

Que 1 - Machine Language

- Machine language is the language understood by a computer.
- It is very difficult to understand by human, but it is the only thing that the computer can work with.
- All programs and programming languages eventually generate or run programs in machine language.
- Machine language is made up of instructions and data that are all binary numbers.
- The limited set of instructions that a CPU can understand directly is called **machine code** (or **machine language** or an **instruction set**).
- Here is a sample machine language instruction: 10110000 01100001
- Back when computers were first invented, programmers had to write programs directly in machine language, which was a very difficult and time consuming thing to do.
- Machine language are first generation language

Que 2 - Assembly language

- Assembly programming is low-level programming language or second generation language.
- some basic syntax to represent machine code for a specific CPU.
- An **assembler** is used to translate the assembly code into the machine code for the computer.
- A program created from assembly can be more efficient and faster than a program created with a machine level.
- Because machine language is so hard to understand for humans to read and understand, assembly language was invented.

- In an assembly language, each instruction is identified by a short form (rather than a set of bits), and names and other numbers can be used. Here is the same instruction as above in assembly language: `mov al, 061h`
- This makes assembly much easier to read and write than machine language.
- However, the CPU can not understand assembly language directly.
- Instead, the assembly program must be translated into machine language before it can be executed by the computer.
- This is done by using a program called an **assembler**.
- However, assembly still has some drawbacks .
- First, assembly languages still require a lot of instructions to do even simple tasks.
- Second, assembly language still isn't very portable -- a program written in assembly for one CPU will likely not work on hardware that uses a different instruction set, and would have to be rewritten or extensively modified.

Advantages	Disadvantages
Assembly language is easier to understand and use as compared to machine language.	Like machine language, it is also machine dependent/specific.
It is easy to locate and correct errors.	Since it is machine dependent, the programmer also needs to understand the hardware.
It is easily modified.	

Que 3 High-Level Language

- High-level computer languages use formats that are similar to English.

- The purpose of developing high-level languages was to enable people to write programs easily, in their own native language environment (English).
- High-level languages are basically symbolic languages that use English words and/or mathematical symbols rather than mnemonic codes.
- Each instruction in the high-level language is translated into many machine language instructions that the computer can understand.

Most common programming languages are considered high-level languages. Examples include:

- | | |
|---------------|----------------|
| 1. C++ | 7. Objective C |
| 2. C# | 8. Pascal |
| 3. Cobol | 9. Perl |
| 4. Fortran | 10. PHP |
| 5. Java | 11. Python |
| 6. JavaScript | 12. Swift |

- Each of these languages use different syntax.
- Some are designed for writing desktop software programs, while others are best-suited for web development.
- But they all are considered high-level since they must be processed by a compiler or interpreter before the code is executed.
- Source code written in languages like C++ and C# must be compiled into machine code in order to run.
- The compilation process converts the human-readable syntax of the high-level language into low-level code for a specific processor.
- Source code written in scripting languages like Perl and PHP can be run through an interpreter, which converts the high-level code into a low-level language on-the-fly.

Advantages

High-level languages
are user-friendly

Disadvantages

A high-level language
has to be translated
into the machine
language by a

	translator, which takes up time
They are similar to English and use English vocabulary and well-known symbols	The object code generated by a translator might be inefficient compared to an equivalent assembly language program
They are easier to learn	
They are easier to maintain	
They are problem-oriented rather than 'machine'-based	
A program written in a high-level language can be translated into many machine languages and can run on any computer for which there exists an appropriate translator	
The language is independent of the machine on which it is used i.e. programs developed in a high-level language can be run on any computer	
text	

Que 4 – Limitations and features of programming language

Limitations

-
- The bugs can get annoying and they are hard to solve sometimes.
 - If you are not good at maths then it will not easy to program, Lots of thinking involved.
 - C language does not have concept of Object-Oriented features.
 - Modification of one part may lead to change the entire code.
-

Features

- **Simplicity:** the language must offer clear and simple concepts that facilitate its learning and application, in a way that is simple to understand and maintain.
- **Naturalness:** this means that its application in the area for which it was designed must be done naturally, providing operators, structures and syntax for operators to work efficiently.
- **Abstraction:** it is the ability to define and use complicated structures or operations while ignoring some details, which influence writing ability.
- **Efficiency:** Programming languages must be translated and executed efficiently so as not to take up too much memory space or require too much time.
- **Structuring:** the language allows programmers to write their codes according to structured programming concepts, to avoid creating errors.
- **Compactness:** with this characteristic, it is possible to express operations in brief, without having to write too many details.
- **Locality:** refers to the codes concentrating on the part of the program with which you are working at a given time.

Que 5 : Difference between Procedural and Non-Procedural language:

	PROCEDURAL LANGUAGE	NON-PROCEDURAL LANGUAGE
1	It is command-driven language.	It is a function-driven language
2	It works through the state of machine.	It works through the mathematical functions.
3	Its semantics(logics) are quite tough.	Its semantics are very simple.
4	It returns only restricted data types and allowed values.	It can return any datatype or value
5	Overall efficiency is very high.	Overall efficiency is low as compared to Procedural Language.
6	Size of the program written in Procedural language is large.	Size of the Non-Procedural language programs are small.
7	It is not suitable for time critical applications.	It is suitable for time critical applications.
8	Iterative loops and Recursive calls both are used in the Procedural languages.	Recursive calls are used in Non-Procedural languages.

Que 6 : Explain Algorithm and flowchart with example.



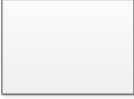


Algorithm

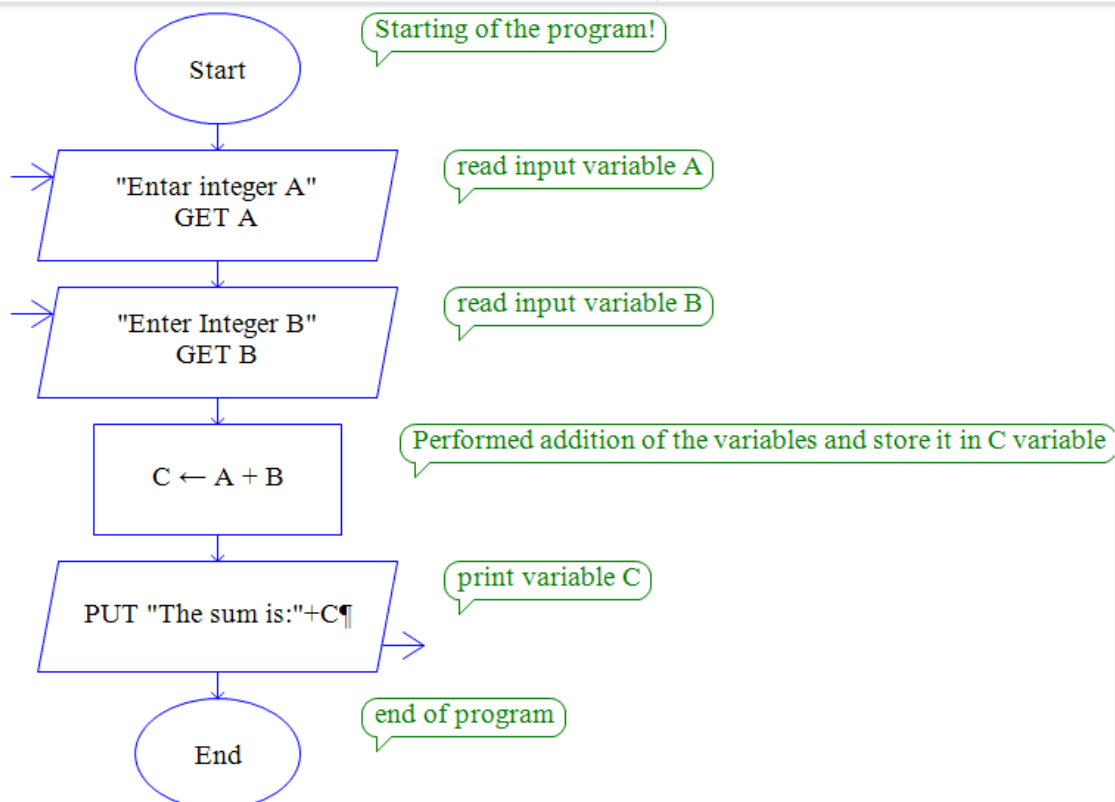
- To write a logical step-by-step method to solve the problem is called the algorithm; in other words, an algorithm is a procedure for solving problems.
- In order to solve a mathematical or computer problem, this is the first step in the process.
- An algorithm includes calculations, reasoning, and data processing. Algorithms can be presented by natural languages, pseudo code, and flowcharts, etc.

Flowchart

- A flowchart is the graphical or pictorial representation of an algorithm with the help of different symbols, shapes, and arrows to demonstrate a process or a program.
- With algorithms, we can easily understand a program.

- The main purpose of using a flowchart is to analyze different methods.
- Several standard symbols are applied in a flowchart:

Terminal Box - Start / End	
Input / Output	
Process / Instruction	
Decision	
Connector / Arrow	



Que 7 : Explain Basic Structure of C Program. with example.



Figure: Basic Structure Of C Program

Moving on to the next bit of this basic structure of a C program article,

Documentation Section

The documentation section is the part of the program where the programmer gives the details associated with the program. He usually gives the name of the program, the details of the author and other details like the time of coding and description. It gives anyone reading the code the overview of the code.

Example

```
/**  
* File Name: Helloworld.c  
* Author: Manthan Naik  
* date: 09/08/2019  
* description: a program to display hello world  
*      no input needed  
*/  
8
```


Moving on to the next bit of this basic structure of a C program article,

Link Section

This part of the code is used to declare all the header files that will be used in the program. This leads to the compiler being told to link the header files to the system libraries.

Example

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

Moving on to the next bit of this basic structure of a C program article,

Definition Section

In this section, we define different constants. The keyword define is used in this part.

```
1  #define PI=3.14
```

Moving on to the next bit of this basic structure of a C program article,

Global Declaration Section

This part of the code is the part where the global variables are declared. All the global variable used are declared in this part. The user-defined functions are also declared in this part of the code.

```
1  float area(float r);
```

```
2  int a=7;
```

Moving on to the next bit of this basic structure of a C program article,

Main Function Section

Every C-programs needs to have the main function. Each main function contains 2 parts. A declaration part and an Execution part. The declaration part is the part where all the variables are declared. The execution part begins with the curly brackets and ends with the curly close bracket. Both the declaration and execution part are inside the curly braces.

```
int main(void)
```

```
{
```

```
int a=10;
```

```
printf(" %d", a);  
return 0;  
}
```

Moving on to the next bit of this basic structure of a C program article,

Sub Program Section

All the user-defined functions are defined in this section of the program.

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

Sample Program

The C program here will find the area of a circle using a user-defined function and a global variable pi holding the value of pi

```
/**  
 * file: circle.c  
 * author: yusuf shakeel  
 * date: 2010-11-25  
 * description: program to find the area of a circle  
 *              using the radius r  
 */  
  
#include <stdio.h>  
  
#define PI 3.1416  
  
float area(float r);  
  
int main(void)  
{  
    float r = 10;  
    printf("Area: %.2f", area(r));  
    return 0;  
}
```

```

}

float area(float r) {
    getch();
    return PI * r * r;
}

```

Que 8 : Explain Character set & C Tokens

Character set

- Like every other language, 'C' also has its own character set. A program is a set of instructions which is used, while executing the program and generating an output.
- The data that is processed by a program consists of various characters and symbols.
- The output generated is also a combination of characters and symbols.

A character set in 'C' is divided into,

- Letters
- Numbers
- Special characters
- White spaces (blank spaces)

1. Letters

- Uppercase characters (A-Z)
- Lowercase characters (a-z)

2. Numbers

- All the digits from 0 to 9

3. White spaces

- Blank space
- New line \n
- Carriage return
- Horizontal tab

4. Special characters

- Special characters in 'C' are shown in the given table,

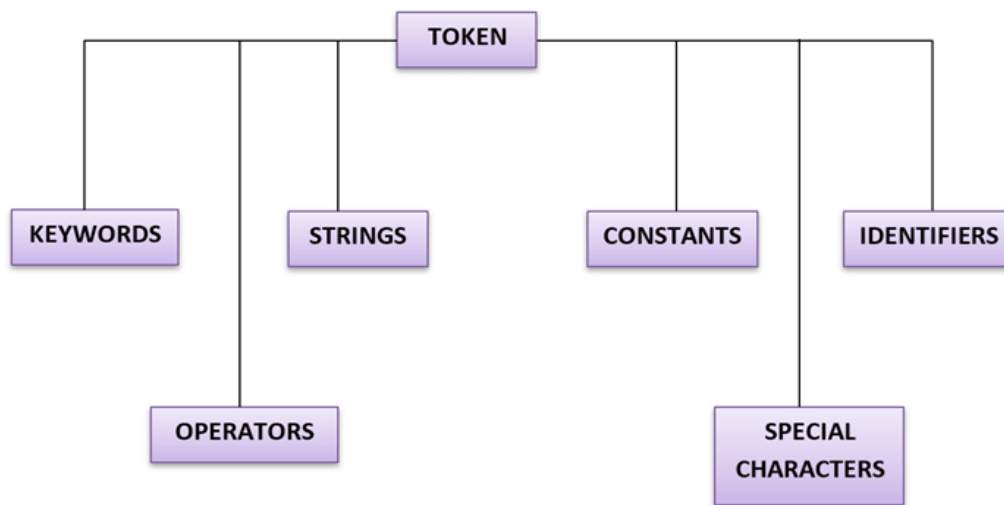
, (comma)	{ (opening curly bracket)
. (period)	} (closing curly bracket)

; (semi-colon)	[(left bracket)
: (colon)] (right bracket)
? (question mark)	((opening left parenthesis)
' (apostrophe)) (closing right parenthesis)
" (double quotation mark)	& (ampersand)
! (exclamation mark)	^ (caret)
(vertical bar)	+ (addition)
/ (forward slash)	- (subtraction)
\ (backward slash)	* (multiplication)
~ (tilde)	/ (division)
_ (underscore)	> (greater than or closing angle bracket)
\$ (dollar sign)	< (less than or opening angle bracket)
% (percentage sign)	# (hash sign)

Token

- TOKEN is the smallest unit in a 'C' program
- The compiler breaks a program into the smallest possible units (tokens) and proceeds to the various stages of the compilation.

- A token is divided into six different types, Keywords, Operators, Strings, Constants, Special Characters, and Identifiers.



Keywords

- In 'C' every word can be either a keyword or an identifier.
- Keywords have fixed meanings, and the meaning cannot be changed
- There are a total of 32 keywords in 'C'.
- Keywords are written in lowercase letters.

Following table represents the keywords in 'C'-

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

do

if

static

while

Identifiers

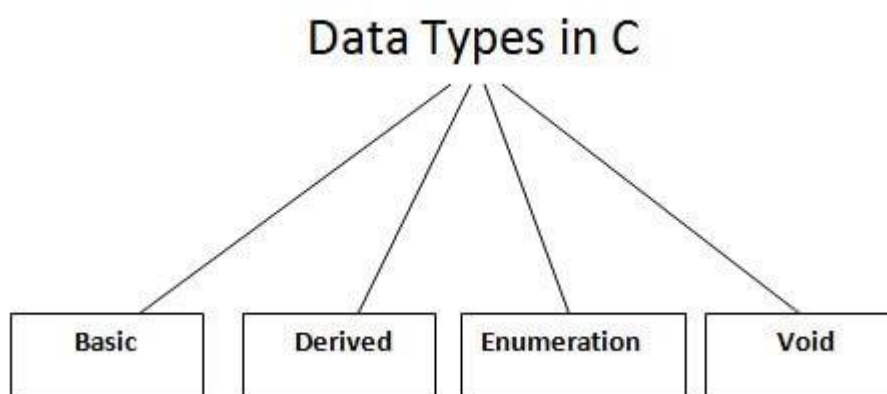
- An identifier is nothing but a name assigned to an element in a program. Example, name of a variable, function, etc.
- Identifiers are the user-defined names consisting of 'C' standard character set.
- As the name says, identifiers are used to identify a particular element in a program.
- Each identifier must have a unique name.
- Following rules must be followed for identifiers:
 1. The first character must always be an alphabet or an underscore.
 2. It should be formed using only letters, numbers, or underscore.
 3. A keyword cannot be used as an identifier.
 4. It should not contain any whitespace character.
 5. The name must be meaningful.

Que 9 : Explain Data types in C Language

A data type specifies the type of data that a variable can store

Each variable in C has an associated data type.

Each data type requires different amounts of memory and has some specific operations which can be performed over it.



There are the following data types in C language.

Types	Data Types
Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
Enumeration Data Type	enum
Void Data Type	void

Integer type

Integers are used to store **whole** numbers.

Size and range of Integer type on 16-bit machine:

Type	Size(bytes)	Range
int or signed int	2	-32,768 to 32767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Floating point type

Floating types are used to store **real** numbers.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
Float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Character type

Character types are used to store characters value.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

The void Type

The void type specifies that has no value is available. It is used in two kinds of situations –

Sr.No.	Types & Description
1	Function returns as void There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, void exit (int status);
2	Function arguments as void There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, int rand(void);

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.

User define datatypes

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.

Que 10 : Explain Constants in C Language

- Constants are like a variable, except that their value never changes during execution once defined.
- C Constants is the most fundamental and essential part of the C programming language.
- Constants in C are the fixed values that are used in a program, and its value remains the same during the entire execution of the program.
- Constants are also called **literals**.
- Constants can be any of the [data types](#).
- It is considered best practice to define constants using only *upper-case* names.
- Constants are categorized into two basic types, and each of these types has its subtypes/categories.
- These are:

Primary Constants

1. Numeric Constants

Integer Constants

Real Constants

2. Character Constants

Single Character Constants

String Constants

Backslash Character Constants

It's referring to a sequence of digits. Integers are of three types viz:

1. Decimal Integer
2. Octal Integer
3. Hexadecimal Integer

Example:

15, -265, 0, 99818, +25, 045, 0X6

Real constant

- The numbers containing fractional parts like 99.25 are called real or floating points constant.

Single Character Constants

- It simply contains a single character enclosed within ' abc ' (a pair of single quote).
- It is to be noted that the character '8' is not the same as 8.
- Character constants have a specific set of integer values known as ASCII values (American Standard Code for Information Interchange).

Example:

'X', '5', ';'

String Constants

- These are a sequence of characters enclosed in double quotes, and they may include letters, digits, special characters, and blank spaces.
- It is again to be noted that "G" and 'G' are different - because "G" represents a string as it is enclosed within a pair of double quotes whereas 'G' represents a single character.

Example:

"Hello!", "2015", "2+1"

Backslash character constant

- C supports some character constants having a backslash in front of it.
- The lists of backslash characters have a specific meaning which is known to the compiler.
- They are also termed as "Escape Sequence".

For Example:

`\t` is used to give a tab

`\n` is used to give a new line

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

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
const int a=10;
```

```
//a=11;
```

```
printf("%d",a);
```

```
getch();
```

```
}
```

Que 11 : Explain Variables in C Language

- A **variable** is a name of the memory location.
- It is used to store data.
- Its value can be changed, and it can be reused many times.
- It is a way to represent memory location through symbol so that it can be easily identified.

syntax :

```
data_type variable_list;
```

Example

```
int a;
```

```
float b;
```

```
char c;
```

- Here, a, b, c are variables.
- The int, float, char are the data types.

We can also provide values while declaring the variables as given below:

```
int a=10,b=20;//declaring 2 variable of integer type
float f=20.8;
char c='A';
```

Rules for defining variables

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only.
- It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

Valid variable names:

```
int a;
int _ab;
int a30;
```

Invalid variable names:

```
int 2;
int a b;
int long;
```

Types of Variables in C

There are many types of variables in c:

1. local variable
2. global variable
3. static variable
4. automatic variable
5. external variable

Local Variable

A variable that is declared inside the function or block is called a local variable.

It must be declared at the start of the block.

```
void function1()
{
int x=10;//local variable
```

```
}
```

You must have to initialize the local variable before it is used.

Global Variable

- A variable that is declared outside the function or block is called a global variable.
- Any function can change the value of the global variable.
- It is available to all the functions.
- It must be declared at the start of the block.

```
int value=20;//global variable
void function1(){
int x=10;//local variable
}
```

Static Variable

A static int variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over.

For example, we can use static int to count a number of times a function is called, but an auto variable can't be used for this purpose.

For example below program prints "1 2"

```
#include <stdio.h>
#include <conio.h>
int fun()
{
    static int count = 0;
    count++;
    return count;
}

int main()
{
    printf("%d ", fun());
    printf("%d ", fun());

    getch();
    return 0;
}
```

If you call this function many times, the **local variable will print the same value** for each function call, e.g. 1,1,1 and so on.

- But the **static variable will print the incremented value** in each function call, e.g. 1, 2, 3 and so on.

Automatic Variable

- All variables in C that are declared inside the block, are automatic variables by default.
- We can explicitly declare an automatic variable using **auto keyword**.

```
void main()
{
int x=10;//local variable (also automatic)
auto int y=20;//automatic variable
}
```

External Variable

- External variables are also known as global variables. These variables are defined outside the function.
- These variables are available globally throughout the function execution.
- The value of global variables can be modified by the functions.
“extern” keyword is used to declare and define the external variables

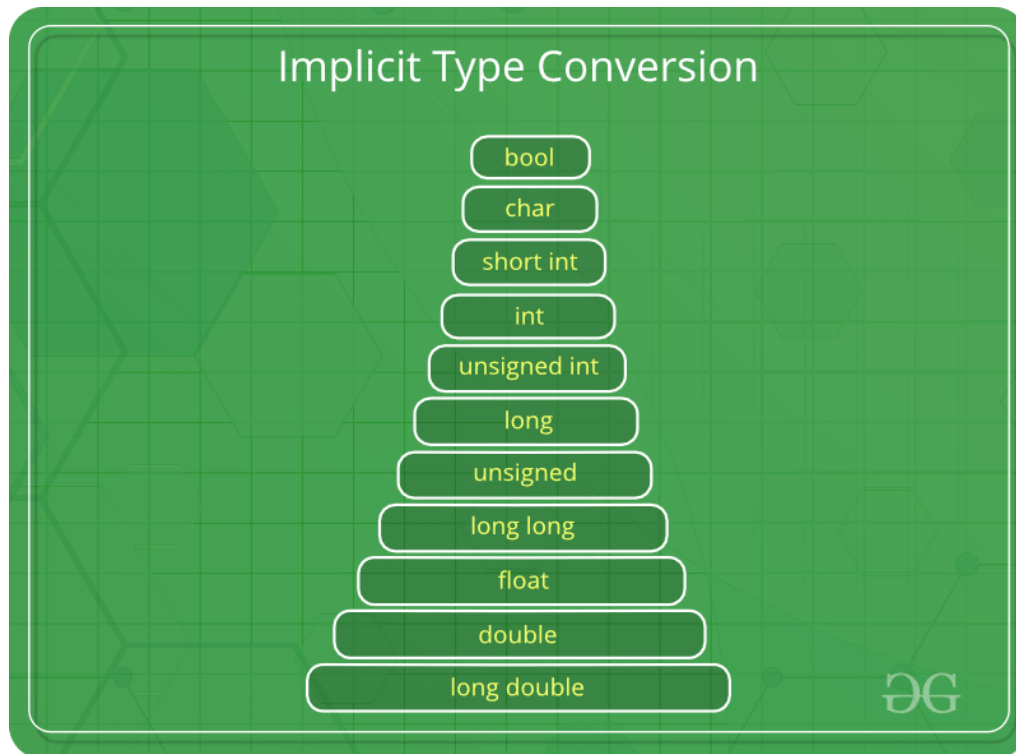
```
#include <stdio.h>
#include <conio.h>
```

```
extern int x = 32;
int b = 8;
int main()
{
    auto int a = 28;
    extern int b;
    printf("The value of auto variable : %d\n", a);
    printf("The value of extern variables x and b : %d,%d\n",x,b);
    x = 15;
    printf("The value of modified extern variable x : %d\n",x);
    getch();
    return 0;
}
```

Que 12: Explain Type Conversion or Type Casting in C.

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. Implicit Type Conversion



Also known as 'automatic type conversion'.

Done by the compiler on its own, without any external trigger from the user.

Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid loss of data.

All the data types of the variables are upgraded to the data type of the variable with largest data type.

**bool -> char -> short int -> int ->
unsigned int -> long -> unsigned ->
long long -> float -> double -> long double**

It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion:


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

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

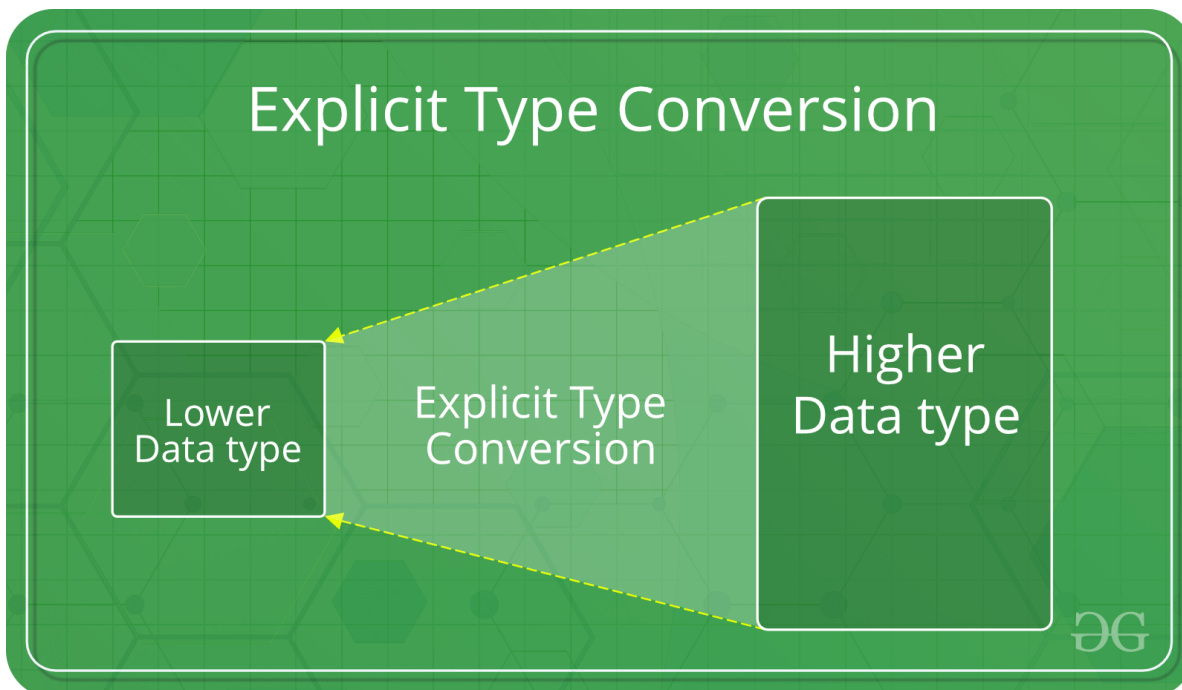
    // x is implicitly converted to float
    float z = x + 1.0;

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

OUTPUT

```
x = 107, z = 108.000000
```

2. Explicit Type Conversion



- This process is also called type casting and it is user defined.
- Here the user can type cast the result to make it of a particular data type.

The syntax in C:

```
(type) expression
```

Type indicated the data type to which the final result is converted.

filter_none

edit

play_arrow

brightness_4

```
// C program to demonstrate explicit type casting
```

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

```
int main()
```

```
{
```

```
    double x = 1.2;
```

```
    // Explicit conversion from double to int
```

```
    int sum = (int)x + 1;
```

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

```
    return 0;
```

```
}
```

Output:

```
sum = 2
```

Advantages of Type Conversion

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps us to compute expressions containing variables of different data types.

Que 13 : Explain Comments in 'C' Language

Comments in C

Comments in C language are used to provide information about lines of code. It is widely used for documenting code. There are 2 types of comments in the C language.

1. Single Line Comments
2. Multi-Line Comments

Single Line Comments

Single line comments are represented by double slash \\. Let's see an example of a single line comment in C.

```
#include<stdio.h>
int main(){
    //printing information
    printf("Hello C");
    return 0;
}
```

Multi Line Comments

Multi-Line comments are represented by slash asterisk * ... *. It can occupy many lines of code, but it can't be nested. Syntax:

```
/*
code
to be commented
*/
```

Let's see an example of a multi-Line comment in C.

```
#include<stdio.h>
int main(){
    /*printing information
    Multi-Line Comment*/
    printf("Hello C");
    return 0;
}
```

Que 14 : Programming Errors in C

- Errors are the problems or the faults that occur in the program, which makes the behaviour of the program abnormal, and experienced developers can also make these faults.
- Programming errors are also known as the bugs or faults, and the process of removing these bugs is known as **debugging**.
- These errors are detected either during the time of compilation or execution.
- Thus, the errors must be removed from the program for the successful execution of the program.

There are mainly five types of errors exist in C programming:

- **Syntax error**
- **Run-time error**
- **Linker error**
- **Logical error**
- **Semantic error**

Syntax error

- Syntax errors are also known as the compilation errors as they occurred at the compilation time, or we can say that the syntax errors are thrown by the compilers.
- These errors are mainly occurred due to the mistakes while typing or do not follow the syntax of the specified programming language.
- These mistakes are generally made by beginners only because they are new to the language.
- These errors can be easily debugged or corrected.

For example:

If we want to declare the variable of type integer,

int a; // this is the correct form

Int a; // this is an incorrect form.

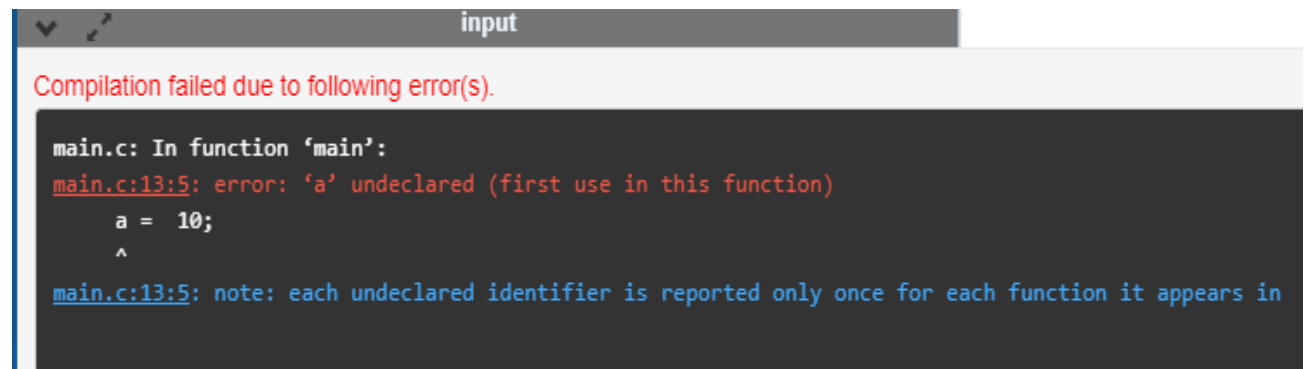
Commonly occurred syntax errors are:

- If we miss the parenthesis (}) while writing the code.
- Displaying the value of a variable without its declaration.
- If we miss the semicolon (;) at the end of the statement.

Let's understand through an example.

```
#include <stdio.h>
int main()
{
    a = 10;
    printf("The value of a is : %d", a);
    return 0;
}
```

Output

A screenshot of a compiler's output window. At the top, a tab labeled 'input' is visible. Below it, a red message states 'Compilation failed due to following error(s)'. The main area shows the following text: 'main.c: In function 'main':', 'main.c:13:5: error: 'a' undeclared (first use in this function)', a code snippet 'a = 10;' with a caret '^' pointing to the 'a', and a blue note: 'main.c:13:5: note: each undeclared identifier is reported only once for each function it appears in'.

```
main.c: In function 'main':
main.c:13:5: error: 'a' undeclared (first use in this function)
    a = 10;
    ^
main.c:13:5: note: each undeclared identifier is reported only once for each function it appears in
```

In the above output, we observe that the code throws the error that 'a' is undeclared. This error is nothing but the syntax error only.

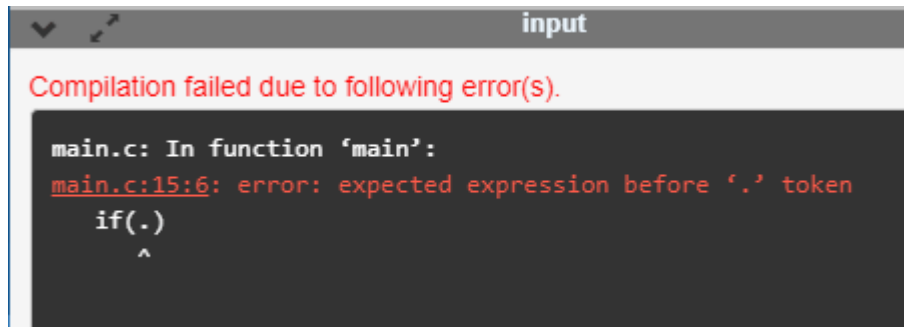
There can be another possibility in which the syntax error can exist, i.e., if we make mistakes in the basic construct. Let's understand this scenario through an example.

```
#include <stdio.h>
int main()
{
    int a=2;
    if(.) // syntax error

    printf("a is greater than 1");
    return 0;
}
```

In the above code, we put the (.) instead of condition in 'if', so this generates the syntax error as shown in the below screenshot.

Output



```
input
Compilation failed due to following error(s).

main.c: In function 'main':
main.c:15:6: error: expected expression before '.' token
    if(.)
       ^
```

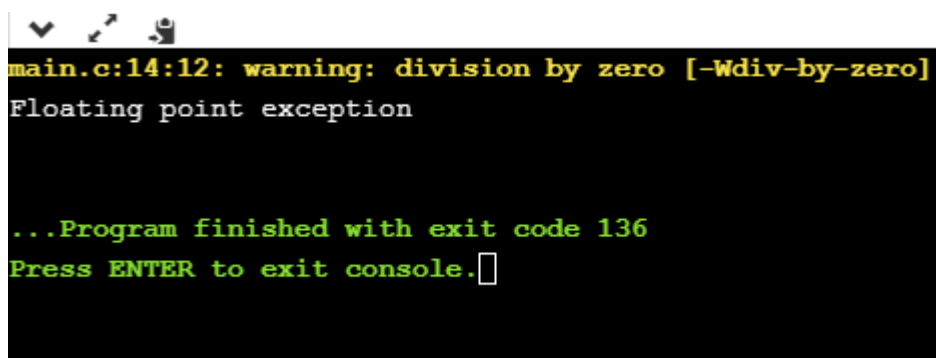
Run-time error

- Sometimes the errors exist during the execution-time even after the successful compilation known as run-time errors.
- When the program is running, and it is not able to perform the operation is the main cause of the run-time error.
- The division by zero is the common example of the run-time error.
- These errors are very difficult to find, as the compiler does not point to these errors.

Let's understand through an example.

```
#include <stdio.h>
int main()
{
    int a=2;
    int b=2/0;
    printf("The value of b is : %d", b);
    return 0;
}
```

Output



```
main.c:14:12: warning: division by zero [-Wdiv-by-zero]
Floating point exception

...Program finished with exit code 136
Press ENTER to exit console.█
```

In the above output, we observe that the code shows the run-time error, i.e., division by zero.

Linker error

- Linker errors are mainly generated when the executable file of the program is not created.
- This can be happened either due to the wrong function prototyping or usage of the wrong header file.
- For example, the **main.c** file contains the **sub()** function whose declaration and definition is done in some other file such as **func.c**. During the compilation, the compiler finds the **sub()** function in **func.c** file, so it generates two object files, i.e., **main.o** and **func.o**.
- At the execution time, if the definition of **sub()** function is not found in the **func.o** file, then the linker error will be thrown.
- The most common linker error that occurs is that we use **Main()** instead of **main()**.

Let's understand through a simple example.

```
#include <stdio.h>
int Main()
{
    int a=78;
    printf("The value of a is : %d", a);
    return 0;
}
```

Output

```
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

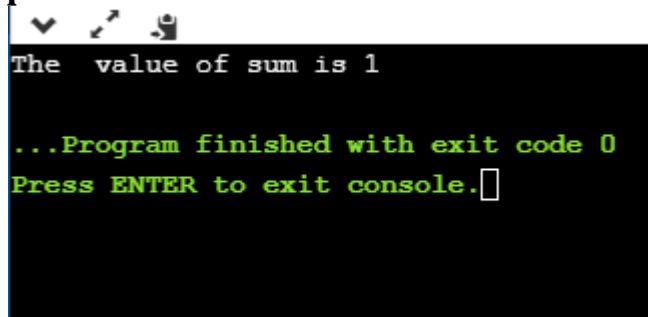
Logical error

- The logical error is an error that leads to an undesired output.
- These errors produce the incorrect output, but they are error-free, known as logical errors.
- These types of mistakes are mainly done by beginners.
- The occurrence of these errors mainly depends upon the logical thinking of the developer.
- If the programmers sound logically good, then there will be fewer chances of these errors.

Let's understand through an example.

```
#include <stdio.h>
int main()
{
    int sum=0; // variable initialization
    int k=1;
    for(int i=1;i<=10;i++); // logical error, as we put the semicolon after loop
    {
        sum=sum+k;
        k++;
    }
    printf("The value of sum is %d", sum);
    return 0;
}
```

Output



In the above code, we are trying to print the sum of 10 digits, but we got the wrong output as we put the semicolon (;) after the for loop, so the inner statements of the for loop will not execute. This produces the wrong output.

Semantic error

- Semantic errors are the errors that occurred when the statements are not understandable by the compiler.

The following can be the cases for the semantic error:

Use of a un-initialized variable.

```
int i;
```

```
i=i+2;
```

Type compatibility

```
int b = "javatpoint";
```

Errors in expressions

```
int a, b, c;
```

```
a+b = c;
```


Array index out of bound

```
int a[10];
```

```
a[10] = 34;
```

Let's understand through an example.

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

```
int main()
```

```
{
```

```
int a,b,c;
```

```
a=2;
```

```
b=3;
```

```
c=1;
```

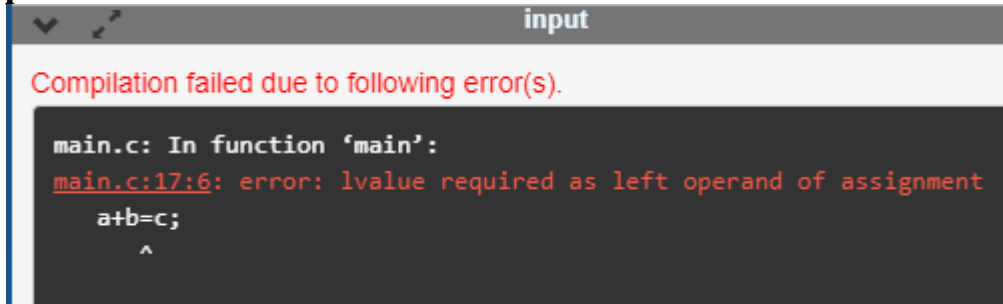
```
a+b=c; // semantic error
```

```
return 0;
```

```
}
```

In the above code, we use the statement **a+b=c**, which is incorrect as we cannot use the two operands on the left-side.

Output

A screenshot of a compiler error message displayed in a window titled "input". The message is in red text on a dark background. It reads: "Compilation failed due to following error(s).", followed by "main.c: In function 'main':", and then "main.c:17:6: error: lvalue required as left operand of assignment". Below this, the code snippet "a+b=c;" is shown with a red caret (^) pointing to the equals sign, indicating the location of the error.

```
input
Compilation failed due to following error(s).
main.c: In function 'main':
main.c:17:6: error: lvalue required as left operand of assignment
a+b=c;
  ^
```