0;
}
```

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

```
0
1
1
2
3
5
8
13
21
34
```

Advantages of using recursion

1. **Recursion** is more elegant and requires a lesser number of variables which makes the program short and clean.
2. **Recursion** can be made to replace complex nesting codes since we don't have to call the program, again and again, to do the same task as it calls itself.

Disadvantages of using recursion

1. It is comparatively difficult to think of the logic of a recursive function.
2. It also sometimes becomes difficult to debug a recursive code.
3. Recursion takes a lot of stack space, usually not considerable when the program is small and running on a PC.
4. Recursion uses more processor time.

8.225) actual and formal arguments in functions

The variables declared in the function prototype or definition are known as **Formal arguments** and the values that are passed to the called function from the main function are known as **Actual arguments**.

The actual arguments and formal arguments must match in number, type, and order.

Following are the two ways to pass arguments to the function:

- **Pass by value**
- **Pass by reference**

Actual arguments

Arguments which are mentioned in the function call is known as the actual argument. For example:

```
1 func1(12, 23);
```

here `12` and `23` are actual arguments.
Actual arguments can be constant, variables, expressions etc.

```
1 func1(a, b); // here actual arguments are variable
2 func1(a + b, b + a); // here actual arguments are expression
```

Formal Arguments

Arguments which are mentioned in the definition of the function is called formal arguments. Formal arguments are very similar to local variables inside the function. Just like local variables, formal arguments are destroyed when the function ends.

```
1 int factorial(int n)
```

```
2 {
3     // write logic here
4 }
```

Here `n` is the formal argument.

Things to remember about actual and formal arguments.

1. Order, number, and type of the actual arguments in the function call must match with formal arguments of the function.
2. If there is type mismatch between actual and formal arguments then the compiler will try to convert the type of actual arguments to formal arguments if it is legal, Otherwise, a garbage value will be passed to the formal argument.
3. Changes made in the formal argument do not affect the actual arguments.

The following program demonstrates this behaviour.

```
1  #include<stdio.h>
2  void func_1(int);
3
4  int main()
5  {
6      int x = 10;
7
8      printf("Before function call\n");
9      printf("x = %d\n", x);
10
11     func_1(x);
12
13     printf("After function call\n");
14     printf("x = %d\n", x);
15
16     // signal to operating system program ran fine
17     return 0;
18 }
19
20 void func_1(int a)
21 {
22     a += 1;
23     a++;
24     printf("\na = %d\n\n", a);
25 }
```

Here the value of variable `x` is `10` before the function `func_1()` is called, after `func_1()` is called, the value of `x` inside `main()` is still `10`. The changes made inside the function `func_1()` doesn't affect the value of `x`. This happens because when we pass values to the functions, a copy of the value is made and that copy is passed to the formal arguments. Hence Formal arguments work on a copy of the original value, not the original value itself, that's why changes made inside `func_1()` is not reflected inside `main()`. This process is known as passing arguments using **Call by Value**, we will discuss this concept in more detail in upcoming chapters.

8.227) is printf a function? What does it return?

The printf() and scanf() functions are required for output and input respectively in C. Both of these functions are library functions and are defined in the stdio.h header file.

Details about the return values of the printf() and scanf() functions are given as follows:

The printf() function

The printf() function is used for printing the output. It returns the number of characters that are printed. If there is some error then it returns a negative value.

A program that demonstrates this is as follows:

```
#include <stdio.h>

int main()
{
   char str[] = "THE SKY IS BLUE";

   printf("\nThe value returned by printf() for the above string is : %d", printf("%s", str));

   return 0;
}
```

The output of the above program is as follows:

```
THE SKY IS BLUE
The value returned by printf() for the above string is : 15
```

Now let us understand the above program.

First, the string is initialized. Then the string is displayed using printf() as well as the value returned by printf(). The code snippet that shows this is as follows:

```
char str[] = "THE SKY IS BLUE";

printf("\nThe value returned by printf() for the above string is
: %d", printf("%s", str));
```

8.228) scnaf and it's return type.
The scanf() function

The scanf() function is used for obtaining the input from the user. It returns the number of input values that are scanned. If there is some input failure or error then it returns EOF (end-of-file).

A program that demonstrates this is as follows:

Example

```
#include
int main()
{
   int x, y, z;

   printf("The value returned by the scanf() function is : %d",
   scanf("%d%d%d", &x, &y, &z));

   printf("\nx = %d", x);
   printf("\ny = %d", y);
   printf("\nz = %d", z);

   return 0;
}
```

The output of the above program is as follows:

```
7 5 4
The value returned by the scanf() function is : 3
x = 7
y = 5
z = 2
```

8.244) importance of header file.

The main role of header file is it is used to share information among various files.To put it brief, if we have several functions say 4 functions named as f1, f2, f3, f4 placed in file say sample.c and if all the functions want to get accessed each other all must be placed in the same file sample.c.

In other words if there is a function say F5 placed in another file say example.c and if the function F1 placed in sample.c wants to access the function F5 placed in example1.c it is not possible. For this what we can do is add the function definition in the main program from which the function is going to get called. But this will make the process tiring if the program has lot of functions because if there is a change in arguments in functions each time programmer has to search for the function definition file and has to make change in the function definition file as well as in the actual function. These will make the process tiring.

To avoid all this and to make the process simple what one can do is have a header file and place the function declarations in a header file say for instance if the header file is sample.h where function declarations area made the programmer ahs to include this header file in the source file where the functions are used as

`#include sample.h`

In this way of there is any modification in function arguments only eth actual function and the definitions defined in the single place namely header file has to be changed.

A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

You request to use a header file in your program by including it with the C preprocessing directive **#include**, like you have seen inclusion of **stdio.h** header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in the header files and include that header file wherever it is required.

Include Syntax

Both the user and the system header files are included using the preprocessing directive **#include**. It has the following two forms −

```
#include <file>
```

This form is used for system header files. It searches for a file named 'file' in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

```
#include "file"
```

This form is used for header files of your own program. It searches for a file named 'file' in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

Include Operation

The **#include** directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the **#include** directive.

A header file is a file containing C declarations and macro definitions (see Macros) to be shared between several source files. You request the use of a header file in your program by *including* it, with the C preprocessing directive '`#include`'.

Header files serve two purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.

8.245) types of header files

Take reference from the above answer and add the below points and also write about standard libraries in c stdlib,conio,stdio,math.h,string.h

A header file in C/C++ contains:

- Function definitions
- Data type definitions
- Macros

Header files offer these features by importing them into your program with the help of a preprocessor directive called **#include.** These preprocessor directives are responsible for instructing the C/C++ compiler that these files need to be processed before compilation.
Every C program should necessarily contain the header file <**stdio.h**> which stands for standard input and output used to take input with the help of **scanf() function** and display the output using **printf() function.**
C++ program should necessarily contain the header file <**iostream**> which stands for input and output stream used to take input with the help of "**cin**>>" **function** and display the output using "**cout**<<" **function**.
From this example, it is clear that each header file of C and C++ has its own specific function associated with it.

Basically, header files are of 2 types:

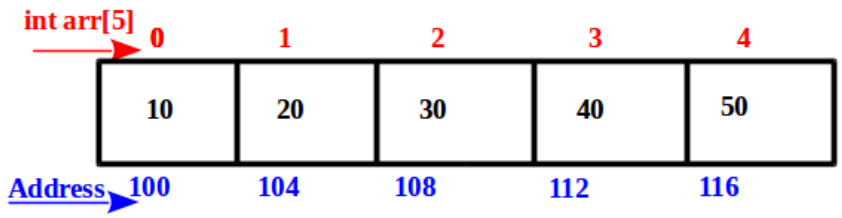
1. **Standard library header files:** These are the pre-existing header files already available in the C/C++ compiler.
2. **User-defined header files:** Header files starting #define can be designed by the user.

8.247) ARRAY OF POINTERS AND POINTER TO ARRAY DIFFERENCE.

Vivek Kumar | July 2, 2017 | c programming | 0 Comments

An **array** is known as the contiguous run of elements while a **pointer** is an address pointing variable.
A pointer could represent the same array.

```
1. int arr[5];
2. int *a;
3. a = arr;
```



Array

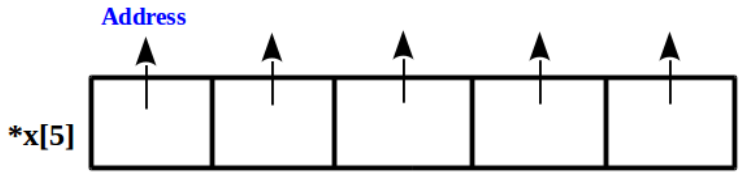
The compiler reads ***arr[2]*** as, get the base address that is 100, next add 2 as the pointer arithmetic got 108 and then dereference it. Hence we got 30.

It's like ****(arr + 2)*.** In the same we can also write ****(a + 2)* or *a[2]*, as *a = arr* means *a = &arr[0]*.** Thus an array acts like a pointer but it's not a pointer. The difference could be seen when both passed to *sizeof* method. *sizeof(arr)* gives 20 while *sizeof(a)* gives 4 (for 32 bit architecture). Therefore, array names in a C program are converted mostly to pointers, leaving cases like involving *sizeof* operator.

ARRAY OF POINTER AND POINTER TO ARRAY:

1. **int *x[5]**
2. **int (*x)[5]**

The first one is an array of pointer while the second one is a pointer to an array of 5 blocks. Let's understand step by step.
So in this post, the table of Precedence and Associativity of operators will be used, so it is better to go through this post to have a better understanding.
Operator [] has higher priority than *.
So in *int *x[5]*, first *x[5]* is read that says an array of 5 blocks. Next an asterisk * is read, means every block of the array is a pointer. See below,



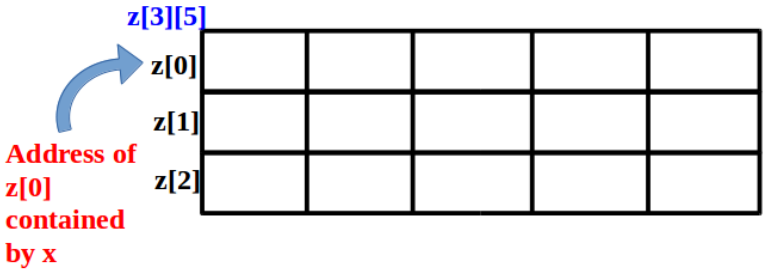
Array of Pointer

This could be seen as there are 5 different pointer variables.

In **int (*x)[5]**, as per the table both () and [] has the same precedence, associativity comes into the picture that says left to right reading. **(*x)**read first means a pointer. Next [5] that says an array of 5 blocks. Therefore, it's a pointer that points to an array of 5 blocks.

IN A 2-D OR MULTIDIMENSIONAL ARRAY

int z[3][5], this is a 2-d array i.e. three 1-d arrays of 5 blocks each. To create a pointer variable pointing to this first set of 5 blocks of this 2-d array, we'll write *x = &z[0]* or in other ways *x = z (because starting index of 2d array and first 1d array is same)*.



Pointer to an array

Doing *x++* here, will give *z[1] i.e. the starting index of 2nd 1-d array*, and for accessing any particular block using x will be like a pointer to pointer accessing. Say *z[1][4]*, it would be **(*(x+1)+4)* or simply *x[1][4]*.

What are the main difference between array of pointers and pointer to array in C programming?

Often, we would see that an array is introduced as a pointer. Technically, that is not correct. Arrays are not pointer. It is like any other variable in C++. So what are the main difference between an array of pointers and the pointer to an array in C programming?

Let's us use figurative analogy to explain.

It's Christmas and you've got a gift somewhere inside your room. You also have a card that tells you where the gift is located. This is an analogy to the pointer and array. The card that points the gift is not the gift itself. The Christmas card is the pointer and the gift is the array.

Here is another scenario. The Christmas card lists several gifts and their location in your room. The card says something like, red wine is on the bed, a green scarf is under the bed, a blue shawl on top of the nightstand.

The card and the list it contains is like an array of pointers. The list is an array with pointers to different things inside your room.

Now, you may ask about the difference between the first and second scenario. And to explain further, the difference is that in the first scenario, you wrapped a gift inside a box and made a pointer to point to the box. In the next, you have a list of pointers to things that were in a box (in this case the box is the room.).

Do you see the difference? What are the main difference between an array of pointers to pointers to array in C? It is just like the difference between a card telling the location of a gift inside your room and a list pointing to different items in your room.

The second one, pointers to array in C programming is used to maintain a reference of things that may change their location over time. The things so

to speak may not be directly related to each other. These things referred to would not make sense if placed in a single box.

The first one is used when you know that your gift or collection will remain same but you will need to pass around. This is another reference that means having different people access it.

Should you feel you have a better answer to the question, What are the main difference between array of pointers and pointer to array in C programming? Feel free to let us know.

Basically, pointer to array is a single variable that holds starting point of the array. Array of pointers is an array of variables each of which can hold an address of a variable.
Start at the variable, then go right, and left, and right...and so on.

```
int* arr1[8];
```
arr1 is an array of 8 pointers to integers.
```
int (*arr2)[8];
```
arr2 is a pointer (the parenthesis block the right-left) to an array of 8 integers.
```
int *(arr3[8]);
```
arr3 is an array of 8 pointers to integers.
This should help you out with complex declarations.

8.246) how to access array elements using a pointer? Same as above.

9.248) define pointer with example

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined −

Live Demo

```
#include <stdio.h>

int main () {

   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
```

```
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

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

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations −

Live Demo

```
#include <stdio.h>

int main () {

   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */
```

```
   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

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

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

Live Demo

```
#include <stdio.h>

int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

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

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows −

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

Pointers in Detail

Pointers have many but easy concepts and they are very important to C programming. The following important pointer concepts should be clear to any C programmer −

Sr.No.	Concept & Description
1	Pointer arithmetic There are four arithmetic operators that can be used in pointers: ++, --, +, -
2	Array of pointers You can define arrays to hold a number of pointers.
3	Pointer to pointer C allows you to have pointer on a pointer and so on.
4	Passing pointers to functions in C Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
5	Return pointer from functions in C C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.

9.249) advantages of pointers

Advantages of Pointers

- Pointers are useful for accessing memory locations.
- Pointers provide an efficient way for accessing the elements of an array structure.
- Pointers are used for dynamic memory allocation as well as deallocation.
- Pointers are used to form complex data structures such as linked list, graph, tree, etc.

Disadvantages of Pointers

- Pointers are a little complex to understand.
- Pointers can lead to various errors such as segmentation faults or can access a memory location which is not required at all.

- If an incorrect value is provided to a pointer, it may cause memory corruption.
- Pointers are also responsible for memory leakage.
- Pointers are comparatively slower than that of the variables.
- Programmers find it very difficult to work with the pointers; therefore it is programmer's responsibility to manipulate a pointer carefully.

9.252) pointer to structure

Pointer to a Structure in C

Contents [show]

We have already learned that a pointer is a variable which points to the address of another variable of any data type like `int`, `char`, `float` etc. Similarly, we can have a pointer to structures, where a pointer variable can point to the address of a structure variable. Here is how we can declare a pointer to a structure variable.

```
1  struct dog
2  {
3      char name[10];
4      char breed[10];
5      int age;
6      char color[10];
7  };
8
9  struct dog spike;
10
11 // declaring a pointer to a structure of type struct dog
12 struct dog *ptr_dog
```

This declares a pointer `ptr_dog` that can store the address of the variable of type `struct dog`. We can now assign the address of variable `spike` to `ptr_dog` using `&` operator.

```
1 ptr_dog = &spike;
```

Now `ptr_dog` points to the structure variable `spike`.

Accessing members using Pointer

There are two ways of accessing members of structure using pointer:

1. Using indirection (`*`) operator and dot (`.`) operator.
2. Using arrow (`->`) operator or membership operator.

Let's start with the first one.

Using Indirection (*) Operator and Dot (.) Operator

At this point `ptr_dog` points to the structure variable `spike`, so by dereferencing it we will get the contents of the `spike`. This means `spike` and `*ptr_dog` are functionally equivalent. To access a member of structure write `*ptr_dog` followed by a dot(`.`) operator, followed by the name of the member. For example:
`(*ptr_dog).name` – refers to the `name` of dog
`(*ptr_dog).breed` – refers to the `breed` of dog
and so on.

Parentheses around `*ptr_dog` are necessary because the precedence of dot(`.`) operator is greater than that of indirection (`*`) operator.

Using arrow operator (->)

The above method of accessing members of the structure using pointers is slightly confusing and less readable, that's why C provides another way to access members using the arrow (`->`) operator. To access members using arrow (`->`) operator write pointer variable followed by `->` operator, followed by name of the member.

```
1 ptr_dog->name - refers to the name of dog

2 ptr_dog->breed - refers to the breed of dog
```

and so on.

Here we don't need parentheses, asterisk (`*`) and dot (`.`) operator. This method is much more readable and intuitive.
We can also modify the value of members using pointer notation.

```
1 strcpy(ptr_dog->name, "new_name");
```

Here we know that the name of the array (`ptr_dog->name`) is a constant pointer and points to the 0th element of the array. So we can't assign a new string to it using assignment operator (`=`), that's why `strcpy()` function is used.

```
1 --ptr_dog->age;
```

In the above expression precedence of arrow operator (`->`) is greater than that of prefix decrement operator (`--`), so first `->` operator is applied in the expression then its value is decremented by 1.
The following program demonstrates how we can use a pointer to structure.

```
1  #include<stdio.h>

2

3  struct dog

4  {

5      char name[10];

6      char breed[10];

7      int age;

8      char color[10];

9  };
```

```
10
11 int main()
12 {
13     struct dog my_dog = {"tyke", "Bulldog", 5, "white"};
14     struct dog *ptr_dog;
15     ptr_dog = &my_dog;
16
17     printf("Dog's name: %s\n", ptr_dog->name);
18     printf("Dog's breed: %s\n", ptr_dog->breed);
19     printf("Dog's age: %d\n", ptr_dog->age);
20     printf("Dog's color: %s\n", ptr_dog->color);
21
22     // changing the name of dog from tyke to jack
23     strcpy(ptr_dog->name, "jack");
24
25     // increasing age of dog by 1 year
26     ptr_dog->age++;
27
28     printf("Dog's new name is: %s\n", ptr_dog->name);
29     printf("Dog's age is: %d\n", ptr_dog->age);
30
31     // signal to operating system program ran fine
32     return 0;
33 }
```

Expected Output:

```
1 Dog's name: tyke
2 Dog's breed: Bulldog
3 Dog's age: 5
4 Dog's color: white
5
6 After changes
7
8 Dog's new name is: jack
9 Dog's age is: 6
```

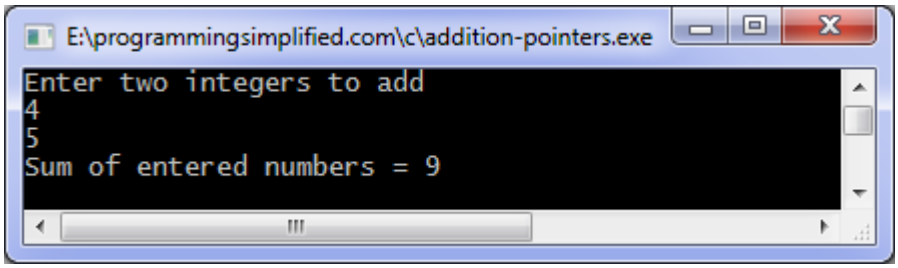
How it works:

In lines 3-9, we have declared a structure of type dog which has four members namely `name`, `breed`, `age` and `color`.

In line 13, a variable called `my_dog` of type `struct dog` is declared and initialized.
In line 14, a pointer variable `ptr_dog` of type `struct dog` is declared.
In line 15, the address of `my_dog` is assigned to `ptr_dog` using `&` operator.
In lines 17-20, the `printf()` statements prints the details of the dog.
In line 23, a new name is assigned to `ptr_dog` using the `strcpy()` function, because we can't assign a string value directly to `ptr_dog->name` using assignment operator.
In line 26, the value of `ptr_dog->age` is incremented by `1` using postfix increment operator.
Recall that postfix `++` operator and `->` have the same precedence and associates from left to right. But since postfix `++` is used in the expression first the value of `ptr_dog->age` is used in the expression then it's value is incremented by `1`.

9.254) adding two numbers using pointers

```c
#include <stdio.h>
int main()
{
   int first, second, *p, *q, sum;

   printf("Enter two integers to add\n");
   scanf("%d%d", &first, &second);

   p = &first;
   q = &second;

   sum = *p + *q;

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

   return 0;
}
```



9.255) write a program to create, initialize and use pointers. Check from above codes.

9.250) Discuss declaration syntax of pointer with example. Check from above.

9.253) pointer arithmetic

Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment

- Decrement
- Addition
- Subtraction
- Comparison

Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

1. new_address= current_address + i * size_of(data type)

Where i is the number by which the pointer get increased.

32-bit

For 32-bit int variable, it will be incremented by 2 bytes.

64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

Let's see the example of incrementing pointer variable on 64-bit architecture.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+1;
8. printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incre
   mented by 4 bytes.
9. return 0;
10. }
```

Output

```
Address of p variable is 3214864300
After increment: Address of p variable is 3214864304
```

Traversing an array by using pointer

```
#include<stdio.h>
void main ()
{
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    int i;
    printf("printing array elements...\n");
    for(i = 0; i< 5; i++)
    {
        printf("%d  ",*(p+i));
    }
}
```

Output

```
printing array elements...
1  2  3  4  5
```

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

1. new_address= current_address - i * size_of(data type)

32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

```
#include <stdio.h>
void main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-1;
printf("After decrement: Address of p variable is %u \n",p); // P will now point to the im
midiate previous location.
}
```

Output

```
Address of p variable is 3214864300
After decrement: Address of p variable is 3214864296
```

C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

1. new_address= current_address + (number * size_of(data type))

32-bit

For 32-bit int variable, it will add 2 * number.

64-bit

For 64-bit int variable, it will add 4 * number.

Let's see the example of adding value to pointer variable on 64-bit architecture.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+3;   //adding 3 to pointer variable
8. printf("After adding 3: Address of p variable is %u \n",p);
9. return 0;
10. }
```

Output

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., 4*3=12 increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., 2*3=6. As integer value occupies 2-byte memory in 32-bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

1. new_address= current_address - (number * size_of(data type))

32-bit

For 32-bit int variable, it will subtract 2 * number.

64-bit

For 64-bit int variable, it will subtract 4 * number.

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-3; //subtracting 3 from pointer variable
printf("After subtracting 3: Address of p variable is %u \n",p);
return 0;
}
```

Output

```
Address of p variable is 3214864300
After subtracting 3: Address of p variable is 3214864288
```

You can see after subtracting 3 from the pointer variable, it is 12 (4*3) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

1. Address2 - Address1 = (Subtraction of two addresses)/size of data type which pointer po ints

Consider the following example to subtract one pointer from an another.

```
#include<stdio.h>
void main ()
{
    int i = 100;
    int *p = &i;
    int *temp;
    temp = p;
    p = p + 3;
```

```
9.      printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);
10. }
```

Output

```
Pointer Subtraction: 1030585080 - 1030585068 = 3
```

Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

- Address + Address = illegal
- Address * Address = illegal
- Address % Address = illegal
- Address / Address = illegal
- Address & Address = illegal
- Address ^ Address = illegal
- Address | Address = illegal
- ~Address = illegal

- A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -
- To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer −
- `ptr++`
- After the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If **ptr** points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

9.256) difference between header and library file

HEADER FILES	LIBRARY FILES
They have the extension .h	They have the extension .lib

HEADER FILES	LIBRARY FILES
They contain function declaration.	They contain function definations
They are available inside “include sub directory” which itself is in Turbo compiler.	They are available inside “lib sub directory” which itself is in Turbo compiler.
Header files are human readable.Since they are in the form of source code.	Library files are non human readable.Since they are in the form of machine code.
Header files in our program are included by using a command #include which is internally handle by pre-processor.	Library files in our program are included in last stage by special software called as linker.

Math.h is a header file which includes the prototype for function calls like sqrt(), pow() etc, whereas **libm.lib**, **libmmd.lib**, **libmmd.dll** are some of the math libraries. In simple terms a header file is like a visiting card and libraries are like a real person, so we use visiting card(Header file) to reach to the actual person(Library).

What is Library File?

A library file will have the function definitions for the declared functions in the header file. Function definitions are the actual implementation of the function. The programmer uses the functions declared in the header files in the program. It is not necessary to implement them from the beginning. When compiling the program, the compiler finds the definitions in library file for the declared functions in the header file.

Even though the header files are included in the program by the programmer, the related library files are found by the compiler automatically. Therefore, the compiler uses the library files to find the actual implementations of the declared functions in the header files. If printf() function is used in the program, the definition for how it works is in the related library file. If math.h is the header file, math.lib is the library file.

9.251) Why is it desirable to pass pointer to a variable as a function parameter than variable?

Passing Pointer to a Function in C Programming

In this example, we are passing a pointer to a function. When we pass a pointer as an argument instead of a variable then the address of the variable is passed instead of the value. So any change made by the function using the pointer is permanently made at the address of passed variable. This technique is known as call by reference in C.

Try this same program without pointer, you would find that the bonus amount will not reflect in the salary, this is because the change made by the function would be done to the local variables of the function. When we use pointers, the value is changed at the address of variable

```
#include <stdio.h>
void salaryhike(int  *var, int b)
{
    *var = *var+b;
}
int main()
{
    int salary=0, bonus=0;
    printf("Enter the employee current salary:");
    scanf("%d", &salary);
    printf("Enter bonus:");
    scanf("%d", &bonus);
    salaryhike(&salary, bonus);
    printf("Final salary: %d", salary);
    return 0;
}
```

Output:

```
Enter the employee current salary:10000
Enter bonus:2000
Final salary: 12000
```

When do we pass arguments by reference or pointer?

In C++, variables are passed by reference due to following reasons:

1) *To modify local variables of the caller function:* A reference (or pointer) allows called function to modify a local variable of the caller function. For example, consider the following example program where *fun()* is able to modify local variable *x* of *main()*.

2) *For passing large sized arguments:* If an argument is large, passing by reference (or pointer) is more efficient because only an address is really passed, not the entire object. For example, let us consider the following *Employee* class and a function *printEmpDetails()* that prints Employee details.
The problem with above code is: every time *printEmpDetails()* is called, a new Employee object is constructed that involves creating a copy of all data members. So a better implementation would be to pass Employee as a reference.

3) *To avoid Object Slicing:* If we pass an object of subclass to a function that expects an object of superclass then the passed object is sliced if it is pass by value. For example, consider the following program, it prints “This is Pet Class”.

4) *To achieve Run Time Polymorphism in a function*
We can make a function polymorphic by passing objects as reference (or pointer) to it. For example, in the following program, print() receives a reference to the base

class object. print() calls the base class function show() if base class object is passed, and derived class function show() if derived class object is passed.

As a side note, it is a recommended practice to make reference arguments const if they are being passed by reference only due to reason no. 2 or 3 mentioned above. This is recommended to avoid unexpected modifications to the objects.

9.275) What are recognizers and generators?

Language Recognizers accept a Language.
Recognizers are Machines. The Machines take a string as input.
The Machines will accept the input if when run, the Machine stops at an accept state. Otherwise the input is rejected. If a Machine M recognizes all strings in Language L, and accepts input provided by a given string S, M is said to accept S. Otherwise M is said to reject S. S is in L if and only if M accepts S.

Language Generators create the strings of a Language.
Generators are string constructors. A generator provides a construction description. If a generator is able to construct all stings in a Language L, and every string S that can be constructed by that generator is in L, we can say that the generator is a generator for the language L. If there is no way to construct a string S from the generator, S is not in L.

1. Define syntax and semantics

Syntax – is the form or structure of the expressions, statements and program units.
Semantics – is the meaning of expressions, statements and program units.

3. Describe the operation of a general language generator.

A general language recognizer is a recognition device capable of reading strings of characters from the alphabet. It would analyze the given string and it would either accept or reject the string based from the language given. These recognition devices are like filters separating correct sentences from those that are incorrectly. A recognizer is used in the syntax analysis part of the compiler. In this role, the recognizer need not test all possible strings of characters from some set to determine whether each is in the language. The syntax analyzer just determines whether the given programs are syntactically correct.

4. Describe the operation of a general language recognizer.

A general language generator is a device that can be used to generate the sentences of the language. It generates unpredictable sentences which makes a generator seems to be a device of limited usefulness as language descriptor. However, people prefer certain forms of generators over recognizer because they can be more easily read and understand them. By contrast, the syntax-checking portion of the compiler which is a language recognizer is not as useful a language descriptor for programmers because it can be used in a trial-and-error mode. For example, to determine the correct syntax of a particular statement using the compiler, the programmer can only submit a guessed-at version and see if the compiler accepts it. On the other hand, it is often possible to determine whether the syntax of a particular statement is correct by comparing it with the structure of the generator.

Dynamic semantics is describing the meaning of the programs. Programmers need to know precisely what statements of a language do. Compile writers determine the semantics of a language for which they are writing compilers from English descriptions

Semantics refers to the meaning behind a sentence in a language. Developing mathematical rules to describe semantics has proven to be an elusive undertaking. It is possible to define a meaning for each token in an alphabet, but when tokens

are placed together there are subtle interactions in meaning that make precise semantic definition of a language quite difficult. This is a major reason for continued difficulties in making modern languages completely portable.

Take the following English language example:

```
The bank president ate a plate of spaghetti.
```

This is a good solid sentence that drives AI gurus nuts. Programs that attempt to understand the semantics of English often translate this to mean the president of a river bank at a plate, and then a bunch of spaghetti fell on his lap. (I last took a course in AI about 4 years ago. This little problem may have been overcome by now.)

Solutions for describing the semantics of a language tend to be exponential, since interactions between a modifier and the object of the modifier have to be described. Consider how you would tell a computer, in a generic way, that *bank* in the above example refers to a financial institution. After you've developed a solution consider how the computer would then handle the following:

```
The boat washed up onto the bank.
```

When developing a compiler there are generally well accepted semantics to most constructs. Assignment statements and for loops exist in some form in most languages. Where semantics are unclear, as in a construct unique to a particular language, English language descriptions generally have to suffice. Hopefully the English language description can be made clear enough that all compilers based on the description will work the same way.

In order to define the syntax for a language we start with an *alphabet*. An alphabet is a series of *tokens*, not necessarily letters, that can be used to create *strings* in the alphabet. A string is a series of tokens pieced together according to the rules of the *language*. Strings are also sometimes referred to as sentences. So in order to define the syntax for a language, we need an alphabet and a set of rules used to manipulate the alphabet. The rules and the alphabet can be expressed together in one of two forms: Chomsky Normal Form (CNF) or Extended Backus Naur Form (EBNF). Both of these forms will be covered in detail. A description of the syntax of a language in either of these forms is called a *grammar*.

If we take English as an example of a language by the above definition we find something quite interesting. Namely that what we normally understand as the alphabet is not what we define above as being the alphabet. According to our mathematical definition of a language, the words of the English language are the alphabet of the English language.

According to the rules of the English language, a sentence must have an object and an action on that object. In addition a sentence can have prepositional phrases, modifiers, and other constructs, which we do not necessarily need for this example. The following sentence is an example of an English language sentence:

```
The cat sat.
```

In this example there are three tokens, or members of the alphabet: *The*, *cat* and *sat*. There is an object *cat* and a verb *sat*. The following are not English language sentences:

```
Sat ran run jump.
The paper on compilers.
```

We will refer to these as *phrases*. The first phrase is made of four tokens. It does not meet the rules of English for two reasons. The first is that it is missing an object, and the second is that it has four actions, not one. The second phrase also contains four tokens, and is not an English sentence because it is missing an object.

The alphabet is sometimes referred to as the *lexical elements* of a language.

Part of the task of a compiler is to determine if a program meets the rules of the language. If not, it contains a syntax error.

9.276) Discuss Back Naur Form

Backus-Naur notation (shortly BNF) is a formal mathematical way to describe a language, (to describe the syntax of the programming languages). The Backus-Naur Form is a way of defining syntax. It consists of • a set of terminal symbols • a set of non-terminal symbols • a set of production rules of the form Left-Hand-Side ::= Right-Hand-Side

where the LHS is a non-terminal symbol and the RHS is a sequence of symbols (terminals or non-terminals). • The meaning of the production rule is that the non-terminal on the left hand side may be replaced by the expression on the right hand side.

• Any sentence which is derived using the production rules is said to be syntactically correct. It is possible to check the syntax of a sentence by building a parse tree to show how the sentence is derived from the production rules. If it is not possible to build such a tree then the sentence has syntax errors. • Syntax rules define how to produce well-formed sentences. But this does not imply that a well formed sentence has any sensible meaning. Semantics define what a sentence means. • It is used to formally define the grammar of a language

How it works ? BNF is sort of like a mathematical game: you start with a symbol (called the start symbol and by convention usually named S in examples) and are then given rules for what you can

replace this symbol with. The language defined by the BNF grammar is just the set of all strings you can produce by following these rules. The rules are called production rules, and look like this: symbol := alternative1 | alternative2 ... • A production rule simply states that the symbol on the left-hand side of the := must be replaced by one of the alternatives on the right hand side. • The alternatives are separated by |s. (One variation on this is to use ::= instead of :=, but the meaning is the same.) Alternatives usually consist of both symbols and something called terminals. • Terminals are simply pieces of the final string that are not symbols. • There is one special symbol in BNF: @, which simply means that the symbol can be removed. If you replace a symbol by @ you do it by just removing the symbol. This is useful because in some cases it is difficult to end the replacement process without using this trick. • So, the language described by a grammar is the set of all strings you can produce with the production rules. If a string cannot in any way be produced by using the rules the string is not allowed in the language.

A real example Below is a sample BNF grammar: S := '-' FN | FN FN := DL | DL '.' DL DL := D | D DL D := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' • The different symbols here are all abbreviations: S is the start symbol, FN produces a fractional number, DL is a digit list, while D is a digit. • Valid sentences in the language described by this grammar are all numbers, possibly fractional, and possibly negative. To produce a number, start with the start symbol S • Then replace the S symbol with one of its productions. In this case we choose not to put a '-' in front of the number, so we use the plain FN production and replace S by FN: • The next step is then to replace the FN symbol with one of its productions. We want a fractional number, so we choose the production that creates two decimal lists with a '.' between them, and after that we keep choosing replacing a symbol with one of its productions once per line in the example below: Step1: DL . DL Step2: D . DL Step3: 3 . DL Step4: 3 . D DL Step5: 3 . D D Step6: 3 . 1 D Step7: 3 . 1 4 Here we've produced the fractional number 3.14.

9.278) discuss the classes of grammars given by Noam Chomsky

Chomsky Normal Form

Language theory owes a great deal to Noam Chomsky. In the 1950s Chomsky studied many spoken languages and attempted to develop a formal method of describing language. Among other achievements, it was Chomsky who first divided language study into syntax and semantics. Another achievement was the development of a language used to specify languages.

Chomsky Normal Form (CNF) uses a series of intermediate tokens to describe syntax rules. The left side of an expression in CNF shows a string with intermediate symbols, the right side shows how that string can be translated. The translation may contain terminal symbols, which are the tokens in the language's alphabet, or it may contain intermediate symbols, or it may contain a combination of the two. Traditionally terminals are shown with lower case letters and intermediate symbols are shown in upper case. A CNF grammar always starts with the intermediate symbol S.

```
S -> a
```

In this exceedingly simple example, the only sentence possible sentence in the grammar is the single token a.

```
S -> aBa
B -> bb
```

This example again shows a grammar that can only accept a single sentence. S is replaced by aBa, and B is replaced by bb, giving the sentence abba. In order to add flexibility to the grammar we need to be able to make choices in how a symbol can be expanded. CNF shows a choice in two ways. The symbol may be repeated on the right side to show multiple rules for expansion:

```
S -> aBa
B -> bb
B -> aa
```

Or the vertical bar can be used to show a choice. The following example is exactly equivalent to the previous example:

```
S -> aBa
B -> bb | aa
```

This grammar can produce two different sentences: abba or aaaa. Using the methods outlined so far we can create a grammar with a large number of intermediate symbols that can produce a large number of distinct sentences. But all the sentences such a grammar will create will have a definite length. In order to create a useful language, either spoken or programming, we need to be able to create sentences of arbitrary length. The only way CNF can be used to define a grammar that can create sentences of arbitrary length is to allow a symbol to appear in its own expansion rules.

```
S -> aBa
B -> bb | aaB
```

Such a language is call recursive. Some sentences that can be generated with this grammar include abba, aaabba, aaaaabba, etc. The number of a's before the final bba is quite arbitrary.

For a more complex example of CNF, we can look at the more or less standard definition of expression used by most computer languages.

```
S -> EXPRESSION
EXPRESSION -> TERM | TERM + EXPRESSION | TERM - EXPRESSION
TERM -> FACTOR | FACTOR * EXPRESSION | FACTOR / EXPRESSION
FACTOR -> NUMBER | ( EXPRESSION )
NUMBER -> 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |
          1 NUMBER | 2 NUMBER | 3 NUMBER | 4 NUMBER |
          5 NUMBER | 6 NUMBER | 7 NUMBER | 8 NUMBER |
          9 NUMBER | 0 NUMBER
```

This grammar is fairly small and compact, but it is also quite complex. There is a lot of ground to cover here. Analyzing it will be well worth the effort, however.

Notice first of all how the expression 123 is covered by this grammar. S is expanded to EXPRESSION, which expands to FACTOR, which expands to TERM, which expands to NUMBER. Number expands to 1 NUMBER. NUMBER then expands to 2 NUMBER. And finally this NUMBER expands to 3.

Next notice how 2 + 3 * 6 expands. S expands to EXPRESSION, which expands to TERM + EXPRESSION. TERM expands to FACTOR, then to NUMBER, then to 2, giving us 2 + EXPRESSION. EXPRESSION expands to TERM, which expands to FACTOR * EXPRESSION. FACTOR expands to NUMBER, which then expands to 3. EXPRESSION expands to TERM, which expands to FACTOR, which expands to NUMBER, which finally expands to 6.

This expression is more complex than it first appears, and requires a deeper analysis. First of all we need to look at the *precedence* of the operators. Given the expression 2 + 3 * 6, according to standard rules of arithmetic we need to perform the 3 * 6 before we add 2, giving a result of 20. Using the above grammar to develop a compiler will do that. If we reversed the rules for TERM and EXPRESSION, we would have the opposite case, where 2 is added to 3, then the result is multiplied by 6, giving 30.

The next thing we need to consider is that all recursion is on the right side of the expansions. All intermediate symbols on the left side are on a lower level than the intermediate symbol being expanded. Such a language is called a *right recursive* language. Consider what would happen if you based a compiler on a grammar with the following rules:

```
S -> EXPRESSION
EXPRESSION -> TERM | EXPRESSION + TERM | EXPRESSION - TERM
TERM -> FACTOR | EXPRESSION * FACTOR | EXPRESSION / FACTOR
FACTOR -> NUMBER | ( EXPRESSION )
NUMBER -> 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |
          1 NUMBER | 2 NUMBER | 3 NUMBER | 4 NUMBER |
          5 NUMBER | 6 NUMBER | 7 NUMBER | 8 NUMBER |
          9 NUMBER | 0 NUMBER
```

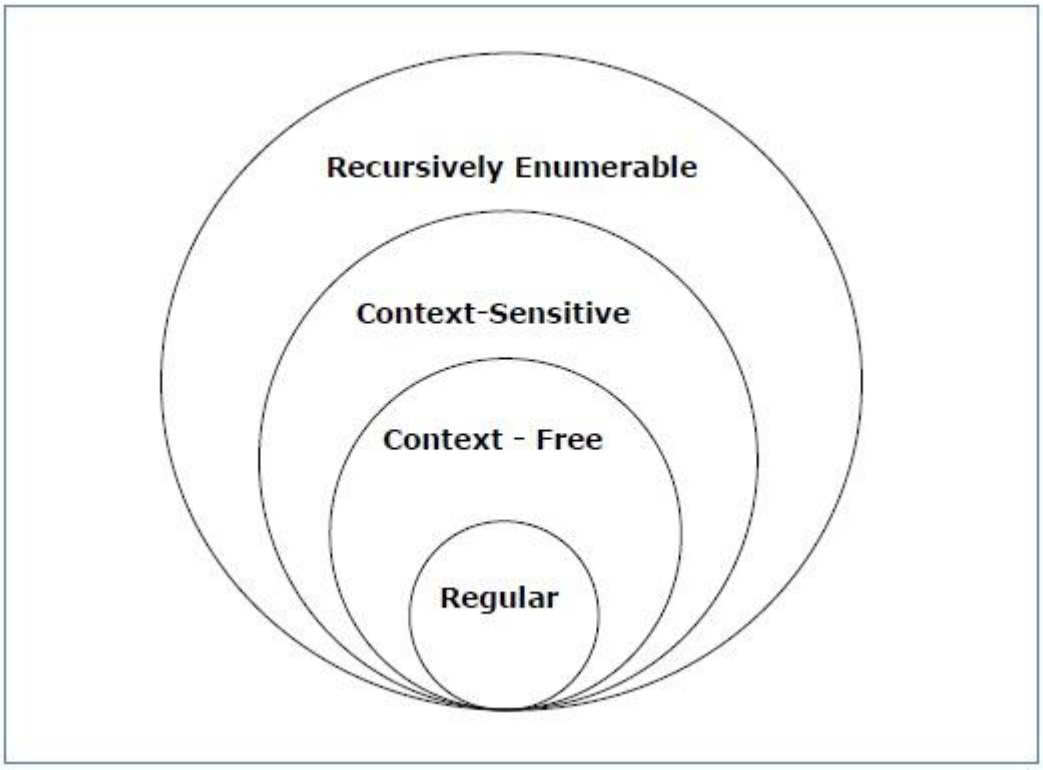
To expand 1 + 2, first S would expand to EXPRESSION, then EXPRESSION would expand to EXPRESSION. This is an infinite loop. Such a left recursive language cannot be read by mechanical means. (Strictly speaking, this is not true - the entire program can be placed on a stack then parsed backwards. This is quite inefficient. Real problems start when one rule is right recursive and another in the same grammar is left recursive).

Another interesting example is one with parentheses. It will be left to the reader to develop such an example and see how it expands.

According to Noam Chomosky, there are four types of grammars − Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other −

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

Take a look at the following illustration. It shows the scope of each type of grammar −



Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X → a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

```
X → ε
X → a | aY
Y → b
```

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form **A → γ**

where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

```
S → X a
X → a
X → aX
X → abc
X → ε
```

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

α A β → α γ β

where **A ∈ N** (Non-terminal)

and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals)

The strings **α** and **β** may be empty, but **γ** must be non-empty.

The rule **S → ε** is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

```
AB → AbBc
A → bcA
B → b
```

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of **α** → **β** where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

Example

```
S → ACaB
Bc → acB
CB → DB
aD → Db
```

The following table summarizes each of Chomsky's four types of grammars, the class of language it generates, the type of automaton that recognizes it, and the form its rules must have.

Grammar	Languages	Automaton	Production rules (constraints)*	Examples[3]
Type-0	Recursively enumerable	Turing machine	(no constraints)	describes a terminating Turing machine
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine		
Type-2	Context-free	Non-deterministic pushdown automaton		
Type-3	Regular	Finite state automaton	and	
* Meaning of symbols:				

- = terminal
- , = non-terminal
- , , , = string of terminals and/or non-terminals
  - , , = maybe empty
  - = never empty

Note that the set of grammars corresponding to recursive languages is not a member of this hierarchy; these would be properly between Type-0 and Type-1.

Every regular language is context-free, every context-free language is context-sensitive, every context-sensitive language is recursive and every recursive language is recursively enumerable. These are all proper inclusions, meaning that there exist recursively enumerable languages that are not context-sensitive, context-sensitive languages that are not context-free and context-free languages that are not regular.[4]

Type-0 grammars[edit]

Main article: Unrestricted grammar

Type-0 grammars include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. These languages are also known as the *recursively enumerable* or *Turing-recognizable* languages.[5] Note that this is different from the recursive languages, which can be *decided* by an always-halting Turing machine.

Type-1 grammars[edit]

Main article: Context-sensitive grammar

Type-1 grammars generate context-sensitive languages. These grammars have rules of the

form with a nonterminal and , and strings of terminals and/or

nonterminals. The strings and may be empty, but must be nonempty. The

rule is allowed if does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input.)

Type-2 grammars[edit]

Main article: Context-free grammar

Type-2 grammars generate the context-free languages. These are defined by rules of the

form with being a nonterminal and being a string of terminals and/or nonterminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context-free languages—or rather its subset of deterministic context-free language—are the theoretical basis for the phrase structure of most programming languages, though their syntax also includes context-sensitive name resolution due to declarations and scope. Often a subset of grammars is used to make parsing easier, such as by an LL parser.

Type-3 grammars[edit]

Main article: Regular grammar

Type-3 grammars generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single nonterminal (right regular). Alternatively, the right-hand side of the grammar can consist of a single terminal, possibly preceded by a single nonterminal (left regular). These generate the same languages. However, if left-regular rules and right-regular rules are combined,

the language need no longer be regular. The rule is also allowed here if does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

9.281) Extended BNF

Extended Backus Naur Form

A short time after Chomsky devised CNF, two researchers, Backus and Naur, independently developed a similar form for specifying language grammar. The Backus Naur form can specify some languages more compactly than CNF. Over the years other researchers have added symbols to Backus Naur Form, creating Extended Backus Naur Form (EBNF). Grammars specified in either CNF or EBNF can be converted directly into a compiler, however most compiler writers prefer to work with EBNF.

EBNF uses the symbol :== to specify the right and left sides of a rule. Terminal symbols are placed in single quotes.

```
S :== 'a' B 'a'
B :== 'bb'
```

The vertical bar is again used to represent a choice in expansion rules and just as in CNF, recursion is used to develop strings of arbitrary length. Two symbols added to EBNF that do not exist in CNF are the square brackets ([]) and curly braces ({}). The square brackets are used to denote zero or one occurrence of an expansion, and curly braces are used to denote an arbitrary, but at least one, number of expansions.

```
S :== 'a' [B]
B :== {'a'}
```

In this example, S expands to a or aB, since there can be 0 or 1 B's. B expands to a or aa or aaa or aaaa, etc. We can rewrite the expression grammar from the last section in EBNF:

```
S :== EXPRESSION
EXPRESSION :== TERM | TERM { [+,-] TERM] }
```

```
TERM :== FACTOR | FACTOR { [*,/] FACTOR] }
FACTOR :== NUMBER | '(' EXPRESSION ')'
NUMBER :== '1' | '2' | '3' | '4' | '5' |
           '6' | '7' | '8' | '9' | '0' |
           '1' NUMBER | '2' NUMBER | '3' NUMBER |
           '4' NUMBER | '5' NUMBER | '6' NUMBER |
           '7' NUMBER | '8' NUMBER | '9' NUMBER | '0' NUMBER
```

This grammar looks a little different. The only recursive rule is in FACTOR. This is because the curly braces allow strings of arbitrary length to be created without recursion. When translating this to a compiler, recursive rules translate to recursive procedure calls, rules with curly braces translate to loops and rules with square brackets translate to if statements.

To expand the expression 1 + 2 + 3 * 4 + 5, start with S expanding to TERM [+ TERM]. The first term expands to FACTOR, then to NUMBER, then to 1. The second term is really an arbitrary number of terms, since it is in square brackets. The compiler will expand it to + 2 + TERM + TERM, then expand the first TERM to FACTOR * FACTOR, giving FACTOR * FACTOR + TERM. Next the FACTORs will be expanded, giving 3 * 4 + TERM, and then the final term will be expanded in the end to 5.

Sometimes it is useful to denote zero to n instances of a rule. This is done by using curly braces with an asterisk after them ({}*).

The same concerns we had to consider with CNF follow for EBNF. A left recursive language is a left recursive language, no matter what grammar is used to specify it, and left recursive languages cannot be parsed by mechanical means. We must also worry about operator precedence no matter what grammar we use.

Even though EBNF is rather more confusing than CNF, we will tend to use it over CNF. First of all EBNF supports loops, so we can avoid some recursion, allowing for more efficient compilers. Also EBNF is more standard for compiler writers than CNF. When we look at translating EBNF into a parser it will become easier to understand how EBNF works.

The process of translating grammars in either form is actually quite mechanical. This has lead to a number of tools for automatically creating compilers. The most well known tool is the UNIX program YACC (Yet Another Compiler Compiler). (I wanted to provide a link here to a page with a good YACC explanation. The best I've found is the SunOS YACC man page, but I did find an excellent page on BISON, the GNU version of YACC.) This program is fed a grammar and outputs source code in C for a parser to match the grammar. I'm not sure if this program uses CNF or EBNF form. (The BEACON back end was written in Ada, so we did not use YACC.) Another tool for automatically developing parsers has been built into the PROLOG programming language. By entering a CNF grammar

into the PROLOG interpreter, you can effectively turn it into a parser for any right recursive language.

9.277) Context Free Grammer

Context free grammar

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

1. G= (V, T, P, S)

Where,

G describes the grammar

T describes a finite set of terminal symbols.

V describes a finite set of non-terminal symbols

P describes a set of production rules

S is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Example:

L= {wcw^R | w € (a, b)*}

Production rules:

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba

3. S ⇒ abbSbba
4. S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

Some text are missing so check Wikipedia regarding this matter.

In formal language theory, a **context-free grammar** (**CFG**) is a certain type of formal grammar: a set of production rules that describe all possible strings in a given formal language. Production rules are simple replacements. For example, the rule

replaces with . There can be multiple replacement rules for any given value. For example,

means that can be replaced with either or In context-free grammars, all rules are one-to-one, one-to-many, or one-to-none. These rules can be applied regardless of context. The left-hand side of the production rule is always a nonterminal symbol. This means that the symbol does not appear in the resulting formal language. So in our case, our language contains the letters and but not

Rules can also be applied in reverse to check whether a string is grammatically correct according to the grammar.

Here is an example context-free grammar that describes all two-letter strings containing the letters or

If we start with the nonterminal symbol then we can use the rule to turn into . We can then apply one of the two later rules. For example, if we apply to the first we get . If we then apply to the second we get . Since both and are terminal symbols, and in context-free grammars terminal symbols never appear on the left hand side of a production rule, there are no more rules that can be applied. This same process can be used, applying the last two rules in different orders in order to get all possible strings within our simple context-free grammar.

Languages generated by context-free grammars are known as context-free languages (CFL). Different context-free grammars can generate the same context-free language. It is important to distinguish the properties of the language (intrinsic properties) from the properties of a particular grammar (extrinsic properties). The language equality question (do two given context-free grammars generate the same language?) is undecidable.

Context-free grammars arise in linguistics where they are used to describe the structure of sentences and words in a natural language, and they were in fact invented by the linguist Noam Chomsky for this purpose. By contrast, in computer science, as the use of recursively-defined concepts increased, they were used more and more. In an early application, grammars are used to describe the structure of programming languages. In a newer application, they are used in an essential part of the Extensible Markup Language (XML) called the *Document Type Definition*.[2]

In linguistics, some authors use the term **phrase structure grammar** to refer to context-free grammars, whereby phrase-structure grammars are distinct from dependency grammars. In computer science, a popular notation for context-free grammars is Backus–Naur form, or *BNF*.

Classification of Context Free Grammars

Context **F**ree **G**rammars (CFG) can be classified on the basis of following two properties:
1) Based on number of strings it generates.

- If CFG is generating finite number of strings, then CFG is **Non-Recursive** (or the grammar is said to be Non-recursive grammar)

- If CFG can generate infinite number of strings then the grammar is said to be **Recursive** grammar

During Compilation, the parser uses the grammar of the language to make a parse tree(or derivation tree) out of the source code. The grammar used must be unambiguous. An ambiguous grammar must not be used for parsing.

2) Based on number of derivation trees.

- If there is only 1 derivation tree then the CFG is unambiguous.
- If there are more than 1 derivation tree, then the CFG is ambiguous.

Examples of Recursive and Non-Recursive Grammars

Recursive Grammars

```
1) S->SaS

   S->b
```

The language(set of strings) generated by the above grammar is :{b, bab, babab,…}, which is infinite.

```
2) S-> Aa

   A->Ab|c
```

The language generated by the above grammar is :{ca, cba, cbba …}, which is infinite.

Note: A recursive context-free grammar that contains no useless rules necessarily produces an infinite language.

Non-Recursive Grammars

```
   S->Aa

   A->b|c
```

The language generated by the above grammar is :{ba, ca}, which is finite.

Types of Recursive Grammars

Based on the nature of the recursion in a recursive grammar, a recursive CFG can be again divided into the following:

- Left Recursive Grammar (having left Recursion)
- Right Recursive Grammar (having right Recursion)
- General Recursive Grammar(having general Recursion)

Note:A linear grammar is a context-free grammar that has at most one non-terminal in the right hand side of each of its productions.

9.283) static semantics. Discuss

The static semantics of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (syntax rather than semantics). Many static semantic rules of a language state its type constraints. Static semantics is so named because the analysis required checking these specifications can be done at compile time.The dynamic semantics is the meaning, of expressions, statement, and program units. Because of the power of the naturalness of the available notation, describing syntax is a relatively simple matter. On the other hand, no universally accepted notation has been devised for dynamic semantics. The dynamic semantics approaches investigate how the interpretation of a natural language expression changes the context.

Static and Dynamic Semantics

Semantic Model

Users and implementors of a language need to understand exactly what each construct in a given programming language means i.e. how it interacts with the underlying hardware. However, different computers have different idiosyncrasies so language designers often describe the meaning in terms of a simplified model.

Computer memory is linear, consisting of consecutively numbered words, each of precisely the same length. We do not really care whether two variables are stored adjacently or not (except possibly for elements of arrays), nor how many bits they occupy.

In our model, there are things called locations into which values may be put, and which are big enough to hold whatever we want to put into them. The only important property about locations is that they are all disjoint.

The actual hardware stores our program, as machine code, in the same memory as our data. Our model does not need to know if or how the program is stored, unless we have a language in which programs can modify themselves. We will consider a program to be a text, containing useful things like identifiers, which does not need to be stored in the actual hardware (we will ignore debuggers, such as dbxtool, so identifiers are useless at run-time). Neverless, our model will talk about the program text, and in particular about identifiers, while we describe what is supposed to happen at run-time. Thus, we can talk about the location known to the program as fred, even though the identifier fred will have been completely eliminated by the time the program runs on the actual hardware.

Static and dynamic properties

We can distinguish between the static (external) world, representing the program text, which does not change, and the dynamic (internal) world, representing the

hardware at run-time, which is where it all happens.
Static objects are constructs (identifiers, statements, expressions etc.) in the text of the program, and have no meaningful existence beyond compile-time.
Dynamic objects are (instances of) values, locations and the like, which live and move and have their being inside the computer at run-time.

A correspondence between some static objects and their dynamic counterparts may be established by **binding**, brought about as a consequence of a declaration.

Static objects depend for their meaning upon their **static environment**, which knowns all about the other relevant static objects which surround it.

In particular, the static environment must include what is known about each identifier from its declaration. In fact, the static environment maps each identifier to its type and the kind of declaration it came from.

The static environment is invariant over time, but varies according to position within the program text.

The **dynamic environment** relates identifiers to the dynamic objects that will be around at run-time i.e. it maps each identifier to information about constants or variables or operations etc. It will vary over time, as the program runs.

9.285) operational semantic

Operational semantics is a category of formal programming language semantics in which certain desired properties of a program, such as correctness, safety or security, are verified by constructing proofs from logical statements about its execution and procedures, rather than by attaching mathematical meanings to its terms (denotational semantics). Operational semantics are classified in two categories: **structural operational semantics** (or **small-step semantics**) formally describe how the *individual steps* of a computation take place in a computer-based system; by opposition **natural semantics** (or **big-step semantics**) describe how the *overall results* of the executions are obtained. Other approaches to providing a formal semantics of programming languages include axiomatic semantics and denotational semantics.

The operational semantics for a programming language describes how a valid program is interpreted as sequences of computational steps. These sequences then *are* the meaning of the program. In the context of functional programs, the final step in a terminating sequence returns the value of the program. (In general there can be many return values for a single program, because the program could be nondeterministic, and even for a deterministic program there can be many computation sequences since the semantics may not specify exactly what sequence of operations arrives at that value.)

Perhaps the first formal incarnation of operational semantics was the use of the lambda calculus to define the semantics of LISP.[1] Abstract machines in the tradition of the SECD machine are also closely related

10.287) what is denotational semantic

In computer science, **denotational semantics** (initially known as **mathematical semantics** or **Scott–Strachey semantics**) is an approach of formalizing the meanings of programming languages by constructing mathematical objects (called *denotations*) that describe the meanings of expressions from the languages. Other approaches provide formal semantics of programming languages including axiomatic semantics and operational semantics.

Broadly speaking, denotational semantics is concerned with finding mathematical objects called domains that represent what programs do. For example, programs (or program phrases) might be represented by partial functions or by games between the environment and the system.

An important tenet of denotational semantics is that *semantics should be compositional*: the denotation of a program phrase should be built out of the denotations of its subphrases.

In computer science, denotational semantics is an approach for providing mathematical meaning to systems and programming languages. In other words, denotational semantics is a formal technique for expressing the semantic definition of a programming language.

Developed in 1960s at Oxford University by Christopher Strachey's Programming Research Group, the methodology comprises notational elegance and mathematical rigor. Although initially designed as an analysis tool, denotational semantics has been used as a tool for implementation and language design.

Techopedia explains *Denotational Semantics*

In denotational semantics, the basic idea is mapping every syntactic entity associated with a programming language into some form of mathematical entity, translating programming language constructs into mathematical objects.

Denotational semantic definition has five parts:

- Semantic equations
- Syntactic categories
- Semantic functions
- Backus normal form (BNF) defining the structure of the syntactic categories
- Value domains

Denotational semantics have been developed for modern languages which have features like exceptions and concurrency. One of the important features of denotational semantics is that semantics should be compositional, meaning denotation of a programming phrase can be constructed from the denotations of its sub-phrases.

There are some distinct advantages associated with denotational semantics. It is the easiest mechanism for describing the meaning of smaller programs compared to other alternatives. Denotational semantics is capable of explaining state in programs. However, denotational semantics tend to be very complex for describing advanced features like goto statements and recursions.

9.284) attribute grammer – check Wikipedia. Some text are missing.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

```
E → E + T { E.value = E.value + T.value }
```

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

```
S → ABC
```

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

```
S → ABC
```

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion : When a non-terminal is expanded to terminals as per a grammatical rule

Attributes may be of two types – Synthesized or Inherited.

1. **Synthesized attributes –**
   A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. The attribute can take value only from its children (Variables in the RHS of the production).

   For eg. let's say A -> BC is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes then it will be synthesized attribute.
2. **Inherited attributes –**
   An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production).
   For example, let's say A -> BC is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be inherited attribute.

9.282) comparison between bnf and ebnf

BNF is the original, most simple, mostly used in academic papers of theoretical context, for communicating to humans. (as opposed to being used in compiler/parser.) There's no one exact specification of BNF.

EBNF means Extended BNF. There's not one single EBNF, but each author or program define their own variant that's slightly different.

ABNF (augmented BNF) is a rather very different format that BNF, but is more standardized. It is harder to read, but is most used in parsers.

In terms of power, they are all equivalent.

They are just syntactical differences. For example, in traditional BNF, the lhs/rhs separator is `::=`, but in books, often `→`. In EBNF and ABNF it's `=`.

Another example, in traditional BNF, nonterminals are written with brackets around it such as `<EXPR>` and terminals are just plain characters.

In ABNF, nonterminals are plain, and terminals are bracketed with double quotes, like this `"+"`.

In BNF, the symbol for alternatives is a vertical line `|`. In ABNF, the symbol for alternatives is a slash `/`.

EBNF and ABNF also features shortcut grammar syntax, such as specifying 0 or more of the preceding nonterminal/terminal. To translate it to BNF, you'll need to introduce several more rules and nonterminals.

In general, BNF notation is good for teaching, explanation, theoretical discussion. It is simple. EBNF and especially ABNF are more used to actually implement grammar and read by parsers.

Advantages of ebnf over BNF

Any grammar defined in EBNF can also be represented in BNF though representations in the latter are generally lengthier. e.g., options and repetitions cannot be directly expressed in BNF and require the use of an intermediate rule or alternative production defined to be either nothing or the optional production for option, or either the repeated production of itself, recursively, for repetition. The same constructs can still be used in EBNF.

The BNF uses the symbols (<, >, |, ::=) for itself, but does not include quotes around terminal strings. This prevents these characters from being used in the languages, and requires a special symbol for the empty string. In EBNF, terminals are strictly enclosed within quotation marks ("…" or '…'). The angle brackets ("<…>") for nonterminals can be omitted.

BNF syntax can only represent a rule in one line, whereas in EBNF a terminating character, the semicolon, marks the end of a rule.

Furthermore, EBNF includes mechanisms for enhancements, defining the number of repetitions, excluding alternatives, comments, etc.

9.280) ambiguity in grammer

Ambiguous grammar:

A Context Free Grammar (CFG) is said to ambiguous if there exists more than one derivation tree for the given input string i.e., more than one **L**eft**M**ost **D**erivation **T**ree (LMDT) or **R**ight**M**ost **D**erivation **T**ree (RMDT).

Definition: G = (V,T,P,S) is a CFG is said to be ambiguous if and only if there exist a string in T* that has more than on parse tree.
where V is a finite set of variables.
T is a finite set of terminals.
P is a finite set of productions of the form, A -> α, where A is a variable and α ∈ (V ∪ T)* S is a designated variable called the start symbol.

If the grammar has ambiguity, then it is not good for compiler construction. No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.

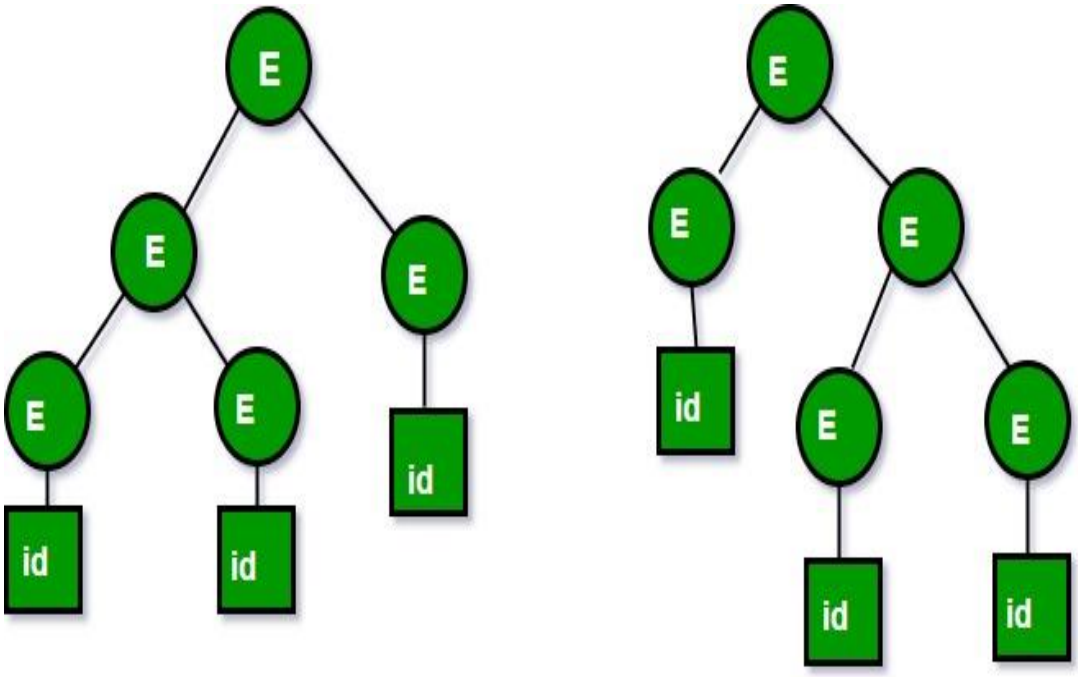
In computer science, an **ambiguous grammar** is a context-free grammar for which there exists a string that can have more than one leftmost derivation or parse tree,[1] while an **unambiguous**

grammar is a context-free grammar for which every valid string has a unique leftmost derivation or parse tree. Many languages admit both ambiguous and unambiguous grammars, while some languages admit only ambiguous grammars. Any non-empty language admits an ambiguous grammar by taking an unambiguous grammar and introducing a duplicate rule or synonym (the only language without ambiguous grammars is the empty language). A language that only admits ambiguous grammars is called an inherently ambiguous language, and there are inherently ambiguous context-free languages. Deterministic context-free grammars are always unambiguous, and are an important subclass of unambiguous grammars; there are non-deterministic unambiguous grammars, however.

For computer programming languages, the reference grammar is often ambiguous, due to issues such as the dangling else problem. If present, these ambiguities are generally resolved by adding precedence rules or other context-sensitive parsing rules, so the overall phrase grammar is unambiguous.[citation needed] The set of all parse trees for an ambiguous sentence is called a *parse forest*.[2]

For Example:

1. Let us consider this grammar : **E -> E+E|id**
We can create 2 parse tree from this grammar to obtain a string **id+id+id** :
The following are the 2 parse trees generated by left most derivation:



Both the above parse trees are derived from same grammar rules but both parse trees are different. Hence the grammar is ambiguous.

Disambiguate the grammar i.e., rewriting the grammar such that there is only one derivation or parse tree possible for a string of the language which the grammar represents.

9.279) utility of a parse tree

Parse trees are an in-memory representation of the input with a structure that conforms to the grammar.

The advantages of using parse trees instead of semantic actions:

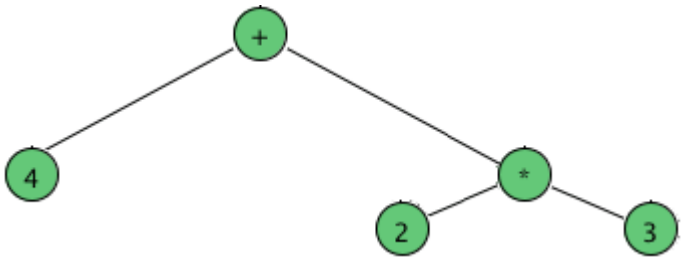
- You can make multiple passes over the data without having to re-parse the input.
- You can perform transformations on the tree.
- You can evaluate things in any order you want, whereas with attribute schemes you have to process in a begin to end fashion.
- You do not have to worry about backtracking and action side effects that may occur with an ambiguous grammar.

So, you like what you see, but maybe you think that the parse tree is too hard to process? With a few more directives you can generate an abstract syntax tree (ast) and cut the amount of evaluation code by at least **50%**. So without any delay, here's the ast calculator grammar:

Abstract Syntax Tree

A computer language is basically a context-free language. The symbols are tokens without any particular semantic meaning, namely, all numbers are the same, perhaps even all literal things (strings, numbers, etc) are regarded equally. All variables are regarded equally, etc. So the point is that we have a finite symbol set.

The first step of a compiler is to create a parse tree of the program, and the second phase is to assign meaning, or semantics to the entities in the tree. In reality, you create an *abstract syntax tree* of the the program. For example, considering the parse tree for `4 + 2 * 3` above, an abstract syntax tree for this expression would look like this:



10.288) what is axiomatic semantic.

Axiomatic semantics is an approach based on mathematical logic for proving the correctness of computer programs. It is closely related to Hoare logic. **Hoare logic** (also known as **Floyd–Hoare logic** or **Hoare rules**) is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs. It was proposed in 1969 by the British computer scientist and logician Tony Hoare, and subsequently refined by Hoare and other researchers.

Axiomatic semantics define the meaning of a command in a program by describing its effect on assertions about the program state. The assertions are logical statements—predicates with variables, where the variables define the state of the program.

Definition: Defining the behavior of an *abstract data type* with axioms.

Aggregate parent (I am a part of or used in ...)
stack, *bag*, *dictionary*, *priority queue*, *queue*, *set*, *cactus stack*.

Note: For example, the abstract data type stack has the operations new(), push(v, S) and popOff(S), among others. These may be defined with the following axioms.

1. *new() returns a stack*
2. *popOff(push(v, S)) = S*
3. *top(push(v, S)) = v*

where S is a stack and v is a value. What does this mean? The first axiom says all we know about new() is that it returns a stack. Informally, we know it returns an empty stack, but "empty" is a concept we would have to define. So we leave it.

The second axiom says that if we push a value onto a stack then pop it off, the result is the same stack. The "=" can be seen as a rewrite operation. The axiom "X = Y" means any time we see X, we can rewrite it to be Y. X may contain variables representing subexpressions. What is the meaning of "popOff(push(1776, new()))"? The second axiom says it means the same as new().

The third axiom assigns meaning to expressions like top(push(1, push(2, new()))): it is 1. This is reasonable, since the top element is the latest one pushed. A series of push and popOff operations and a top operation may be reduced with these axioms.

What stack does new() return, then? We still haven't said; top(new()) is just not defined. But that is how a stack works: the top of an empty stack is not defined. So our formalism corresponds to our mental notion of a stack. If we want to, we can add more axioms for richer semantics, as is done in the stack entry.

Definition - What does *Axiomatic Semantics* mean?

Axiomatic semantics are semantic expressions of the relationships inherent in a piece of code. These expressions can be helpful in describing how some piece of software works.

Techopedia explains *Axiomatic Semantics*

An interesting thing about axiomatic semantics as contrasted to other types of expressions is that they are fairly agnostic of specific results and conditions. Rather, axiomatic semantics describe the way that a system works. One way to think of this is using the root word, axiom, which implies some broader truism about a system. For example, an axiomatic semantical statement about a certain function would describe what it is meant to do, what sort of argument it takes, and what sort of result it returns. This would not require knowledge of the actual variables involved.

10.26) Data hierarchy and their mode of operation

Data are the principal resources of an organization. Data stored in computer systems form a hierarchy extending from a single bit to a database, the major record-keeping entity of a firm. Each higher rung of this hierarchy is organized from the components below it.

Data are logically organized into:

1. Bits (characters)

2. Fields

3. Records

4. Files

5. Databases

Bit (Character) - a bit is the smallest unit of data representation (value of a bit may be a 0 or 1). Eight bits make a byte which can represent a character or a special symbol in a character code.

Field - a field consists of a grouping of characters. A data field represents an attribute (a characteristic or quality) of some entity (object, person, place, or event).

Record - a record represents a collection of attributes that describe a real-world entity. A record consists of fields, with each field describing an attribute of the entity.

File - a group of related records. Files are frequently classified by the application for which they are primarily used (employee file). A ***primary key*** in a file is the field (or fields) whose value identifies a record among others in a data file.

Database - is an integrated collection of logically related records or files. A database consolidates records previously stored in separate files into a common pool of data records that provides data for many applications. The data is managed by systems software called database management systems (DBMS). The data stored in a database is independent of the application programs using it and of the types of secondary storage devices on which it is stored.

Data Hierarchy refers to the systematic organization of data, often in a hierarchical form. Data organization involves fields, records, files and so on. A data field holds a single fact or attribute of an entity. Consider a date field, e.g. "September 19, 2004". This can be treated as a single date field, or 3 fields, namely, month, day of month and year. A record is a collection of related fields. An Employee record may contain a name field, address fields, birthdate field and so on. A file is a collection of related records. If there are 100 employees, then each employee would have a record and the collection of 100 such records would constitute a file. Files are integrated into a database. This is done using a Database Management System. If there are other facets of employee data that we wish to capture, then other files such as Employee Training History file and Employee Work History file could be created as well. The above is a view of data seen by a computer user. The above structure can be seen in the hierarchical model, which is one way to organize data in a database

10.27 b) Hybrid compilers

Hybrid compiler is a compiler which translates a human readable source code to an intermediate byte code for later interpretation. So these languages do have both features of a compiler and an interpreter. These types of compilers are commonly known as Just In-time Compilers (JIT).

Java is one good example for these types of compilers.

10.30) modular programming

Modular Approach in Programming

Modular programming is the process of subdividing a computer program into separate sub-programs. A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system.

- Some programs might have thousands or millions of lines and to manage such programs it becomes quite difficult as there might be too many of syntax errors or logical errors present in the program, so to manage such type of programs concept of **modular programming** approached.
- Each sub-module contains something necessary to execute only one aspect of the desired functionality.
- Modular programming emphasis on breaking of large programs into small problems to increase the maintainability, readability of the code and to make the program handy to make any changes in future or to correct the errors.

Points which should be taken care of prior to modular program development:

1. Limitations of each and every module should be decided.
2. In which way a program is to be partitioned into different modules.
3. Communication among different modules of the code for proper execution of the entire program.

Advantages of Using Modular Programming Approach –

1. **Ease of Use :**This approach allows simplicity, as rather than focusing on the entire thousands and millions of lines code in one go we can access it in the form of modules. This allows ease in debugging the code and prone to less error.
2. **Reusability :**It allows the user to reuse the functionality with a different interface without typing the whole program again.
3. **Ease of Maintenance :** It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

Overview

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.[1]

Concept of Modularization

One of the most important concepts of programming is the ability to group some lines of code into a unit that can be included in our program. The original wording for this was a sub-program. Other names include: macro, sub-routine, procedure, module and function. We are going to use the term **function** for that is what they are called in most of the predominant programming languages of today. Functions are important because they allow us to take large complicated programs and to divide them into smaller manageable pieces. Because the function is a smaller piece of the overall program, we can concentrate on what we want it to do and test it to make sure it works properly. Generally, functions fall into two categories:

1. **Program Control** – Functions used to simply sub-divide and control the program. These functions are unique to the program being written. Other programs may use similar functions, maybe even functions with the same name, but the content of the functions are almost always very different.
2. **Specific Task** – Functions designed to be used with several programs. These functions perform a specific task and thus are usable in many different programs because the other programs also need to do the specific task. Specific task functions are sometimes referred to as building blocks. Because they are already coded and tested, we can use them with confidence to more efficiently write a large program.

10.51.ii) discuss importance of c language

Benefits of C

- As a middle level language, C combines the features of both high level and low level languages. It can be used for low-level programming, such as scripting for drivers and kernels and it also supports functions of high level programming languages, such as scripting for software applications etc.
- C is a structured programming language which allows a complex program to be broken into simpler programs called functions. It also allows free movement of data across these functions.
- C language is case-sensitive.
- C is highly portable and is used for scripting system applications which form a major part of Windows, UNIX and Linux operating system.
- C is a general purpose programming language and can efficiently work on enterprise applications, games, graphics, and applications requiring calculations.
- C language has a rich library which provides a number of built-in functions. It also offers dynamic memory allocation.

1.1. Building block for many other programming languages

C is considered to be the most fundamental language that needs to be studied if you are beginning with any programming language. Many programming languages such as Python, C++, Java, etc are built with the base of the C language.

1.2. Powerful and efficient language

C is a robust language as it contains many data types and operators to give you a vast platform to perform all kinds of operations.

Take a break & Learn Different Data Types in C

1.3. Portable language

C is very flexible, or we can say machine independent that helps you to run your code on any machine without making any change or just a few changes in the code.

1.4. Built-in functions

There are only 32 keywords in ANSI C, having many built-in functions. These functions are helpful when building a program in C.

1.5. Quality to extend itself

Another crucial ability of C is to extend itself. We have already studied that the C language has its own set of ***functions in the C*** library. So, it becomes easy to use these functions. We can add our own functions to the C Standard Library and make code simpler.

1.6. Structured programming language

C is structure-based. It means that the issues or complex problems are divided into smaller blocks or functions. This modular structure helps in easier and simpler testing and maintenance.

1.7. Middle-level language

C is a middle-level programming language that means it supports high-level programming as well as low-level programming. It supports the use of kernels and drivers in low-level programming and also supports system software applications in the high-level programming language.

1.8. Implementation of algorithms and data structures

The use of algorithms and data structures in C has made program computations very fast and smooth. Thus, the C language can be used in complex calculations and operations such as MATLAB.

1.9. Procedural programming language

C follows a proper procedure for its functions and subroutines. As it uses procedural programming, it becomes easier for C to identify code structure and to solve any problem in a specific series of code. In procedural programming ***C variables*** and functions are declared before use.

1.10. Dynamic memory allocation

C provides dynamic memory allocation that means you are free to allocate memory at run time. For example, if you don't know how much memory is required by objects in your program, you can still run a program in C and assign the memory at the same time.

Learn the Basic Structure of C Program in 7 Mins

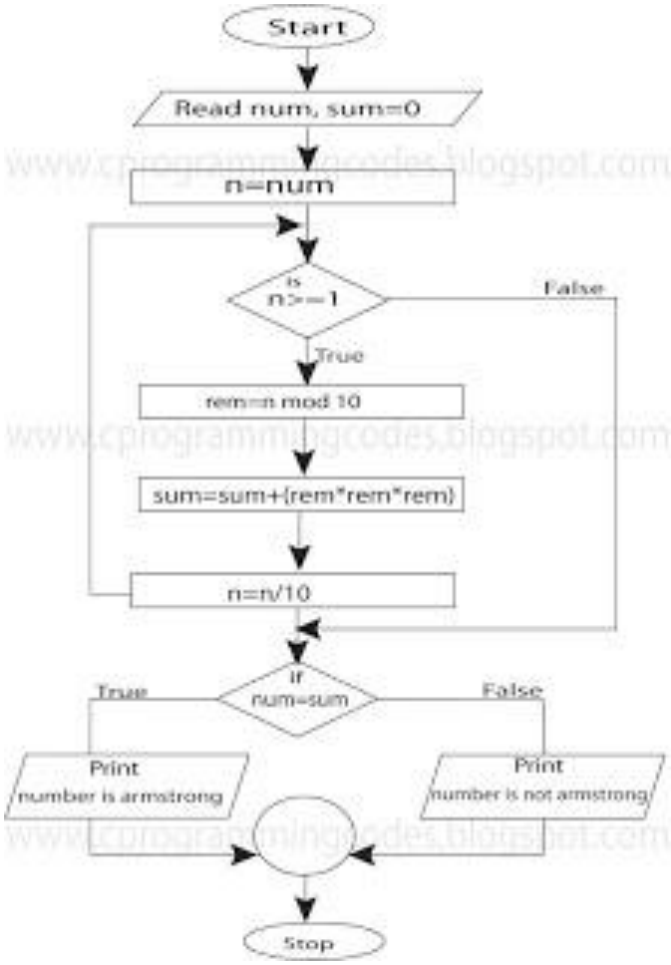
1.11. System-programming

C follows a system based programming system. It means the programming is done for the hardware devices.

So, with this, we are aware of *why C considered a very powerful language and why is it important to know the advantages of C?*

When we study anything new, it becomes important to know the benefits that we gain from that technology. This allows us to grow our interest and implement our knowledge in a practical scenario. Now, let us move on to the “Advantages and Disadvantages of the C Programming Language”.

10.53 i) algorithm and flowchart for Armstrong number




```
Step 1: Start
Step 2: Declare Variable
sum, temp, num
Step 3: Read num from User
Step 4: Initialize Variable
sum=0 and temp=num
Step 5: Repeat Until num>=0
            5.1 sum=sum +
cube of last digit i.e
[(num%10)*(num%10)*(num%10)]
            5.2 num=num/10
Step 6: IF sum==temp
            Print "Armstrong
Number"
        ELSE
            Print "Not
Armstrong Number"
Step 7: Stop
```

10.54.ii) A typical programming task is divided into problem solving and implementation phase – justify

Problem Solving Phase-
Analysis and Specification
Understand (define) the problem and what the solution must do

Algorithm Development
Develop a comprehensive unambiguous logical sequence of steps to solve the problem

Verification of Algorithm
Follow steps closely (manually) to see if solution works

Implementation Phase-
Program Development

Translate algorithm into a program written in a programming language

Program Testing
Test program for syntactical and logical errors. Fix the errors.

Maintenance Phase-
Use
Use the program to solve real world problems

Maintain
Modify the program to meet changing requirements
----------OR----------

ALGORITHMS AND FLOWCHARTS. A typical programming task can be divided into two phases: Problem solving phase ♦ produce an ordered sequence of steps that."— Presentation transcript:

ALGORITHMS AND FLOWCHARTS

A typical programming task can be divided into two phases: Problem solving phase ♦ produce an ordered sequence of steps that describe solution of problem ♦ this sequence of steps is called an algorithm Implementation phase ♦ implement the program in some programming language

Steps in Problem Solving First produce a general algorithm (one can use pseudocode) Refine the algorithm successively to get step by step detailed algorithm that is very close to a computer language. Pseudocode is an artificial and informal language that helps programmers develop algorithms. Pseudocode is very similar to everyday English.

Pseudocode & Algorithm Example 1: Write an algorithm to determine a student's final grade and indicate whether it is passing or failing. The final grade is calculated as the average of four marks.

The Flowchart (Dictionary) A schematic representation of a sequence of operations, as in a manufacturing process or computer program. (Technical) A graphical representation of the sequence of operations in an information system or program. Information system flowcharts show how data flows from source documents through the computer to final distribution to users. Program flowcharts show the sequence of instructions in a single program or subroutine. Different symbols are used to draw each type of flowchart.

The Flowchart A Flowchart ♦ shows logic of an algorithm ♦ emphasizes individual steps and their interconnections ♦ e.g. control flow from one action to the next

11.78 i) Difference between int and unsigned int

What is a difference between unsigned int and signed int in C?

The signed and unsigned integer type has the same **storage** (according to the standard at least 16 bits) and alignment but still, there is a lot of difference them, in bellows lines, I am describing some difference between the signed and unsigned integer.

- A signed integer can store the positive and negative value both but beside it unsigned integer can only store the positive value.
- The range of nonnegative values of a signed integer type is a sub-range of the corresponding unsigned integer type.
  For example,

Assuming size of the integer is 2 bytes.

signed int -32768 to +32767

unsigned int 0 to 65535

- When computing the unsigned integer, it never gets overflow because if the computation result is greater than the largest value of the unsigned integer type, it is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

11.78 ii) requirements of int and long int

Refer to question 11.76 mentioned waaaayyy earlier. Ty!

11.78 iii) difference between float and double data type

Float	Double
This is generally used for graphic based libraries for making the processing power of your programs faster, as it is simpler to manage by compilers.	This is the most commonly used data type in programming languages for assigning values having a real or decimal based number within, such as 3.14 for pi.
It has single precision.	It has the double precision or you can say two times more precision than float.
According to IEEE, it has a 32-bit floating point precision.	According to IEEE, it has a 64-bit floating point precision.
Float takes 4 bytes for storage.	Double takes 8 bytes for storage.
A value having a range within 1.2E-38 to 3.4E+38 can be assigned to float variables.	A value having range within 2.3E-308 to 1.7E+308 can be assigned to double type variables
Has a precision of 6 decimal places.	Has a precision of 15 decimal places.

11.80) isdigit(), islower(),isupper(),toupper(),tolower() write short note.

C isdigit()

The isdigit() function checks whether a character is numeric character (0-9) or not.

Function Prototype of isdigit()

```
int isdigit( int arg );
```

Function isdigit() takes a single argument in the form of an integer and returns the value of type `int`.

Even though, isdigit() takes integer as an argument, character is passed to the function. Internally, the character is converted to its ASCII value for the check.

It is defined in <ctype.h> header file.

C isdigit() Return value

Return Value	Remarks
Non-zero integer (x > 0)	Argument is a numeric character.
Zero (0)	Argument is not a numeric character.

Example: C isdigit() function

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

int main()
{
    char c;
    c='5';
    printf("Result when numeric character is passed: %d", isdigit(c));

    c='+';
    printf("\nResult when non-numeric character is passed: %d",
isdigit(c));
```

```
12.
13.     return 0;
14. }
```

Output

```
Result when numeric character is passed: 1
Result when non-numeric character is passed: 0
```

C islower()

The islower() function checks whether a character is lowercase alphabet (a-z) or not.

Function Prototype of islower()

```
int islower( int arg );
```

Function islower() takes a single argument in the form of an integer and returns a value of type `int`.

Even though islower() takes integer as an argument, character is passed to the function. Internally, the character is converted to its ASCII value for the check.

It is defined in <ctype.h> header file.

C islower() Return Value

Return Value	Remarks
Non-zero number (x > 0)	Argument is a lowercase alphabet.
Zero (0)	Argument is not a lowercase alphabet.

C isupper()

The isupper() function checks whether a character is an uppercase alphabet (A-Z) or not.

C isupper() Prototype

```
int isupper(int argument);
```

Function isupper() takes a single argument in the form of an integer and returns a value of type `int`.

Even though, isupper() takes integer as an argument, character is passed to the function. Internally, the character is converted to its ASCII for the check.

It is defined in <ctype.h> header file.

C isupper() Return Value

Return Value	Remarks
Non-zero integer (x > 0)	Argument is an uppercase alphabet.
Zero (0)	Argument is not an uppercase alphabet.

C toupper()

The toupper() function converts a lowercase alphabet to an uppercase alphabet, if the argument passed is an lowercase alphabet.

C toupper() Prototype

```
int toupper( int arg );
```

Function toupper() takes a single argument in the integer form and returns a value of type `int`.

Even though, toupper() takes integer as an argument, character is passed to the function. Internally, the character is converted to its corresponding ASCII value for the check.

If the argument passed is other than a lowercase alphabet, it returns the same character passed to the function.

It is defined in <ctype.h> header file.

```
1. #include <stdio.h>
2. #include <ctype.h>
3. int main()
4. {
5.     char c;
6.
```

```
    c = 'm';
    printf("%c -> %c", c, toupper(c));

    // Displays the same argument passed if other characters than lowercase character is passed to toupper().
    c = 'D';
    printf("\n%c -> %c", c, toupper(c));

    c = '9';
    printf("\n%c -> %c", c, toupper(c));
    return 0;
}
```

Output

```
m -> M
D -> D
9 -> 9
```

C tolower()

The tolower() function takes an uppercase alphabet and convert it to a lowercase character.

If the arguments passed to the tolower() function is other than an uppercase alphabet, it returns the same character that is passed to the function.

It is defined in ctype.h header file.

Function Prototype of tolower()

```
int tolower(int argument);
```

The character is stored in integer form in C programming. When a character is passed as an argument, corresponding ASCII value (integer) of the character is passed instead of that character itself.

```
#include <stdio.h>
#include <ctype.h>
int main()
{
    char c, result;

    c = 'M';
    result = tolower(c);
    printf("tolower(%c) = %c\n", c, result);

    c = 'm';
```

```
    result = tolower(c);
    printf("tolower(%c) = %c\n", c, result);

    c = '+';
    result = tolower(c);
    printf("tolower(%c) = %c\n", c, result);

    return 0;
}
```

Output

```
tolower(M) = m
tolower(m) = m
tolower(+) = +
```

11.81 i and ii) discuss printf and scanf and their return types– done previously .. 5.77 check

11.82 i) discuss the difference between %s and %c specifier in printf and scanf in c – 5.77 check

11.82 ii) short note on escape sequences in c

An escape sequence in C language is a sequence of characters that doesn't represent itself when used inside string literal or character.

It is composed of two or more characters starting with backslash \. For example: \n represents new line.

List of Escape Sequences in C

Escape Sequence	Meaning
\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	Single Quote

\"	Double Quote
\?	Question Mark
\nnn	octal number
\xhh	hexadecimal number
\0	Null

Escape Sequence Example

```
1. #include<stdio.h>
2. int main(){
3.     int number=50;
4.    printf("You\nare\nlearning\n\'c\' language\n\"Do you know C language\"");
5. return 0;
6. }
```

Output:

```
You
are
learning
'c' language
"Do you know C language"
```

11.108 i) odd even using bitwise operator

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

int isOdd (int n)
{
  if (n & 1)
    return 1;
  else
    return 0;
}

int main(void)
{
  unsigned int n;
  printf("Enter a positive integer: ");
  scanf("%u", &n);
```

```
  if (isOdd(n))
    printf("%d is odd", n);
  else
    printf("%d is even", n);
}

OUTPUT
======
Enter a positive integer: 11
11 is odd
```

11.109 i) What is an expression?

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in the table given below.

Algebraic Expression	**C Expression**
a x b – c	a * b – c
(m + n) (x + y)	(m + n) * (x + y)
(ab / c)	a * b / c
$3x^2$ +2x + 1	3*x*x+2*x+1
(x / y) + c	x / y + c

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form
`Variable = expression;`
Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted.
Example of evaluation statements are

x = a * b – c
y = b / c * a
z = a – b / c + d;

The following program illustrates the effect of presence of parenthesis in expressions.

```
main ()
{
    float a, b, c x, y, z;
```

```
    a = 9;
            b = 12;
            c = 3;
            x = a - b / 3 + c * 2 - 1;
            y = a - b / (3 + c) * (2 - 1);
            z = a - ( b / (3 + c) * 2) - 1;
            printf ("x = %f",x);
            printf ("y = %f",y);
            printf ("z = %f",z);

}
```

output

```
x = 10.00
y = 7.00
z = 4.00
```

Precedence in Arithmetic Operators

An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C.

```
High priority * / %
Low priority + -
```

Rules for evaluation of expression

• First parenthesized sub expression left to right are evaluated.
• If parenthesis are nested, the evaluation begins with the innermost sub expression.
• The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
• The associability rule is applied when two or more operators of the same precedence level appear in the sub expression.
• Arithmetic expressions are evaluated from left to right using the rules of precedence.
• When Parenthesis are used, the expressions within parenthesis assume highest priority.

11.109 iii) What is mixed mode operation and automatic conversion?

If operands in an expression contains both **INTEGER** and **REAL** constants or variables, this is a *mixed mode arithmetic expression*.

In mixed mode arithmetic expressions, **INTEGER** operands are *always* converted to **REAL** before carrying out any computations. As a result, the result of a mixed mode expression is of **REAL** type. The following is a table showing this fact.

Operator	INTEGER	REAL
INTEGER	**INTEGER**	**REAL**
REAL	**REAL**	**REAL**

The rules for evaluating mixed mode arithmetic expressions are simple:

- Use the rules for evaluating *single mode arithmetic expressions* for scanning.
- After locating an operator for evaluation, do the following:
  - if the operands of this operator are of the same type, compute the result of this operator.
  - otherwise, one of the operand is an integer while the other is a real number. In this case, convert the integer to a real (*i.e.*, adding **.0** at the end of the integer operand) and compute the result. Note that since both operands are real numbers, the result is a real number.
- **There is an exception, though.** In **a**n**, where **a** is a real and **n** is a positive integer, the result is computed by multiplying **n** copies of **a**. For example, **3.5**3** is computed as **3.5*3.5*3.5**

Simple Examples:

- **1 + 2.5** is **3.5**
- **1/2.0** is **0.5**
- **2.0/8** is **0.25**
- **-3**2.0** is **-9.0**
- **4.0**(1/2)** is first converted to **4.0**0** since **1/2** is a single mode expression whose result is **0**. Then, **4.0**0** is **1.0**

In C we can have expressions consisting of constants and variables of different data types.

Automatic conversions-

Types of conversions

There are two type of conversions in C.

- Implicit type conversion
- Explicit type conversion

Implicit type conversion

C performs automatic conversions of type in order to evaluate the expression. This is called implicit type conversion.

For example, if we have an integer data type value and a double data type value in an expression then C will automatically convert integer type value to double in order to evaluate the expression.

Rules for implicit type conversion

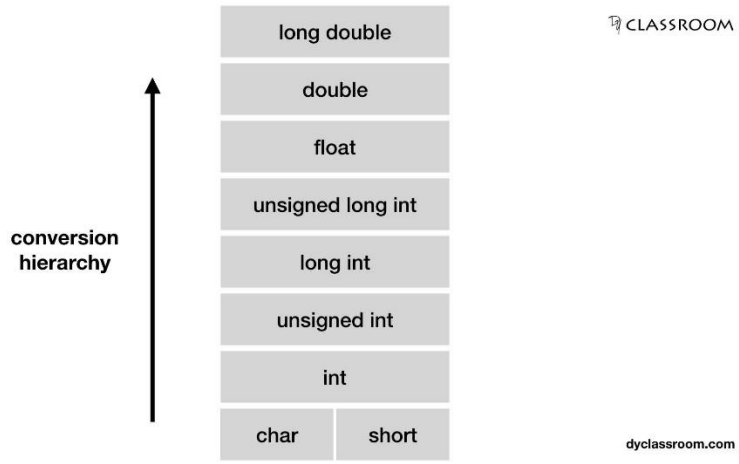
Following are the rules for the implicit type conversion in C.

First, all `char` and `short` are converted to `int` data type.

Then,

- If any of the operand in the expression is `long double` then others will be converted to `long double` and we will get the result in `long double`.
- Else, if any of the operand is `double` then other will be converted into `double` and the result will be in `double`.
- Else, if any of the operand is `float` then other will be converted into `float` and the result will be in `float`.
- Else, if any of the operand is `unsigned long int` then others will be converted into `unsigned long int` and we will get the result in `unsigned long int`.
- Else, if any of the operand is `long int` and another is in `unsigned int` then,
    - If `unsigned int` can be converted to `long int` then it will be converted into `long int` and the result will be in `long int`.
    - Else, both will be converted into `unsigned long int` and the result will be in `unsigned long int`.
- Else, if any of the operand is `long int` then other will be converted to `long int` and we will get the result in `long int`.
- Else, if any of the operand is `unsigned int` then other will be converted into `unsigned int` and the result will be in `unsigned int`.

Remember the following hierarchy ladder of implicit type conversion.



If we downgrade from a higher data type to a lower data type then it causes lose of bits.

For example: Moving from `double` to `float` causes rounding of digits.

Downgrading from `float` to `int` causes truncation of the fractional part.

Explicit type conversion

In explicit type conversion we decide what type we want to convert the expression.

Syntax of explicit type conversion is:

```
(type) expression
```

Where, **type** is any of the type we want to convert the **expression** into.

In the following example we are converting floating point numbers into integer.

```
#include <stdio.h>

int main(void)
```

```
{

  //variables

  float

    x = 24.5,

    y = 7.2;



  //converting float to int

  int result = (int) x / (int) y;



  //output

  printf("Result = %d\n", result);



  printf("End of code\n");

  return 0;

}
```

Output

```
Result = 3

End of code
```

In the above code `(int) x` converts the value 24.5 into 24 and `(int) y` converts the value 7.2 into 7 so, we get 24/7 i.e., 3 as result because `result` is of type int and hence the decimal part is truncated.

11.109.ii) operators in c - same for 11.110

C Programming Operators

In this tutorial, you will learn about different operators in C programming with the help of examples.

An operator is a symbol that operates on a value or a variable. For example: `+` is an operator to perform addition.

C has a wide range of operators to perform various operations.

C Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

Example 1: Arithmetic Operators

```c
// Working of arithmetic operators
#include <stdio.h>
int main()
{
    int a = 9,b = 4, c;

    c = a+b;
    printf("a+b = %d \n",c);
    c = a-b;
    printf("a-b = %d \n",c);
    c = a*b;
    printf("a*b = %d \n",c);
    c = a/b;
    printf("a/b = %d \n",c);
    c = a%b;
    printf("Remainder when a divided by b = %d \n",c);

    return 0;
}
```

Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

The operators `+`, `-` and `*` computes addition, subtraction, and multiplication respectively as you might have expected.
In normal calculation, `9/4 = 2.25`. However, the output is `2` in the program.
It is because both the variables `a` and b are integers. Hence, the output is also an integer. The compiler neglects the term after the decimal point and shows answer `2` instead of `2.25`.
The modulo operator `%` computes the remainder. When `a=9` is divided by `b=4`, the remainder is `1`. The `%` operator can only be used with integers.
Suppose `a = 5.0`, `b = 2.0`, `c = 5` and `d = 2`. Then in C programming,

```
// Either one of the operands is a floating-point number

a/b = 2.5

a/d = 2.5

c/b = 2.5




// Both operands are integers

c/d = 2
```

C Increment and Decrement Operators

C programming has two operators increment `++` and decrement `--` to change the value of an operand (constant or variable) by 1.
Increment `++` increases the value by 1 whereas decrement `--` decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example 2: Increment and Decrement Operators

```c
// Working of increment and decrement operators
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;
```

```c

    printf("++a = %d \n", ++a);
    printf("--b = %d \n", --b);
    printf("++c = %f \n", ++c);
    printf("--d = %f \n", --d);

    return 0;
}
```

Output

```
++a = 11
--b = 99
++c = 11.500000
++d = 99.500000
```

Here, the operators `++` and `--` are used as prefixes. These two operators can also be used as postfixes like `a++` and `a--`. Visit this page to learn more about how increment and decrement operators work when used as postfix.

C Assignment Operators

An assignment operator is used for assigning a value to a variable. The most common assignment operator is `=`

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example 3: Assignment Operators

```c
// Working of assignment operators
#include <stdio.h>
int main()
{
    int a = 5, c;
```

```
6.
7.     c = a;      // c is 5
8.     printf("c = %d\n", c);
9.     c += a;     // c is 10
10.    printf("c = %d\n", c);
11.    c -= a;     // c is 5
12.    printf("c = %d\n", c);
13.    c *= a;     // c is 25
14.    printf("c = %d\n", c);
15.    c /= a;     // c is 5
16.    printf("c = %d\n", c);
17.    c %= a;     // c = 0
18.    printf("c = %d\n", c);
19.
20.    return 0;
21. }
```

Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	`5 == 3` is evaluated to 0
>	Greater than	`5 > 3` is evaluated to 1
<	Less than	`5 < 3` is evaluated to 0
!=	Not equal to	`5 != 3` is evaluated to 1
>=	Greater than or equal to	`5 >= 3` is evaluated to 1

Operator	Meaning of Operator	Example
<=	Less than or equal to	`5 <= 3` is evaluated to 0

Example 4: Relational Operators

```
// Working of relational operators
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;

    printf("%d == %d is %d \n", a, b, a == b);
    printf("%d == %d is %d \n", a, c, a == c);
    printf("%d > %d is %d \n", a, b, a > b);
    printf("%d > %d is %d \n", a, c, a > c);
    printf("%d < %d is %d \n", a, b, a < b);
    printf("%d < %d is %d \n", a, c, a < c);
    printf("%d != %d is %d \n", a, b, a != b);
    printf("%d != %d is %d \n", a, c, a != c);
    printf("%d >= %d is %d \n", a, b, a >= b);
    printf("%d >= %d is %d \n", a, c, a >= c);
    printf("%d <= %d is %d \n", a, b, a <= b);
    printf("%d <= %d is %d \n", a, c, a <= c);

    return 0;
}
```

Output

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression `((c==5) && (d>5))` equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression `((c==5) || (d>5))` equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression `!(c==5)` equals to 0.

Example 5: Logical Operators

```
// Working of logical operators

#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;

    result = (a == b) && (c > b);
    printf("(a == b) && (c > b) is %d \n", result);
    result = (a == b) && (c < b);
    printf("(a == b) && (c < b) is %d \n", result);
    result = (a == b) || (c < b);
    printf("(a == b) || (c < b) is %d \n", result);
    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) is %d \n", result);
    result = !(a != b);
    printf("!(a == b) is %d \n", result);
    result = !(a == b);
    printf("!(a == b) is %d \n", result);

    return 0;
}
```

Output

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

Explanation of logical operator program

- `(a == b) && (c > 5)` evaluates to 1 because both operands `(a == b)` and `(c > b)` is 1 (true).
- `(a == b) && (c < b)` evaluates to 0 because operand `(c < b)` is 0 (false).

- `(a == b) || (c < b)` evaluates to 1 because `(a = b)` is 1 (true).
- `(a != b) || (c < b)` evaluates to 0 because both operand `(a != b)` and `(c < b)` are 0 (false).
- `!(a != b)` evaluates to 1 because operand `(a != b)` is 0 (false). Hence, !(a != b) is 1 (true).
- `!(a == b)` evaluates to 0 because `(a == b)` is 1 (true). Hence, `!(a == b)` is 0 (false).

C Bitwise Operators

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Visit bitwise operator in C to learn more.

Other Operators

Comma Operator

Comma operators are used to link related expressions together. For example:

```
1. int a, c = 5, d;
```

The sizeof operator

The `sizeof` is a unary operator that returns the size of data (constants, variables, array, structure, etc).

Example 6: sizeof Operator

```c
#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;
}
```

Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
```

Other operators such as ternary operator `?:`, reference operator `&`, dereference operator `*` and member selection operator `->` will be discussed in later tutorials.

11.110) same as 11.109

11.111) explain bodmas rule in c language

Explain the rules for evaluation of expression in c

In the C programming language, an expression is evaluated based on the operator precedence and associativity. When there are multiple operators in an expression, they are evaluated according to their precedence and associativity. The operator with higher precedence is evaluated first and the operator with the least precedence is evaluated last.

Types of Expression Evaluation in C

Based on the operators and operators used in the expression, they are divided into several types. Types of Expression Evaluation in C are:

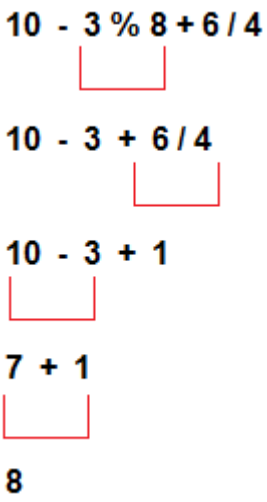
- **Integer expressions** – expressions which contains integers and operators
- **Real expressions** – expressions which contains floating point values and operators
- **Arithmetic expressions** – expressions which contain operands and arithmetic operators
- **Mixed mode arithmetic expressions** – expressions which contain both integer and real operands
- **Relational expressions** – expressions which contain relational operators and operands
- **Logical expressions** – expressions which contain logical operators and operands
- **Assignment expressions and so on…** – expressions which contain assignment operators and operands

Associativity

It represents which operator should be evaluated first if an expression is containing more than one operator with same priority.

Operator	Priority	Associativity
{}, (), []	1	Left to right
++, --, !	2	Right to left
*, /, %	3	Left to right
+, -	4	Left to right
<, <=, >, >=, ==, !=	5	Left to right
&&	6	Left to right
||	7	Left to right
?:	8	Right to left

=, +=, -=, *=, /=, %=	9	Right to left



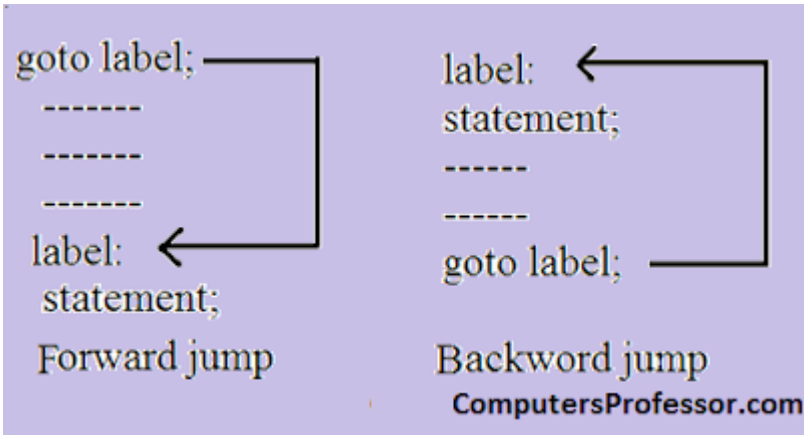
12.117 ii) explain forward and backward jump using goto statement in c

C supports the goto statement to branch unconditionally from one point to another in the program.

The goto requires a label inorder to identify the place where the branch is to be made. A label is any valid variable name, & must be followed by a colon.

The label is placed immediately before the statement where the control is to be transferred. The general forms of goto & label statements are

Syntax:



The label cab be anywhere in the program either before or after the goto label; statement.

If the label: is before the statement goto label; a loop will be formed & some statements will be executed repeatedly. Such a jump is known as a backward jump.
On the other hand, if the label: is placed after the goto label; some statements will be skipped & the jump is known as forward jump.

12.199) discuss briefly string functions – check geeksforgeeks

String functions	Description
strcat ( )	Concatenates str2 at the end of str1
strncat ( )	Appends a portion of string to another
strcpy ( )	Copies str2 into str1
strncpy ( )	Copies given number of characters of one string to another
strlen ( )	Gives the length of str1
strcmp ( )	Returns 0 if str1 is same as str2. Returns <0 if strl < str2. Returns >0 if str1 > str2
strcmpi ( )	Same as strcmp() function. But, this function negotiates case. "A" and "a" are treated as same.
strchr ( )	Returns pointer to first occurrence of char in str1
strrchr ( )	last occurrence of given character in a string is found
strstr ( )	Returns pointer to first occurrence of str2 in str1
strrstr ( )	Returns pointer to last occurrence of str2 in str1
strdup ( )	Duplicates the string
strlwr ( )	Converts string to lowercase
strupr ( )	Converts string to uppercase
strrev ( )	Reverses the given string

strset ( )	Sets all character in a string to given character
strnset ( )	It sets the portion of characters in a string to given character
strtok ( )	Tokenizing given string using delimiter

Strings in C: Strings are defined as an array of characters. The difference between a character array and a string is the string is terminated with a special character '\0'.
Some of the most commonly used String functions are:

- **strcat**: The strcat() function will append a copy of the source string to the end of destination string. The strcat() function takes two arguments:
  1) dest
  2) src
  It will append copy of the source string in the destination string. **The terminating character at the end of dest is replaced by the first character of src .**
  Return value: The strcat() function returns dest, the pointer to the destination string.

filter_none
edit
play_arrow
brightness_4

```
// CPP program to demonstrate
// strcat
#include <cstring>
#include <iostream>
using namespace std;
int main()
{
    char dest[50] = "This is an";
    char src[50] = " example";

    strcat(dest, src);
    cout << dest;
    return 0;
}
```

Output:

```
This is an example
```

- **strrchr**: In C/C++, strrchr() is a predefined function used for string handling. cstring is the header file required for string functions.
  This function Returns a pointer to the last occurrence of a character in a string. The character whose last occurrence we want to find in passed as the second argument to the function and the string in which we have to find the character is passed as the first argument to the function.
  Syntax

```
char *strrchr(const char *str, int c)
```

  Here, str is the string and c is the character to be located. It is passed as its int promotion, but it is internally converted back to char.

  Example:

filter_none

edit
play_arrow
brightness_4

```
// C code to demonstrate the working of
// strrchr()

#include <stdio.h>
#include <string.h>

// Driver function
int main()
{

    // initializing variables
    char st[] = "GeeksforGeeks";
    char ch = 'e';
    char* val;

    // Use of strrchr()
    // returns "ks"
    val = strrchr(st, ch);

    printf("String after last %c is :  %s \n",
           ch, val);
```

```
    char ch2 = 'm';


    // Use of strrchr()
    // returns null
    // test for null
    val = strrchr(st, ch2);


    printf("String after last %c is :  %s ",
           ch2, val);


    return (0);
}
```

Output:

```
String after last e is :  eks

String after last m is :  (null)
```

- **strcmp**: strcmp() is a built-in library function and is declared in **<string.h>** header file. This function takes two strings as arguments and compare these two strings lexicographically.
  Syntax::

```
int strcmp(const char *leftStr, const char *rightStr );
```

In the above prototype, function srtcmp takes two strings as parameters and returns an integer value based on the comparison of strings.

- strcmp() compares the **two strings lexicographically** means it starts comparison character by character starting from the first character until the characters in both strings are equal or a NULL character is encountered.
- If first character in both strings are equal, then this function will check the second character, if this is also equal then it will check the third and so on
- This process will be continued until a character in either string is NULL or the characters are unequal.

filter_none

edit
play_arrow
brightness_4

```
// C program to illustrate

// strcmp() function

#include <stdio.h>
```

```
#include <string.h>


int main()
{


    char leftStr[] = "g f g";
    char rightStr[] = "g f g";


    // Using strcmp()
    int res = strcmp(leftStr, rightStr);


    if (res == 0)
        printf("Strings are equal");
    else
        printf("Strings are unequal");


    printf("\nValue returned by strcmp() is:  %d", res);
    return 0;
}
```

Output:

```
Strings are equal
Value returned by strcmp() is:  0
```

- strcpy: strcpy() is a standard library function in C/C++ and is used to copy one string to another. In C it is present in **string.h** header file and in C++ it is present in **cstring** header file.

Syntax:

```
char* strcpy(char* dest, const char* src);
```

Paramters: This method accepts following paramters:

- **dest**: Pointer to the destination array where the content is to be copied.
- **src:** string which will be copied.

Return Value: After copying the source string to the destination string, the strcpy() function returns a pointer to the destination string.

Below program explains different usages of this library function:

filter_none

edit
play_arrow
brightness_4

```
// C program to illustrate

// strcpy() function ic C/C++

#include <stdio.h>

#include <string.h>


int main()

{

    char str1[] = "Hello Geeks!";

    char str2[] = "GeeksforGeeks";

    char str3[40];

    char str4[40];

    char str5[] = "GfG";


    strcpy(str2, str1);

    strcpy(str3, "Copy successful");

    strcpy(str4, str5);

    printf("str1: %s\nstr2: %s\nstr3: %s\nstr4: %s\n",

           str1, str2, str3, str4);

    return 0;

}
```

Output:

```
str1: Hello Geeks!

str2: Hello Geeks!

str3: Copy successful

str4: GfG
```

- strlen: The **strlen()** function calculates the length of a given string.The **strlen()** function is defined in **string.h** header file. It doesn't count null character '\0'.

Syntax:

```
int strlen(const char *str);
```

Parameter:

- **str:** It represents the string variable whose length we have to find.

Return: This function returns the length of string passed.

Below programs illustrate the strlen() function in C:

Example 1:-

filter_none

edit

play_arrow

brightness_4

```
// c program to demonstrate

// example of strlen() function.


#include <stdio.h>

#include <string.h>


int main()

{

    char ch[] = { 'g', 'e', 'e',

                  'k', 's', '\0' };


    printf("Length of string is: %lu",

           strlen(ch));


    return 0;

}
```

Output:

```
Length of string is: 5
```

- **strncat**: In C/C++, strncat() is a predefined function used for string handling. **string.h** is the header file required for string functions.
This function appends not more than **n** characters from the string pointed to by src to the end of the string pointed to by dest plus a terminating Null-character. The initial character of string(src) overwrites the Null-character present at the end of string(dest). Thus, length of the string(dest) becomes strlen(dest)+n. But, if the length of the string(src) is less than **n**, only the content up to the terminating null-character is copied and length of the string(dest) becomes strlen(src) + strlen(dest).

  The behavior is undefined if

  - the strings overlap.
  - the dest array is not large enough to append the contents of src.

Syntax:

```
char *strncat(char *dest, const char *src, size_t n)
```

Parameters: This method accepts following paramters:

- **dest**: the string where we want to append.
- **src**: the string from which 'n' characters are going to append.
- **n**: represents maximum number of character to be appended. size_t is an unsigned integral type.

Return Value: The strncat() function shall return the pointer to the string(dest).

Program:

filter_none

edit

play_arrow

brightness_4

```
// C, C++ program demonstrate functionality of strncat()

#include <stdio.h>

#include <string.h>


int main()

{


    // Take any two strings

    char src[50] = "efghijkl";

    char dest[50] = "abcd";


    // Appends 5 character from src to dest

    strncat(dest, src, 5);


    // Prints the string

    printf("Source string : %s\n", src);

    printf("Destination string : %s", dest);


    return 0;

}
```

Output:

```
Source string : efghijkl

Destination string : abcdefghi
```

- **strncmp**: **std::strncmp()** function lexicographically compares not more than count characters from the two null-terminated strings and returns an integer based on the outcome.
  - This function takes two strings and a number **num** as arguments and compare at most first **num** bytes of both the strings.
  - **num** should be at most equal to the length of the longest string. If **num** is defined greater than the string length than comparison is done till the null-character('\0') of either string.
  - This function compares the two strings lexicographically. It starts comparison from the first character of each string. If they are equal to each other, it continues and compare the next character of each string and so on.
  - This process of comparison stops until a terminating null-character of either string is reached or **num** characters of both the strings matches.

Syntax :

```
int strncmp(const char *str1, const char *str2, size_t count);

Parameters:
str1 and str2: C string to be compared.
count: Maximum number of characters to compare.
 size_t is an unsigned integral type.

Return Value:
Value                                   Meaning
Less than zero                          str1 is less than str2.
Zero                                    str1 is equal to str2.
Greater than zero                       str1 is greater than str2.
```

Example:

filter_none

edit
play_arrow
brightness_4

```
// C, C++ program to demonstrate

// functionality of strncmp()


#include <stdio.h>

#include <string.h>


int main()

{

    // Take any two strings

    char str1[10] = "aksh";

    char str2[10] = "akash";
```

```
    // Compare strings using strncmp()
    int result = strncmp(str1, str2, 4);

    if (result == 0) {
        // num is the 3rd parameter
        // of strncmp() function
        printf("str1 is equal to str2 upto num characters\n");
    }
    else if (result > 0)
        printf("str1 is greater than str2\n");
    else
        printf("str2 is greater than str1\n");

    printf("Value returned by strncmp() is: %d",
           result);

    return 0;
}
```

Output:

```
str1 is greater than str2
Value returned by strncmp() is: 18
```

- **strncpy**: The strncpy() function is similar to strcpy() function, except that at most n bytes of src are copied. If there is no NULL character among the first n character of src, the string placed in dest will not be NULL-terminated. If the length of src is less than n, strncpy() writes additional NULL character to dest to ensure that a total of n character are written.
  Syntax:

```
char *strncpy( char *dest, const char *src, size_t n )
```

Parameters: This function accepts two parameters as mentioned above and described below:

- **src:** The string which will be copied.
- **dest:** Pointer to the destination array where the content is to be copied.
- **n:** The first n character copied from src to dest.

Return Value: It returns a pointer to the destination string.
Example:

edit
play_arrow
brightness_4

```
// C Program  to illustrate the

// strcpy() function in C/C++

#include <stdio.h>

#include <string.h>

int main()

{

    char src[] = "geeksforgeeks";


    // The destination string size is 14.

    char dest[14];


    // copying n bytes of src into dest.

    strncpy(dest, src, 14);

    printf("Copied string: %s\n", dest);


    return 0;

}
```

Output:

```
Copied string: geeksforgeeks
```

- strrchr: The **strrchr()** function in C/C++ locates the last occurrence of a character in a string. It returns a pointer to the last occurrence in the string. The terminating null character is considered part of the C string. Therefore, it can also be located to retrieve a pointer to the end of a string. It is defined in **cstring** header file.
  Syntax :
- `const char* strrchr( const char* str, int ch )`
- `or`
- `char* strrchr( char* str, int ch )`

Parameter : The function takes two mandatory parameters which are described below:

- **str :** specifies the pointer to the null terminated string to be searched for.
- **ch:** specifies the character to be search for.

Return Value: The function returns a pointer to the last location of **ch** in string, if the **ch** is found. If not found, it returns a null pointer.
Below programs illustrate the above function:

Program 1:

filter_none

edit
play_arrow
brightness_4

```
// C code to demonstrate the working of
// strrchr()

#include <stdio.h>
#include <string.h>

// Driver function
int main()
{

    // initializing variables
    char st[] = "GeeksforGeeks";
    char ch = 'e';
    char* val;

    // Use of strrchr()
    // returns "ks"
    val = strrchr(st, ch);

    printf("String after last %c is : %s \n",
           ch, val);

    char ch2 = 'm';

    // Use of strrchr()
    // returns null
    // test for null
    val = strrchr(st, ch2);
```

```
        printf("String after last %c is : %s ",
                ch2, val);


        return (0);
    }
```

Output:

```
String after last e is : eks
String after last m is : (null)
```

12.200. print initials of a name

```
ALGORITHM:-

step 1 : Start
step 2 : Take an character array
step 3 : Take the input from user
step 4 : While(string[i]!='')
step 5 : If [i]th value is ' '(space)
then the print the i+1 character
step 6 : Exit


CopyPROGRAM CODE:-

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

int main()
{
    char str[20];
    int i=0;
    printf("Enter a string: ");
    gets(str);
    printf("%c", *str);
    while(str[i] != '')
```

```
    {
        if(str[i] == ' ')
        {
            i++;
            printf("%c", *(str + i));
        }
        i++;
    }
    return 0;
}
```

CopyOUTPUT:-

```
Enter a string:
C Programming Simply

CPS
```

13.201) copy one string to another without using library functions

```c
#include <stdio.h>
int main()
{
    char s1[100], s2[100], i;

    printf("Enter string s1: ");
    scanf("%s",s1);

    for(i = 0; s1[i] != '\0'; ++i)
    {
        s2[i] = s1[i];
    }

    s2[i] = '\0';
    printf("String s2: %s", s2);

    return 0;
}
```

13.202) explain memory allocation in array

We have already discussed that whenever an array is declared in the program, contiguous memory to it elements are allocated. Initial address of the array – address of the first element of the array is called base address of the array. Each element will occupy the memory space required to accommodate the values for its type, i.e.; depending on elements datatype, 1, 4 or 8 bytes of memory is allocated for each elements. Next successive memory address is allocated to the next element in the array. This process of allocating memory goes on till the number of element in the array gets over.

One Dimensional Array

Below diagram shows how memory is allocated to an integer array of N elements. Its base address – address of its first element is 10000. Since it is an integer array, each of its element will occupy 4 bytes of space. Hence first element occupies memory from 10000 to 10003. Second element of the array occupies immediate next memory address in the memory, i.e.; 10004 which requires another 4 bytes of space. Hence it occupies from 10004 to 10007. In this way all the N elements of the array occupies the memory space.

If the array is a character array, then its elements will occupy 1 byte of memory each. If it is a float array then its elements will occupy 8 bytes of memory each. But this is not the total size or memory allocated for the array. They are the sizes of individual elements in the array. If we need to know the total size of the array, then we need to multiply the number of elements with the size of individual element.

i.e.; **Total memory allocated to an Array = Number of elements * size of one element**

Total memory allocated to an Integer Array of N elements = Number of elements * size of one element

= N * 4 bytes
= 10 * 4 bytes = **40 Bytes**, where N = 10
= 500 * 4 bytes = **2000 Bytes**, where N = 500

Total memory allocated to an character Array of N elements= Number of elements * size of one element

= N * 1 Byte
= 10 * 1 Byte = **10 Bytes**, where N = 10
= 500 * 1 Byte = **500 Bytes**, where N=500

This is how memory is allocated for the single dimensional array.

Multidimensional Array

In the case of multidimensional array, we have elements in the form of rows and columns. Here also memories allocated to the array are contiguous. But the elements assigned to the memory location depend on the two different methods:

Row Major Order

Let us consider a two dimensional array to explain how row major order way of storing elements works. In the case of 2D array, its elements are considered as rows and columns of a table. When we represent an array as intArr[i][j], the first index of it represents the row elements and the next index represents the column elements of each row. When we store the array elements in row major order, first we will store the elements of first row followed by second row and so on. Hence in the memory we can find the elements of first row followed by second row and so on. In memory there will not be any separation between the rows. We have to code in such a way that we have to count the number of elements in each row depending on its column index. But in memory all the rows and their columns will be contiguous. Below diagram will illustrate the same for a 2D array of size 3X3 i.e.; 3 rows and 3 columns.

Array indexes always start from 0. Hence the first element of the 2D array is at intArr[0][0]. This is the first row-first column element. Since it is an integer array, it occupies 4 bytes of space. Next memory space is occupied by the second element of the first row, i.e.; intArr [0][1] – first row-second column element. This continues till all the first row elements are occupied in the memory. Next it picks the second row elements and is placed in the same way as first row. This goes on till all the elements of the array are occupies the memory like below. This is how it is placed in the memory. But seeing the memory address or the value stored in the memory we cannot predict which is the first row or second row or so.

Total size/ memory occupied by 2D array is calculated as

Total memory allocated to 2D Array = Number of elements * size of one element
= Number of Rows * Number of Columns * Size of one element

Total memory allocated to an Integer Array of size MXN = Number of elements * size of one element

=M Rows* N Columns * 4 Bytes
= 10*10 * 4 bytes = **400 Bytes**, where M =N = 10
= 500*5 *4 bytes= **10000 Bytes**, where M=500 and N= 5

Total memory allocated to an character Array of N elements= Number of elements *

size of one element

= M Rows* N Columns * 1 Byte

= 10*10 * 1 Byte = **100 Bytes**, where N = 10

= 500*5 * 1 Byte = **2500 Bytes**, where M=500 and N= 5

Column Major Order

This is the opposite method of row major order of storing the elements in the memory. In this method all the first column elements are stored first, followed by second column elements and so on.

Total size/ memory occupied by 2D array is calculated as in the same way as above.

Total memory allocated to 2D Array = Number of elements * size of one element

= Number of Rows * Number of Columns * Size of one element

Total memory allocated to an Integer Array of size MXN = Number of elements * size of one element

=M Rows* N Columns * 4 Bytes

= 10*10 * 4 bytes = **400 Bytes**, where M =N = 10

= 500*5 *4 bytes= **10000 Bytes**, where M=500 and N= 5

Total memory allocated to an character Array of N elements= Number of elements * size of one element

= M Rows* N Columns * 1 Byte

= 10*10 * 1 Byte = 100 Bytes, where N = 10

= 500*5 * 1 Byte = 2500 Bytes, where M=500 and N= 5

If an array is 3D or multidimensional array, then the method of allocating memory is either row major or column major order. Whichever is the method, memory allocated for the whole array is contiguous and its elements will occupy them in the order we choose – row major or column major. The total size of the array is the **total number of elements * size of one element**.

13.202 ii) union of two arrays

```
#include <stdio.h>
int main()
{
int a[10], b[10], flag = 0, n1, n2, i, j;
```

```
printf("Enter array1 size : ");
scanf("%d",&n1);
printf("\nEnter array2 size : ");
scanf("%d",&n2);
printf("\nEnter array1 element : ");
for(i = 0;i < n1;i++)
scanf("%d",&a[i]);
printf("\nEnter array2 element : ");
for(i = 0;i < n2;i++)
scanf("%d",&b[i]);
printf("\nUnion:");
for(i = 0;i < n1;i++)
printf("%d, ",a[i]);
for(i = 0;i < n2;i++)
{
for(j = 0;j < n1;j++)
{
if(b[i] == a[j])
{
flag = 1;
}
}
if(flag == 0)
{
printf("%d, ",b[i]);
}
flag = 0;
}
return 0;
}
```

13.203.i) convert lowercase to upper case string in c

Function strupr in C

```
#include <stdio.h>
#include <string.h>
int main()
{
   char string[1000];

   printf("Input a string to convert to upper
case\n");
   gets(string);

   printf("The string in upper case:
%s\n", strupr(string));

   return  0;
}
```

Change string to upper case without strupr

```
#include <stdio.h>
void upper_string(char []);

int main()
{
   char string[100];

   printf("Enter a string to convert it into upper
case\n");
   gets(string);

   upper_string(string);

   printf("The string in upper case: %s\n", string);

   return 0;
}

void upper_string(char s[]) {
   int c = 0;

   while (s[c] != '\0') {
      if (s[c] >= 'a' && s[c] <= 'z') {
         s[c] = s[c] - 32;
      }
      c++;
```

```
    }
}
```

13.229.ii macros vs functions

Macros are no longer recommended as they cause following issues. There is a better way in modern compilers that is inline functions and const variable. Below are disadvantages of macros:

a) There is no type checking

b) Difficult to debug as they cause simple replacement.

c) Macro don’t have namespace, so a macro in one section of code can affect other section.

d) Macros can cause side effects as shown in above CUBE() example.

MACRO	FUNCTION
Macro is Preprocessed	Function is Compiled
No Type Checking is done in Macro	Type Checking is Done in Function
Using Macro increases the code length	Using Function keeps the code length unaffected
Use of macro can lead to side effect at later stages	Functions do not lead to any side effect in any case
Speed of Execution using Macro is Faster	Speed of Execution using Function is Slower
Before Compilation, macro name is replaced by macro value	During function call, transfer of control takes place
Macros are useful when small code is repeated many times	Functions are useful when large code is to be written

MACRO	FUNCTION
Macro does not check any Compile-Time Errors	Function checks Compile-Time Errors

13.230.i) swap two numbers using call by reference

```
#include <stdio.h>

/* Swap function declaration */
void swap(int * num1, int * num2);

int main()
{
    int num1, num2;

    /* Input numbers */
    printf("Enter two numbers: ");
    scanf("%d%d", &num1, &num2);

    /* Print original values of num1 and num2 */
    printf("Before swapping in main n");
    printf("Value of num1 = %d \n", num1);
    printf("Value of num2 = %d \n\n", num2);

    /* Pass the addresses of num1 and num2 */
    swap(&num1, &num2);

    /* Print the swapped values of num1 and num2 */
    printf("After swapping in main n");
    printf("Value of num1 = %d \n", num1);
    printf("Value of num2 = %d \n\n", num2);

    return 0;
}


/**
 * Function to swap two numbers
 */
void swap(int * num1, int * num2)
{
    int temp;

    // Copy the value of num1 to some temp variable
    temp = *num1;

    // Copy the value of num2 to num1
    *num1= *num2;

    // Copy the value of num1 stored in temp to num2
    *num2= temp;
```

```
    printf("After swapping in swap function n");
    printf("Value of num1 = %d \n", *num1);
    printf("Value of num2 = %d \n\n", *num2);
}
```

13.230 ii) passing array to function as parameter

Just like variables, array can also be passed to a function as an argument . In this guide, we will learn how to pass the array to a function using call by value and call by reference methods.

To understand this guide, you should have the knowledge of following C Programming topics:

1. C – Array
2. Function call by value in C
3. Function call by reference in C

Passing array to function using call by value method

As we already know in this type of function call, the actual parameter is copied to the formal parameters.

```
#include <stdio.h>
void disp( char ch)
{
   printf("%c ", ch);
}
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;
}
```

Output:

```
a b c d e f g h i j
```

Passing array to function using call by reference

When we pass the address of an array while calling a function then this is called function call by reference. When we pass an address as an argument, the function declaration should have a pointer as a parameter to receive the passed address.

```c
#include <stdio.h>
void disp( int *num)
{
    printf("%d ", *num);
}

int main()
{
     int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
     for (int i=0; i<10; i++)
     {
         /* Passing addresses of array elements*/
         disp (&arr[i]);
     }

     return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 0
```

How to pass an entire array to a function as an argument?

In the above example, we have passed the address of each array element one by one using a for loop in C. However you can also pass an entire array to a function like this:

Note: The array name itself is the address of first element of that array. For example if array name is arr then you can say that **arr** is equivalent to the **&arr[0]**.

```c
#include <stdio.h>
void myfuncn( int *var1, int var2)
{
        /* The pointer var1 is pointing to the first element of
         * the array and the var2 is the size of the array. In the
         * loop we are incrementing pointer so that it points to
         * the next element of the array on each increment.
         *
         */
    for(int x=0; x<var2; x++)
    {
        printf("Value of var_arr[%d] is: %d \n", x, *var1);
        /*increment pointer for next element fetch*/
        var1++;
    }
}

int main()
{
     int var_arr[] = {11, 22, 33, 44, 55, 66, 77};
     myfuncn(var_arr, 7);
     return 0;
}
```

Output:

```
Value of var_arr[0] is: 11
Value of var_arr[1] is: 22
Value of var_arr[2] is: 33
Value of var_arr[3] is: 44
Value of var_arr[4] is: 55
Value of var_arr[5] is: 66
Value of var_arr[6] is: 77
```

13.232.i) gcd using recursive functions

```c
#include <stdio.h>
int hcf(int n1, int n2);
int main()
{
   int n1, n2;
   printf("Enter two positive integers: ");
   scanf("%d %d", &n1, &n2);

   printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1,n2));
   return 0;
}

int hcf(int n1, int n2)
{
    if (n2 != 0)
       return hcf(n2, n1%n2);
    else
       return n1;
}
```

13.231.i) direct and indirect recursion

Direct Recursion

When in the body of a method there is a call to the same method, we say that the method is **directly recursive**. That means **Direct recursion** occurs when a method invokes itself.

Indirect Recursion or mutually recursive

If method A calls method B, method B calls method C, and method C calls method A we call the methods A, B and C **indirectly recursive** or **mutually recursive**.

Indirect recursion occurs when a method invokes another method, eventually resulting in the original method being invoked again.

Chains of calls in indirect recursion can contain multiple methods, as well as branches, i.e. in the presence of one condition one method to be called, and provided a different condition another to be called.

The depth of indirection may vary.

Indirect recursion requires the same attention to base cases.

Direct Recursion	Indirect Recursion
In the direct recursion, only one function is called by itself.	In indirect recursion more than one function are by the other function and number of times.
direct recursion makes overhead.	The indirect recursion does not make any overhead as direct recursion
The direct recursion called by the same function	While the indirect function called by the other function
In direct function, when function called next time, value of local variable will stored	but in indirect recursion, value will automatically lost when any other function is called local variable
Direct function engaged memory location	while local variable of indirect function not engaged it

Structure of direct function
`int num()`
`{`
`...`
`...`
`int num();`
`}`
	Structure of indirect function
	`int num()`
	`{`
	`...`
	`...`
	`int sum();`
	`}`
	`int sum()`
	`{`
	`...`

	...
	int num();
	
	}

13.231.i) calculate power using recursion

```c
#include <stdio.h>

int power(int n1, int n2);

int main()
{
    int base, powerRaised, result;

    printf("Enter base number: ");
    scanf("%d",&base);

    printf("Enter power number(positive integer): ");
    scanf("%d",&powerRaised);

    result = power(base, powerRaised);

    printf("%d^%d = %d", base, powerRaised, result);
    return 0;
}

int power(int base, int powerRaised)
{
    if (powerRaised != 0)
        return (base*power(base, powerRaised-1));
    else
        return 1;
}
```

Output

```
Enter base number: 3
Enter power number(positive integer): 4
3^4 = 81
```

13.259.ii) pointer to pointer concept

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int −

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example −

Live Demo

```
#include <stdio.h>

int main () {

   int  var;
   int  *ptr;
   int  **pptr;

   var = 3000;

   /* take the address of var */
   ptr = &var;

   /* take the address of ptr using address of operator & */
   pptr = &ptr;

   /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);

   return 0;
}
```

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

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

We already know that a pointer points to a location in memory and thus used to store the address of variables. So, when we define a pointer to pointer. The first pointer is used to store the address of the variable. And the second pointer is used to store the address of the first pointer. That is why they are also known as double pointers.

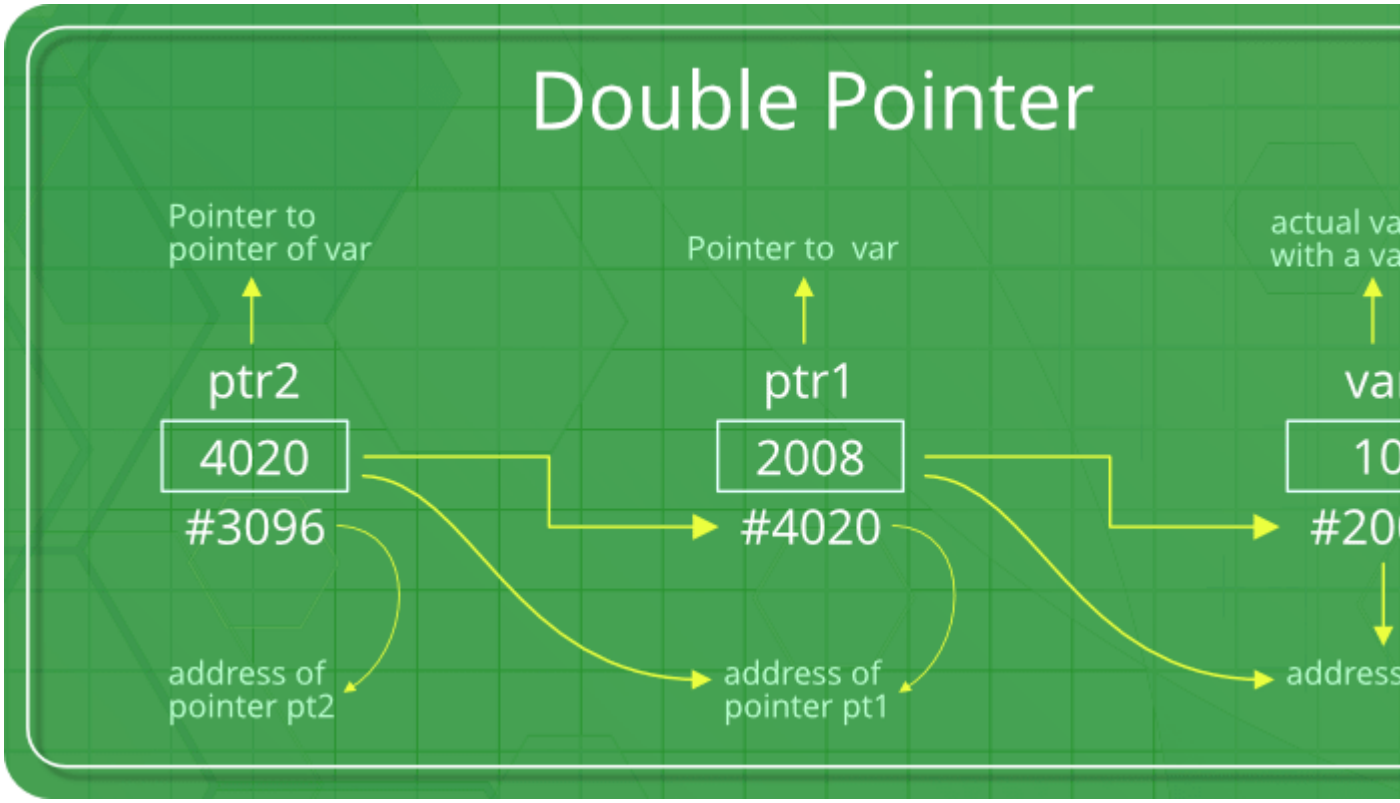
How to declare a pointer to pointer in C?

Declaring Pointer to Pointer is similar to declaring pointer in C. The difference is we have to place an additional '*' before the name of pointer.

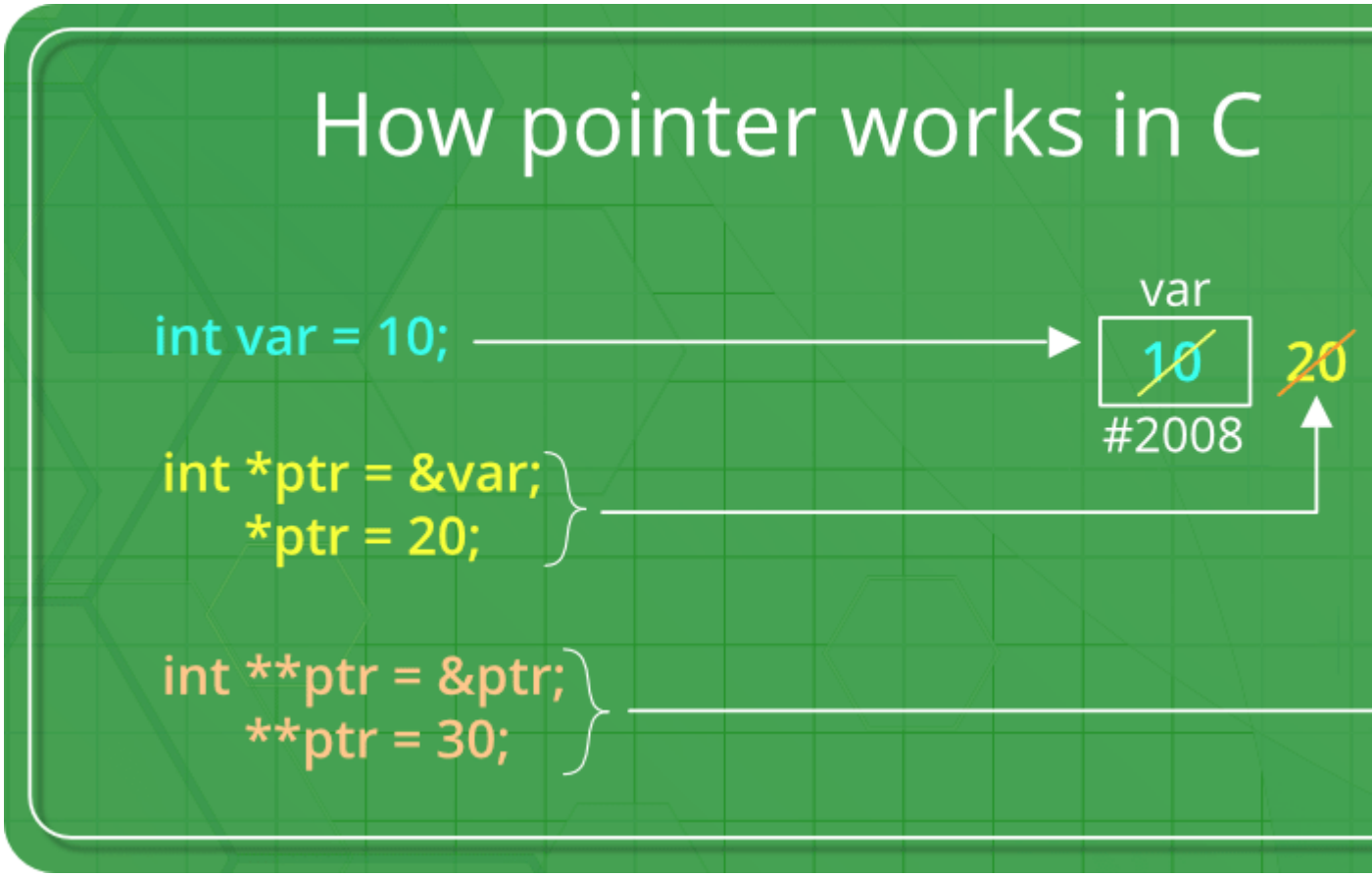
Syntax:

```
int **ptr;    // declaring double pointers
```

Below diagram explains the concept of Double Pointers:



The above diagram shows the memory representation of a pointer to pointer. The first pointer ptr1 stores the address of the variable and the second pointer ptr2 stores the address of the first pointer.



Let us understand this more clearly with the help of the below program:

filter_none
edit
play_arrow
brightness_4

```
#include <stdio.h>

// C program to demonstrate pointer to pointer
int main()
{
    int var = 789;

    // pointer for var
    int *ptr2;

    // double pointer for ptr2
    int **ptr1;

    // storing address of var in ptr2
    ptr2 = &var;

    // Storing address of ptr2 in ptr1
    ptr1 = &ptr2;

    // Displaying value of var using
```

```
    // both single and double pointers
    printf("Value of var = %d\n", var );
    printf("Value of var using single pointer = %d\n", *ptr2 );
    printf("Value of var using double pointer = %d\n", **ptr1);

  return 0;
}
```

Output:

```
Value of var = 789

Value of var using single pointer = 789

Value of var using double pointer = 789
```

13.261 i) explain pointer to function

Function Pointers

As we know by definition that pointers point to an address in any memory location, they can also point to at the beginning of executable code as functions in memory.

A pointer to function is declared with the * ,the general statement of its declaration is:

```
return_type (*function_name)(arguments)
```

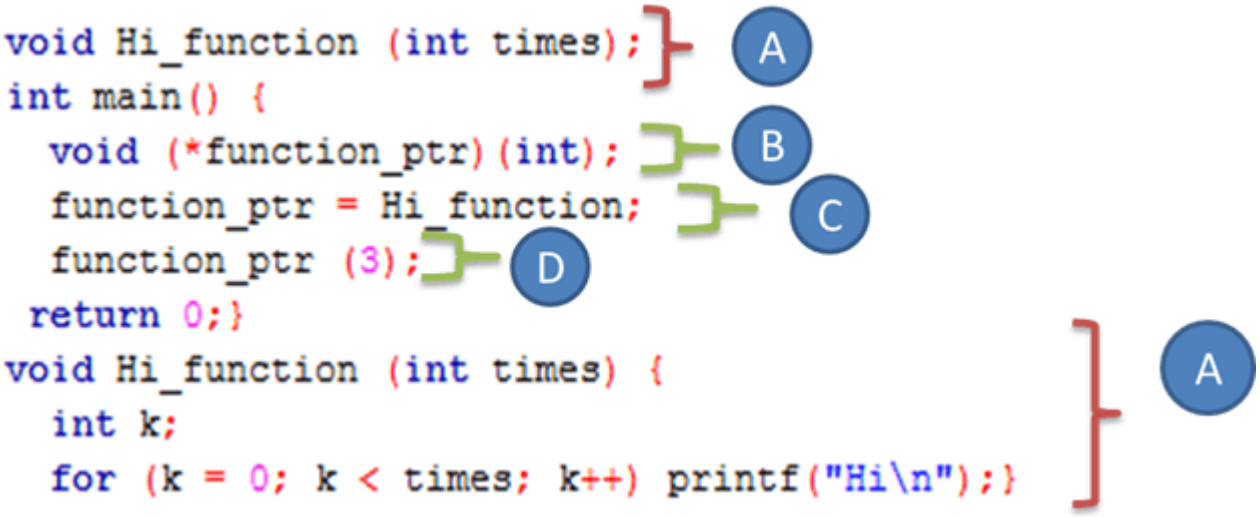
You have to remember that the parentheses around (*function_name) are important because without them, the compiler will think the function_name is returning a pointer of return_type.

After defining the function pointer, we have to assign it to a function. For example, the next program declares an ordinary function, defines a function pointer, assigns the function pointer to the ordinary function and after that calls the function through the pointer:

```
#include <stdio.h>
void Hi_function (int times); /* function */
int main() {
  void (*function_ptr)(int);  /* function pointer Declaration */
  function_ptr = Hi_function;  /* pointer assignment */
  function_ptr (3);  /* function call */
 return 0;}
void Hi_function (int times) {
  int k;
  for (k = 0; k < times; k++) printf("Hi\n");}
```

Output:

```
Hi
Hi
Hi
```



1. We define and declare a standard function which prints a Hi text k times indicated by the parameter times when the function is called
2. We define a pointer function (with its special declaration) which takes an integer parameter and doesn't return anything.
3. We initialize our pointer function with the Hi_function which means that the pointer points to the Hi_function().
4. Rather than the standard function calling by taping the function name with arguments, we call only the pointer function by passing the number 3 as arguments, and that's it!

Keep in mind that the function name points to the beginning address of the executable code like an array name which points to its first element. Therefore, instructions like function_ptr = &Hi_function and (*funptr)(3) are correct.

NOTE: It is not important to insert the address operator & and the indirection operator * during the function assignment and function call.

13. 261 ii) add two matrix using pointer

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

int a[5][5],b[5][5],row,col;

void add(int(*)[5]);

int main()
{
```

```
    int c[5][5],i,j;
    clrscr();
    printf(\"Enter row : \");
    scanf(\"%d\",&row);
    printf(\"Enter column : \");
    scanf(\"%d\",&col);
    printf(\"Enter matrix A :\\n\");
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        {
            scanf(\"%d\",&a[i][j]);
        }
    }
    printf(\"Enter matrix B :\\n\");
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        {
            scanf(\"%d\",&b[i][j]);
        }
    }
    add(c);
    printf(\"Addition :\\n\");
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        {
            printf(\"%d\\t\",c[i][j]);
        }
        printf(\"\\n\");
    }
    getch();
    return 0;
}

void add(int c[5][5])
{
    int i,j;
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        {
            c[i][j]=a[i][j]+b[i][j];
        }
    }
}
```

13.259.i) copy string using pointers

```
#include <stdio.h>
#define MAX_SIZE 100 // Maximum size of the string

int main()
{
    char text1[MAX_SIZE], text2[MAX_SIZE];
    char * str1 = text1;
    char * str2 = text2;
```

```
    /* Input string from user */
    printf("Enter any string: ");
    gets(text1);

    /* Copy text1 to text2 character by character */
    while(*(str2++) = *(str1++));

    printf("First string = %s\n", text1);
    printf("Second string = %s\n", text2);

    return 0;
}
```

LISP –check from ma’am notes