CS101: Problem Solving through C Programming

Sachchida Nand Chaurasia

Assistant Professor

Department of Computer Science
Banaras Hindu University
Varanasi
Email id: snchaurasia@bhu.ac.in, sachchidanand.mca07@gmail.com

January 25, 2021

C Language Introduction I

✓ C is a procedural programming language.

✓ It was initially developed by Dennis Ritchie in the year 1972.

✓ It was mainly developed as a system programming language to write an operating system.

✓ The main features of C language include low-level access to memory, a simple set of keywords, and clean style, these features make C language suitable for system programmings like an operating system or compiler development.

✓ Many later languages have borrowed syntax/features directly or indirectly from C language.

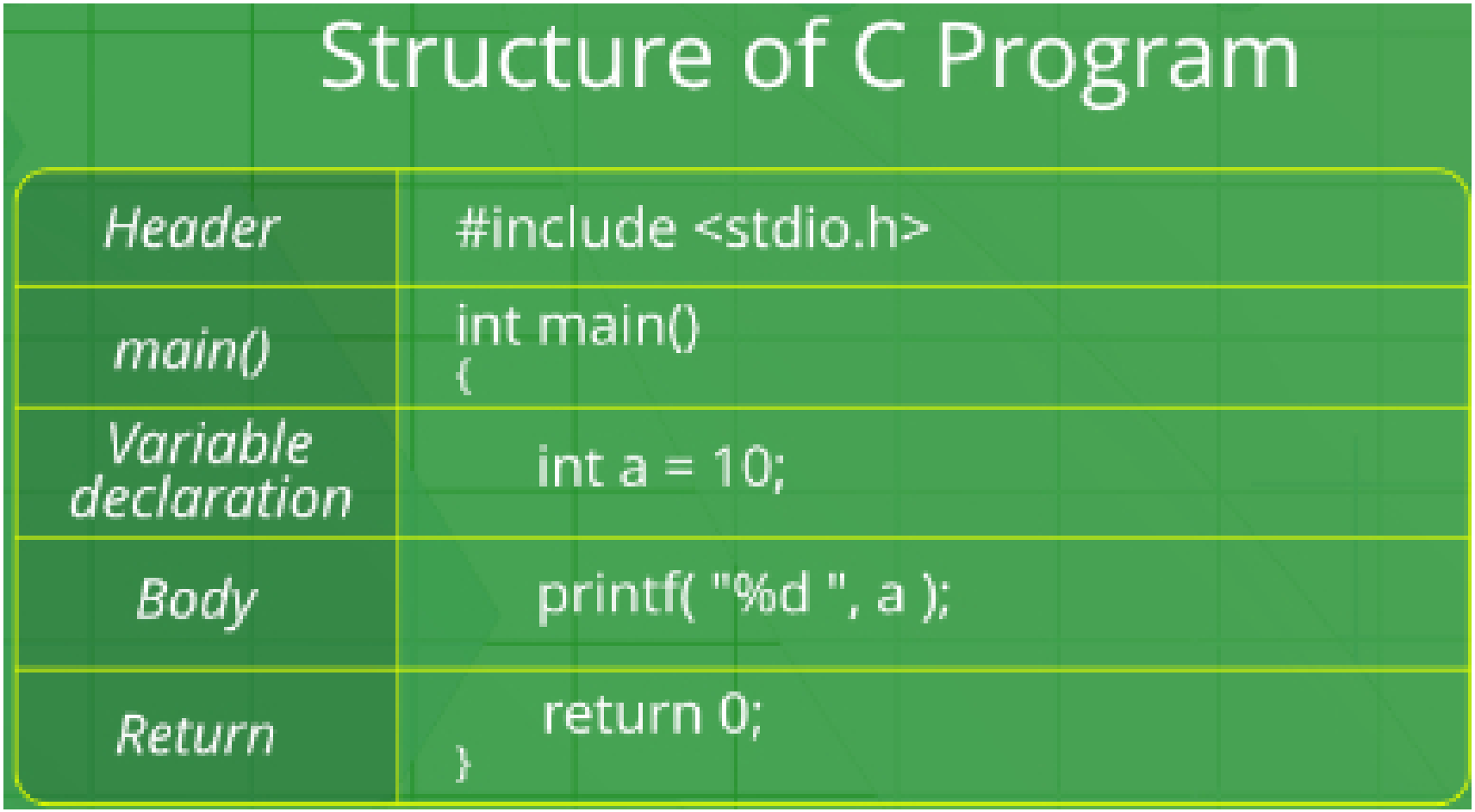
✓ Like syntax of Java, PHP, JavaScript, and many other languages are mainly based on C language.

C Language Introduction II

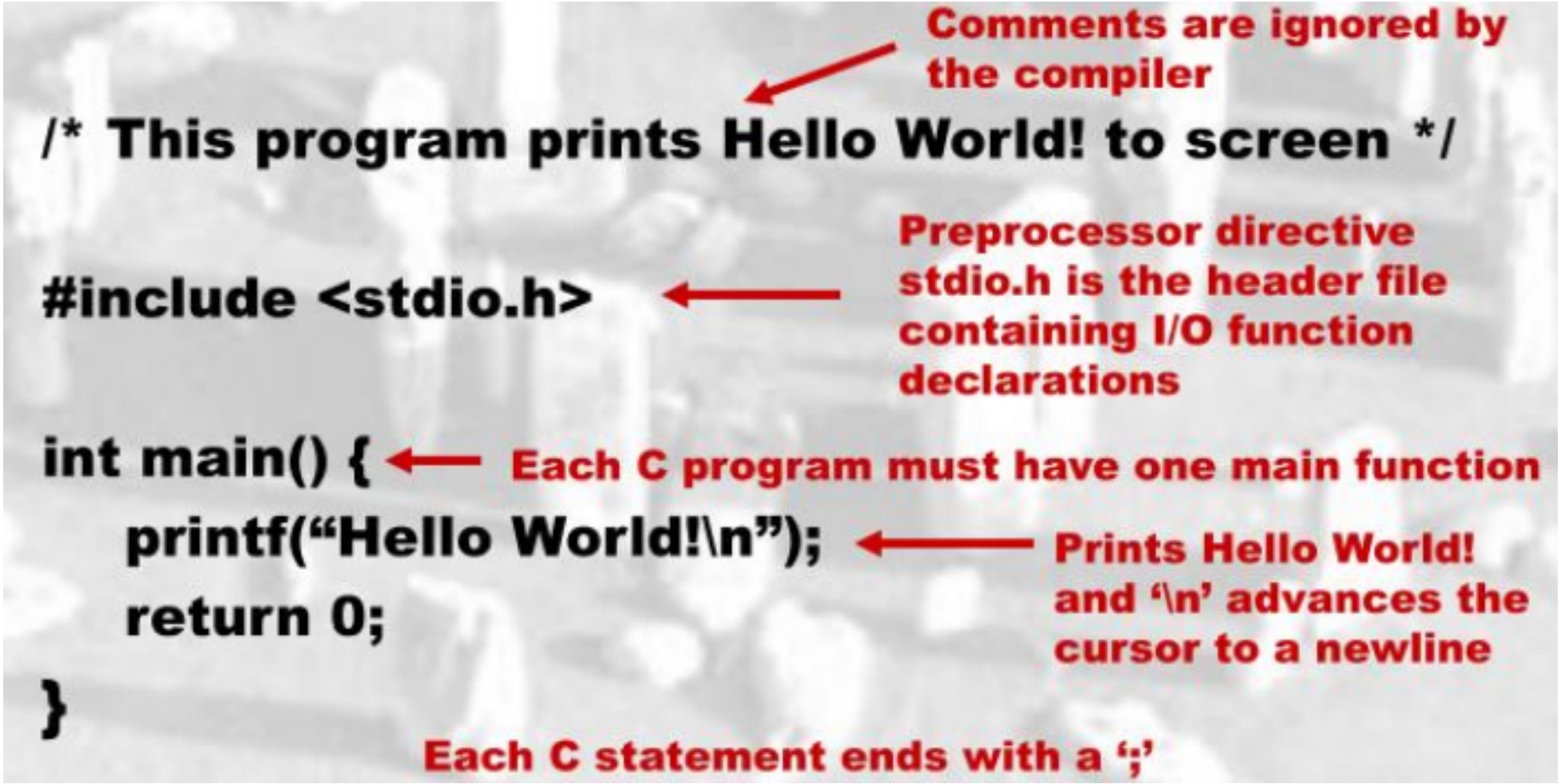
```c
#include <stdio.h>
int main(void)
{
        /* my first program in C */
        printf("Hello, World !");
        return 0;
}
```

```c
#include<stdio.h> //include information about standard library
main(void) //define a Junction named main that receives no argument values statements oj main are enclosed in
      braces
{
      printf("hello, world\n"); //main calls library function printf to print this sequence of characters; \n
      represents the newline character
}
```

C Language Introduction III



C Language Introduction IV



The components of the above structure are:

C Language Introduction V

1. Header Files Inclusion: The first and foremost component is the inclusion of the Header files in a C program.

   ✓ A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files.

   Some of C Header files:

   - stddef.h – Defines several useful types and macros.
   - stdint.h – Defines exact width integer types.
   - stdio.h – Defines core input and output functions
   - stdlib.h – Defines numeric conversion functions, pseudo-random network generator, memory allocation
   - string.h – Defines string handling functions
   - math.h – Defines common mathematical functions

C Language Introduction VI

Syntax to include a header file in C:

```
#include
```

2. Main Method Declaration: The next part of a C program is to declare the main(void) function. The syntax to declare the main function is: Syntax to Declare main method:

```
int main(void)
{

}
```

C Language Introduction VII

3. Variable Declaration: The next part of any C program is the variable declaration. It refers to the variables that are to be used in the function. Please note that in the C program, no variable can be used without being declared. Also in a C program, the variables are to be declared before any operation in the function.

   Example:

```
1 int main(void)
2 {
3             int a;
4         .
5         .
6 }
```

C Language Introduction VIII

4. Body: Body of a function in C program, refers to the operations that are performed in the functions. It can be anything like manipulations, searching, sorting, printing, etc.

   Example:

```
1 int main(void)
2 {
3             int a;
4             printf("%d", a);
5         .
6         .
7 }
```

C Language Introduction IX

5. Return Statement: The last part in any C program is the return statement. The return statement refers to the returning of the values from a function. This return statement and return value depend upon the return type of the function. For example, if the return type is void, then there will be no return statement. In any other case, there will be a return statement and the return value will be of the type of the specified return type. Example:

```
#include <stdio.h>
int main(void)
{
        printf("Hello, World !");
        return 0;
}
```

C Language Introduction X

Let us analyze the program line by line.

Line 1: [#include <stdio.h>]:

✓ In a C program, all lines that start with # are processed by preprocessor which is a program invoked by the compiler.

✓ In a very basic term, preprocessor takes a C program and produces another C program.

✓ The produced program has no lines starting with #, all such lines are processed by the preprocessor.

✓ In the above example, preprocessor copies the preprocessed code of stdio.h to our file.

The .h files are called header files in C. These header files generally contain declaration of functions. We need stdio.h for the function printf() used in the program.

C Language Introduction XI

Line 2 [int main(void)]:

✓ There must to be starting point from where execution of compiled C program begins.

✓ In C, the execution typically begins with first line of main(void).

✓ The void written in brackets indicates that the main doesn't take any parameter.

✓ main(void) can be written to take parameters also.

✓ The int written before main indicates return type of main(void).

✓ The value returned by main indicates status of program termination.

C Language Introduction XII

Line 3 and 6: [{ and }]:

✓ In C language, a pair of curly brackets define a scope and mainly used in functions and control statements like if, else, loops.

✓ All functions must start and end with curly brackets.

C Language Introduction XIII

Line 4 [printf("Hello, World !");]:

✓ printf() is a standard library function to print something on standard output.

✓ The semicolon at the end of printf indicates line termination.

✓ In C, semicolon is always used to indicate end of statement.

C Language Introduction XIV

Line 5 [return 0;]:

✓ The return statement returns the value from main(void).

✓ The returned value may be used by operating system to know termination status of your program.

✓ The value 0 typically means successful termination.

C Language Introduction XV

✓ The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.

✓ The c compilation process converts the source code taken as input into the object code or machine code. The compilation process can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.

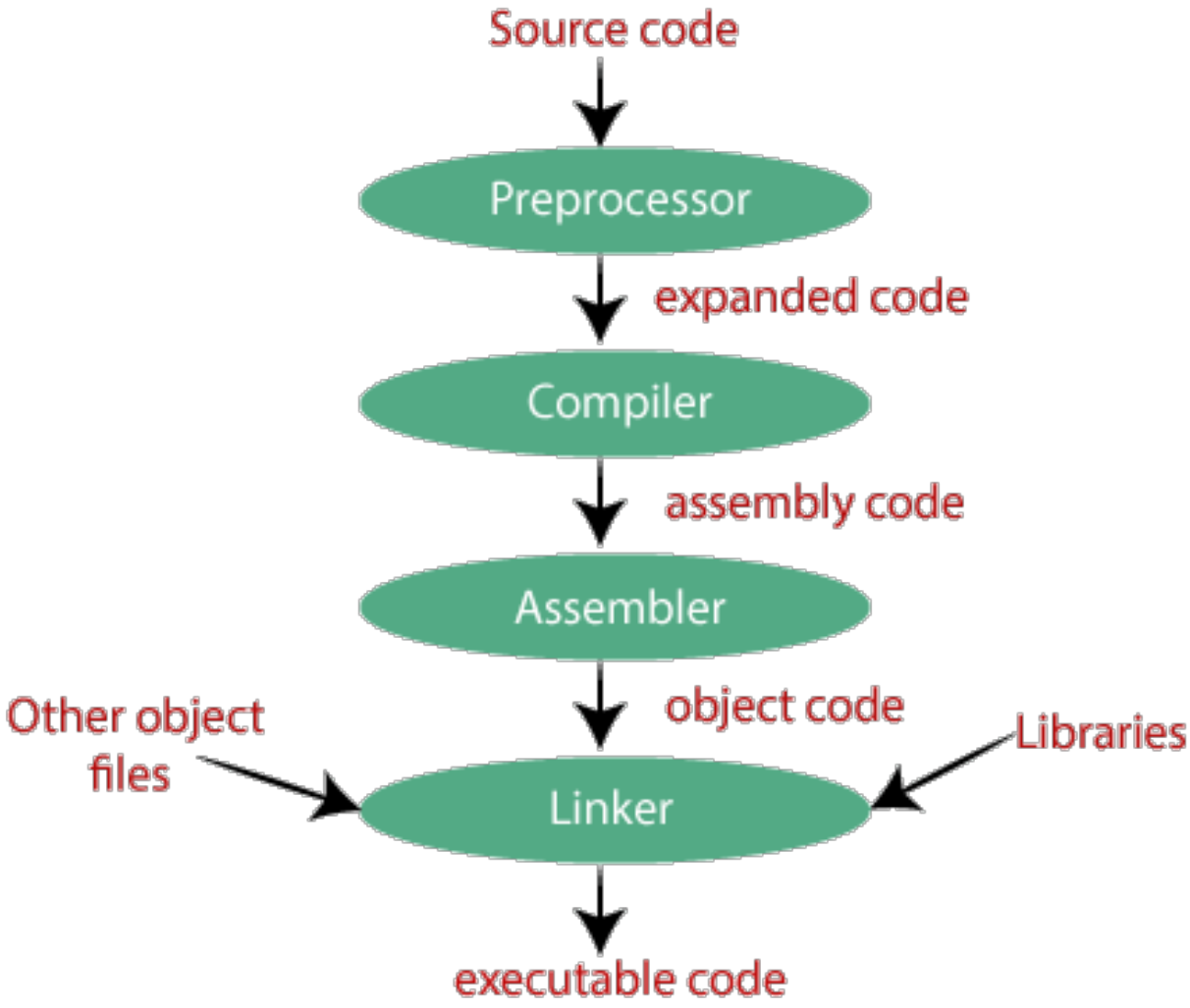
✓ The preprocessor takes the source code as an input, and it removes all the comments from the source code. The preprocessor takes the preprocessor directive and interprets it. For example, if <stdio.h>, the directive is available in the program, then the preprocessor interprets the directive and replace this directive with the content of the 'stdio.h' file.

C Language Introduction XVI

✓ The following are the phases through which our program passes before being transformed into an executable form:

1. Preprocessor
2. Compiler
3. Assembler
4. Linker

C Language Introduction XVII



Preprocessor:

✓ The source code is the code which is written in a text editor and the source code file is given an extension ".c".

✓ This source code is first passed to the preprocessor, and then the preprocessor expands this code.

✓ After expanding the code, the expanded code is passed to the compiler.

Compiler:

✓ The code which is expanded by the preprocessor is passed to the compiler.

✓ The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

Assembler:

✓ The assembly code is converted into object code by using an assembler.

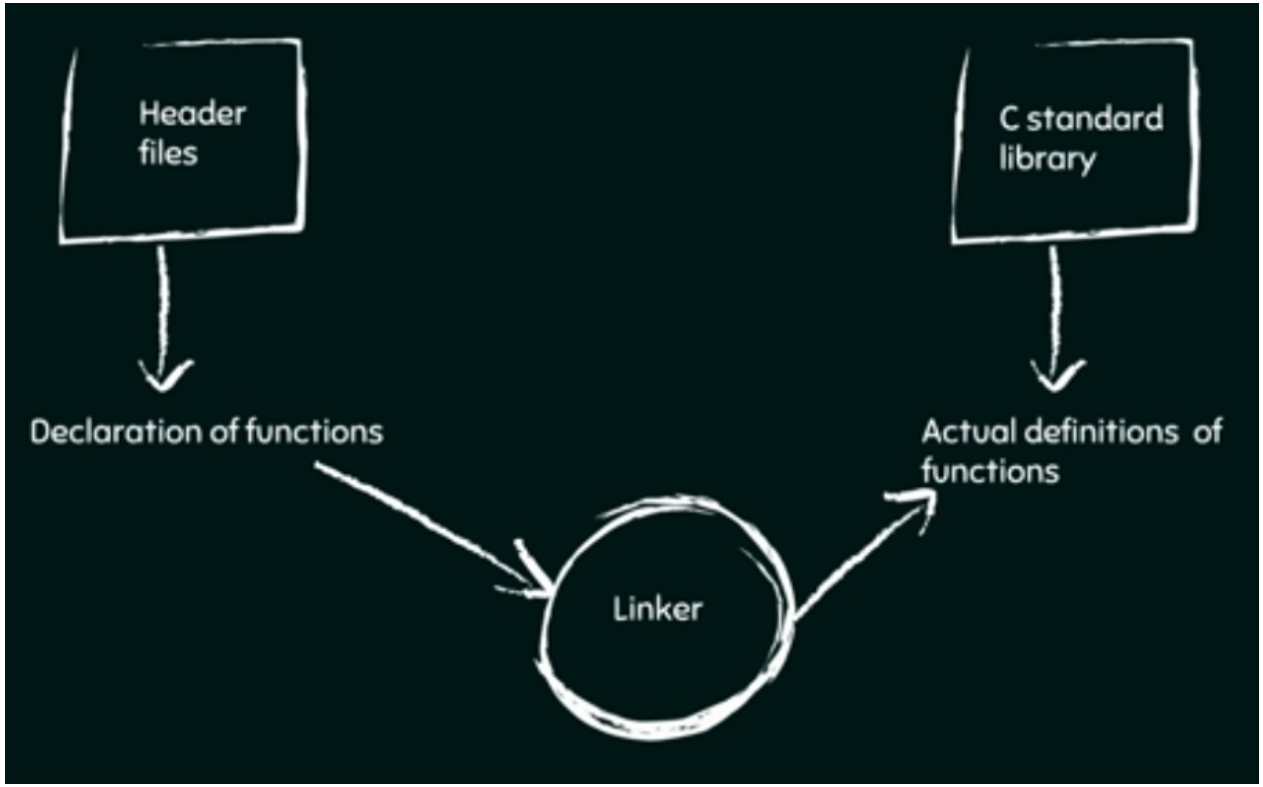
✓ The name of the object file generated by the assembler is the same as the source file.

✓ The extension of the object file in DOS is '.obj,' and in UNIX, the extension is 'o'.

✓ If the name of the source file is 'hello.c', then the name of the object file would be 'hello.obj'.

C Language Introduction XXI

Linker:



C Language Introduction XXII

✓ Mainly, all the programs written in C use library functions.

✓ These library functions are pre-compiled, and the object code of these library files is stored with ’.lib’ (or ’.a’) extension.

✓ The main working of the linker is to combine the object code of library files with the object code of our program.

✓ Sometimes the situation arises when our program refers to the functions defined in other files; then linker plays a very important role in this.

✓ It links the object code of these files to our program.

✓ Therefore, we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files.

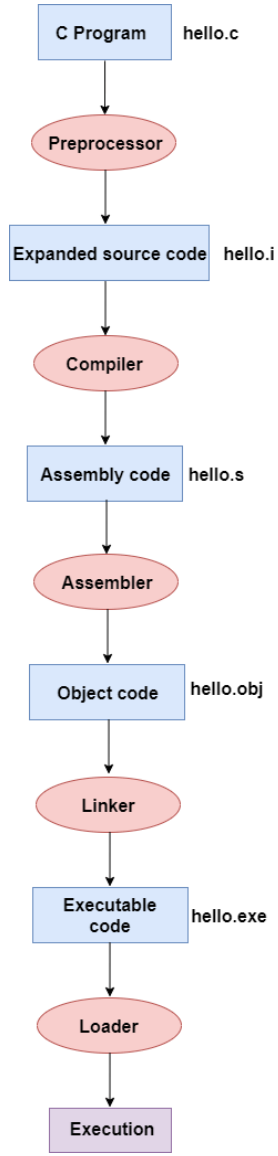
✓ The output of the linker is the executable file.

C Language Introduction XXIII

✓ In DOS, the extension of the executable file is '.exe', and in UNIX, the executable file can be named as 'a.out'.

✓ For example, if we are using printf() function in a program, then the linker adds its associated code in an output file.

C Language Introduction XXIV



C Language Introduction XXV

In the above flow diagram, the following steps are taken to execute a program:

✓ Firstly, the input file, i.e., hello.c, is passed to the preprocessor, and the preprocessor converts the source code into expanded source code.

✓ The extension of the expanded source code would be hello.i.

✓ The expanded source code is passed to the compiler, and the compiler converts this expanded source code into assembly code.

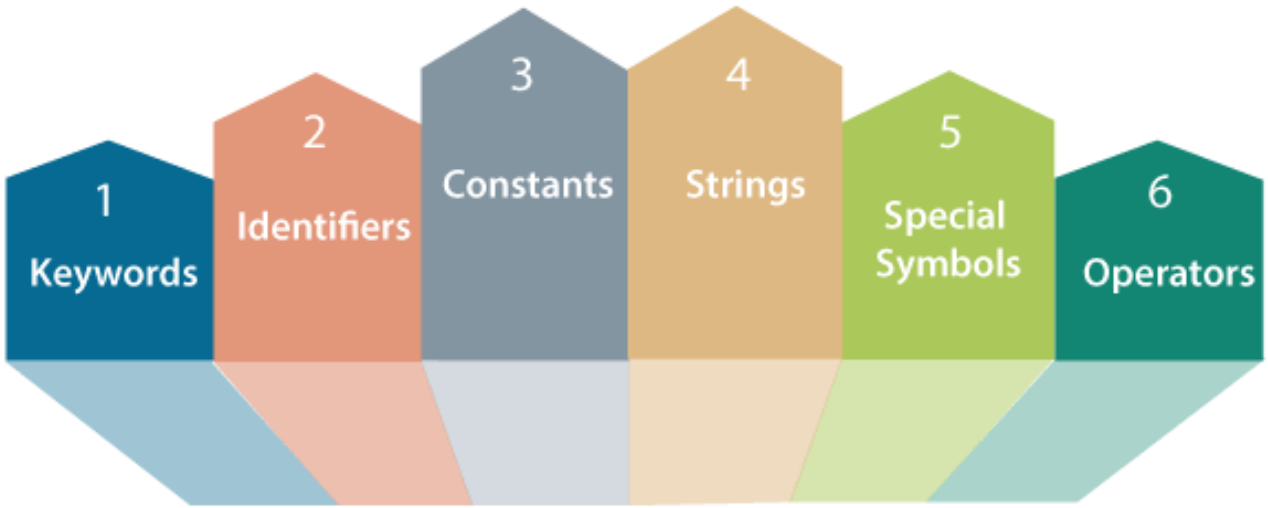
✓ The extension of the assembly code would be hello.s. This assembly code is then sent to the assembler, which converts the assembly code into object code.

✓ After the creation of an object code, the linker creates the executable file.

✓ The loader will then load the executable file for the execution.

C Tokens I

A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:



Classification of C Tokens

1. Keywords

C Tokens II

2. Identifiers
3. Constants
4. Strings
5. Special Symbols
6. Operators

Keywords:

✓ Keywords are pre-defined or reserved words in a programming language.

✓ Each keyword is meant to perform a specific function in a program.

✓ Since keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed.

C Tokens III

✓ You cannot redefine keywords. However, you can specify the text to be substituted for keywords before compilation by using C preprocessor directives.

C Tokens IV

- ✓ C language supports 32 keywords which are given below:

C Tokens V

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

Identifiers:

✓ Identifiers are used as the general terminology for the naming of variables, functions and arrays.

✓ These are user-defined names consisting of an arbitrarily long sequence of letters and digits with either a letter or the underscore(_) as a first character.

✓ Identifier names must differ in spelling and case from any keywords.

✓ You cannot use keywords as identifiers; they are reserved for special use.

✓ Once declared, you can use the identifier in later program statements to refer to the associated value.

✓ A special kind of identifier, called a statement label, can be used in goto statements.

C Tokens VII

There are certain rules that should be followed while naming C identifiers:

- They must begin with a letter or underscore(_).
- They must consist of only letters, digits, or underscore. No other special character is allowed.
- It should not be a keyword.
- It must not contain white space.
- It should be up to 31 characters long as only the first 31 characters are significant.
- main: method name.
- a: variable name.

C Tokens VIII

Constants:

✓ Constants are also like normal variables. But, the only difference is, their values can not be modified by the program once they are defined.

✓ Constants refer to fixed values. They are also called literals.

✓ Constants may belong to any of the data type.

✓ const can be used to declare constant variables. Constant variables are variables which, when initialized, can't change their value. Or in other words, the value assigned to them cannot be modified further down in the program. **Syntax:**

```
1 const data_type var_name = var_value; (or)
2 const data_type *variable_name;
```

C Tokens IX

Types of Constants:

1. Integer constants – Example: 0, 1, 1218, 12482
2. Real or Floating-point constants – Example: 0.0, 1203.03, 30486.184
3. Octal & Hexadecimal constants – Example: octal: $(013)_8 = (11)_{10}$, Hexadecimal: $(013)_{16} = (19)_{10}$
4. Character constants -Example: 'a', 'A', 'z'
5. String constants -Example: "BSCPMKSMK"

Strings:

✓ Strings are nothing but an array of characters ended with a null character ('\0'). This null character indicates the end of the string.

✓ Strings are always enclosed in double-quotes. Whereas, a character is enclosed in single quotes in C and C++.

Declarations for String:

```
1 char string[20] =
2 {
3       'g', 'e', 'e', 'k', 's', 'f', 'o', 'r', 'g', 'e', 'e', 'k', 's', '\0'
4 }
5 ;
6 char string[20] = "BSCPMKSMK";
7 char string [] = "BSCPMKSMK";
```

C Tokens XI

✓ when we declare char as "string[20]", 20 bytes of memory space is allocated for holding the string value.

✓ When we declare char as "string[]", memory space will be allocated as per the requirement during the execution of the program.

Special Symbols:

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.[] () { }, ; * = #

- **Brackets[]:** Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.
- **Parentheses():** These special symbols are used to indicate function calls and function parameters.
- **Braces:** These opening and ending curly braces mark the start and end of a block of code containing more than one executable statement.
- **Comma (,):** It is used to separate more than one statements like for separating parameters in function calls.

C Tokens XIII

- **Colon(:):** It is an operator that essentially invokes something called an initialization list.
- **Semicolon(;):** It is known as a statement terminator. It indicates the end of one logical entity. That's why each individual statement must be ended with a semicolon.
- **Asterisk (*):** It is used to create a pointer variable.
- **Assignment operator(=):** It is used to assign values.
- **Pre-processor (#):** The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.

Operators:

✓ Operators are symbols that trigger an action when applied to C variables and other objects.

✓ The data items on which operators act upon are called operands.

✓ Depending on the number of operands that an operator can act upon, operators can be classified as follows:

- Unary Operators: Those operators that require only a single operand to act upon are known as unary operators.For Example increment and decrement operators.
- Binary Operators: Those operators that require two operands to act upon are called binary operators. Binary operators are classified into :

C Tokens XV

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Unary Operators
6. Bitwise Operators

C Tokens XVI

1. Arithmetic Operators: Arithmetic operators are used to perform arithmetic operations on variables and data.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

Example:

```
1 a + b; a - b;  a * b;  a / b;  a % b;
```

C Tokens XVII

2. Assignment Operators: Assignment operators are used in C to assign values to variables.

Operator	Example	Equivalent to
=	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;

C Tokens XVIII

3. Relational Operators: Relational operators are used to check the relationship between two operands.

```
// check is a is less than b
a < b; // Here, < operator is the relational operator. It checks if a is less than b or not. It returns
       either true or false.
```

Operator	Description	Example
==	Is Equal To	3 == 5 returns **False**
!=	Not Equal To	3 != 5 returns **True**
>	Greater Than	3 > 5 returns **False**
<	Less Than	3 < 5 returns **True**
>=	Greater Than or Equal To	3 >= 5 returns **False**
<=	Less Than or Equal To	3 <= 5 returns **True**

C Tokens XIX

4. Logical Operators: Logical operators are used to check whether an expression is **True** or **False**. They are used in decision making.

Operator	Example	Meaning
&& (Logical AND)	expression1 && expression2	**True** only if both expression1 and expression2 are **True**
|| (Logical OR)	expression1 || expression2	**True** if either expression1 or expression2 is **True**
! (Logical NOT)	!expression	**True** if expression is **False** and vice versa

C Tokens XX

```
1  int num1=12;
2  int num2=15;
3  int num3=0;
4  int val;
5  val = num1 && num2;
6  val = num1 || num2;
7  val = !num1;
8  val = !num3;
9  ----------------------
10 Output: ????
```

C Tokens XXI

5. Unary Operators: Unary operators are used with only one operand. For example, ++ is a unary operator that increases the value of a variable by 1. That is, ++5 will return 6.

Operator	Meaning
+	Unary plus: not necessary to use since numbers are positive without using it
-	Unary minus: inverts the sign of an expression
++	Increment operator: increments value by 1
- -	Decrement operator: decrements value by 1
!	Logical complement operator: inverts the value of a boolean

C Tokens XXII

```
1  int num = 5;
2  // increase num by 1
3  ++num;
4  // decrement num by 1
5  --num;
6  // + as unary operator
7  num = +5;
8  // - as unary operator
9  num = -55;
10 // ! as logical complement
11 num = !num;
12 ----------------------
13 Output: ????
```

C Tokens XXIII

6. Bitwise Operators: Bitwise operators in C are used to perform operations on individual bits.

```
Bitwise complement Operation of 35
35 = 00100011 (In Binary)
~ 00100011
  ________
   11011100  = 220 (In decimal)
Here, ~ is a bitwise operator. It inverts the value of each bit (0 to 1 and 1 to 0).
```

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive XOR
~	Bitwise Complement
«	Left Shift
»	Right Shift

C Tokens XXIV

```c
unsigned int a = 60;   /* 60 = 0011 1100 */
unsigned int b = 13;  /* 13 = 0000 1101 */
int c = 0;
// & bitwise AND
c = a & b;  /* 12 = 0000 1100 */
// | bitwise OR
c = a | b;  /* 61 = 0011 1101 */
// ^ bitwise XOR
c = a ^ b;  /* 49 = 0011 0001 */
// ~ bitwise complement
c = ~a; /*-61 = 1100 0011 */
// << left shift
c = a << 2; /* 240 = 1111 0000 */
// >> right shift
c = a >> 2; /* 15 = 0000 1111 */
```

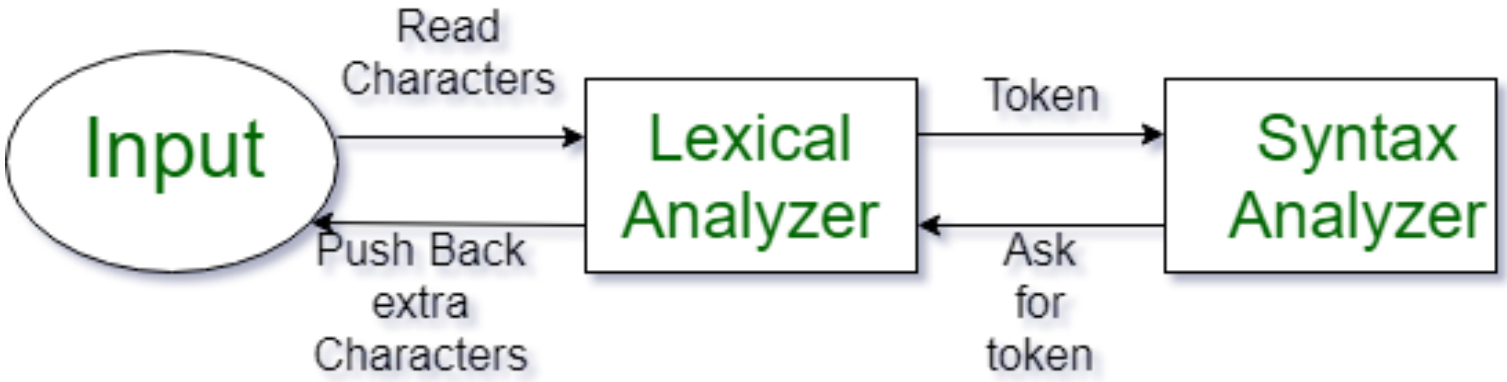
Introduction of Lexical Analyzer I

✓ Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens.

✓ The lexical analyzer is the part of the compiler that detects the token of the program and sends it to the syntax analyzer.

✓ Token is the smallest entity of the code, it is either a keyword, identifier, constant, string literal, symbol. Examples of different types of tokens in C.

Introduction of Lexical Analyzer II



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, . . .)

Introduction of Lexical Analyzer III

- 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.
```

Introduction of Lexical Analyzer IV

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. It always matches the longest character sequence.
2. Tokenization i.e. Dividing the program into valid tokens.
3. Remove white space characters.
4. Remove comments.
5. It also provides help in generating error messages by providing row numbers and column numbers.

Introduction of Lexical Analyzer V

✓ The lexical analyzer(scanner) identifies the error with the help of the 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 refers to it's variable in the symbol table referencing all details
```

For example, consider the program:

```
int a=5; // int a = 5 ; just for understanding.
Tokens:
|int| |a| |=| |5| |; |
```

Another example:

Introduction of Lexical Analyzer VI

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

Introduction of Lexical Analyzer VII

```
26        'return'
27        '0'
28        '; '
29        '
30  }
31  '
```

```
1  //How many tokens?
2
3  printf("BSCPMKSMK");
```

```
1  int main()
2  {
3          int a = 10, b = 20;
4          printf("sum is :%d",a+b);
5          return 0;
6  }
7  ----------
8  //How many tokens?
```

```
1  int max(int i);
2  ------------------
3  //Count number of tokens :
```

Introduction of Lexical Analyzer VIII

- Lexical analyzer first read int and finds it to be valid and accepts as token
- max is read by it and found to be a 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| |)| |; |
```

```
 //Count number of tokens :

printf("i = %d, &i = %x", i, &i);
```

Lvalues and Rvalues in C:

There are two kinds of expressions in C -

lvalue - Expressions that refer to a *memory location* are called "lvalue" expressions. An lvalue may appear as either the left-hand or right-hand side of an assignment operator (=).

Any of the following C expressions can be l-value expressions:

- An identifier of integral, floating, pointer, structure, or union type
- A subscript ([ ]) expression that does not evaluate to an array
- A member-selection expression (-> or .)
- A unary-indirection (*) expression that does not refer to an array
- An l-value expression in parentheses
- A const object (a nonmodifiable l-value)

C Operator Precedence and Associativity II

```c
int g = 20; // valid statement
10 = 20; // invalid statement; would generate compile-time error.
```

```c
// declare a an object of type 'int'
int a;
// a is an expression referring to an 'int' object as l-value
a = 1;
int b = a; // Ok, as l-value can appear on right
// Switch the operand around '=' operator
9 = a;
// Compilation error: as assignment is trying to change the  value of assignment operator
```

```c
char *p ;
short  i;
long l;

(long *) p = &l ;       /* Legal cast   */
(long) i = l ;          /* Illegal cast */
```

C Operator Precedence and Associativity III

rvalue - The term rvalue refers to a *data value* that is stored at some address in memory.

✓ An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right-hand side but not on the left-hand side of an assignment (=).

✓ Variables are lvalues and so they may appear on the left-hand side of an assignment.

✓ Numeric literals are rvalues and so they may not be assigned and cannot appear on the left-hand side.

C Operator Precedence and Associativity IV

```
1  // declare a, b an object of type 'int'
2  int a = 1, b;
3  a + 1 = b; // Error, left expression is  is not variable(a + 1)
4  // declare pointer variable 'p', and 'q'
5  int *p, *q; // *p, *q are lvalue
6  *p = 1; // valid l-value assignment
7  // below is invalid - "p + 2" is not an l-value  p + 2 = 18;
8  q = p + 5; // valid - "p + 5" is an r-value
9  // Below is valid - dereferencing pointer expression gives an l-value
10 *(p + 2) = 18;
11 p = &b;
12 int arr[20]; // arr[12] is an lvalue; equivalent  to *(arr+12)
13 // Note: arr itself is also an lvalue
14 struct S
15 {
16         int m;
17
18 }
19 ;
20 struct S obj; // obj and obj.m are lvalues
21 // ptr-> is an lvalue; equivalent to (*ptr).m
22 // Note: ptr and *ptr are also lvalues
23 struct S* ptr = &obj;
```

Precedence and Associativity of Operators

✓ Precedence of operators come into picture when in an expression we need to decide which operator will be evaluated first.

✓ Operator with higher precedence will be evaluated first.

```
1 int a=1;
2 int b=4;
3 int c;
4 //expression
5 c= a + b;
6 // Which one is correct
7 (c=a) + b or
8 c = (a+b)
```

C Operator Precedence and Associativity VI

***Lower number means higher precedence.**

Precedence	Operator	Type	Associativity
1	( ) [ ] .	Parentheses Array subscript Member selection	Left to Right
2	++ - -	Unary post-increment Unary post-decrement	Left to Right
3	++ - - + - ! ~ (type)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left

C Operator Precedence and Associativity VII

4	* / %	Multiplication Division Modulus	Left to right
5	+ -	Addition Subtraction	Left to right
6	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
7	< <= > >=	Relational less than Relational less than or equal Relational greater than Relational greater than or equal	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right

C Operator Precedence and Associativity VIII

9	&	Bitwise AND	Left to right
10	^	Bitwise exclusive OR	Left to right
11	|	Bitwise inclusive OR	Left to right
12	&&	Logical AND	Left to right
13	||	Logical OR	Left to right
14	? :	Ternary conditional	Right to left
15	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left
16	,	Comma	Left to Right

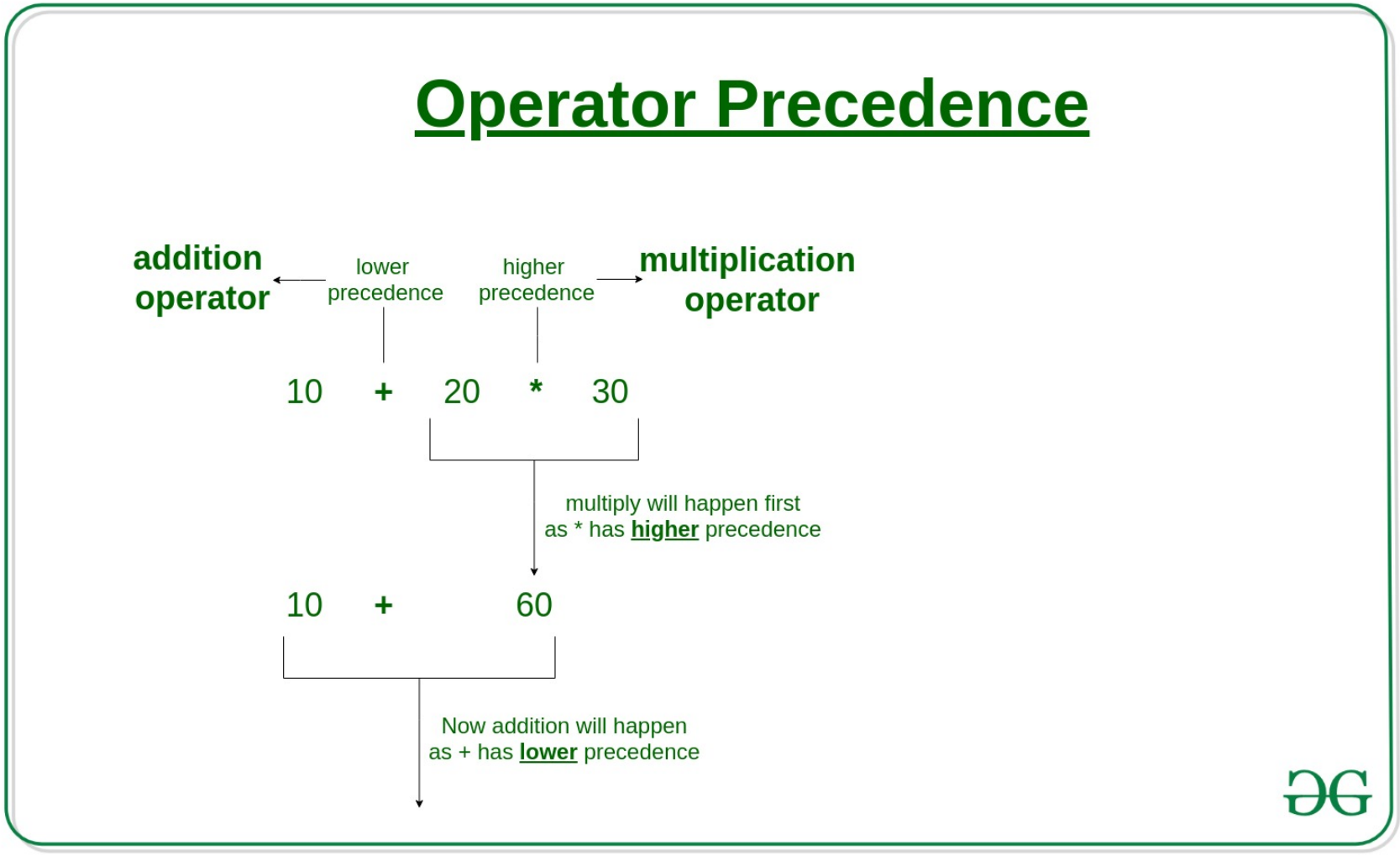
C Operator Precedence and Associativity IX

Operator precedence determines which operator is performed first in an expression with more than one operators with different precedence.

For example: 10 + 20 * 30

```
2+3*5;
(2+3)*5=25 or
2+(3*5)=17
```

C Operator Precedence and Associativity X




```
10 + 20 * 30 is calculated as 10 + (20 * 30)
and not as (10 + 20) * 30
```

C Operator Precedence and Associativity XI

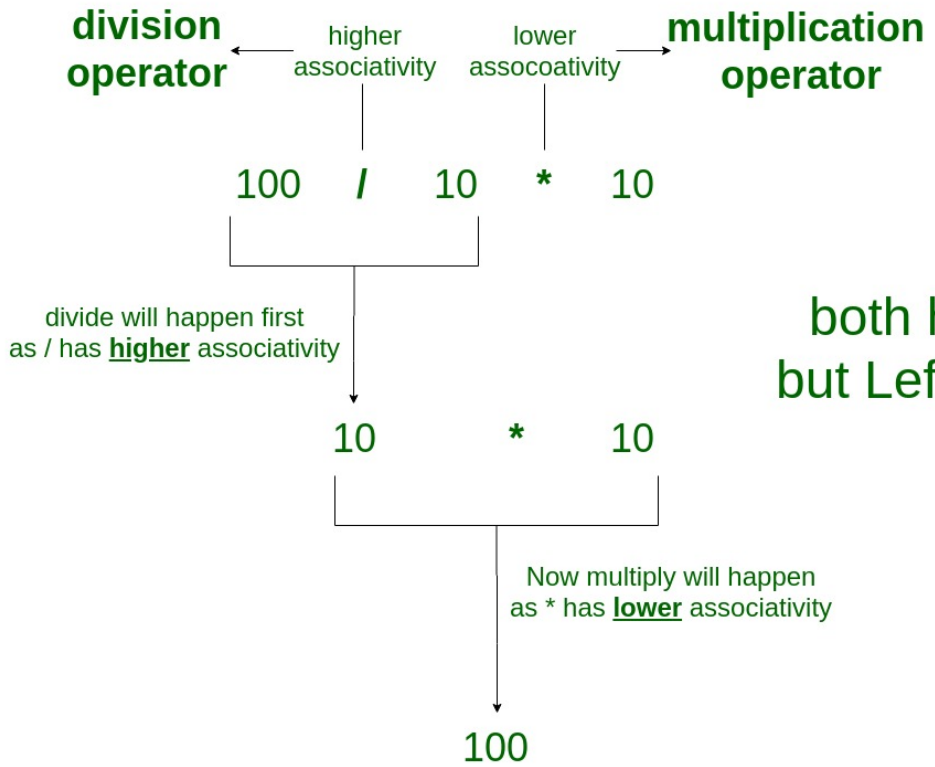
Operators Associativity is used when two operators of same precedence appear in an expression. Associativity can be either Left to Right or Right to Left.

For example: '*' and '/' have same precedence and their associativity is Left to Right.

```
10/2*5;
//if left-to-right
(10/2)*5=25
//if right-to-left
10/(2*5)=1
```

C Operator Precedence and Associativity XII

Operator Associativity



/ and ***
both have the same precedence
but Left to Right (**LTR**) associativity

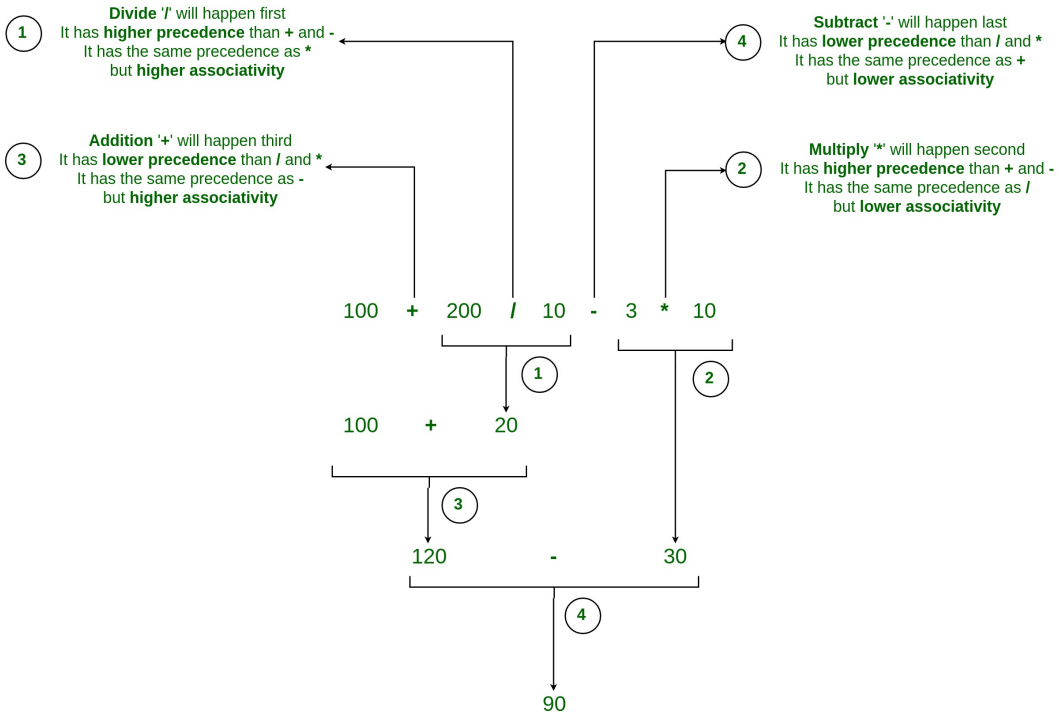
C Operator Precedence and Associativity XIII

Operators Precedence and Associativity are two characteristics of operators that determine the evaluation order of sub-expressions in absence of brackets.

100 + 200 / 10 - 3 * 10

C Operator Precedence and Associativity XIV

Operator Precedence and Associativity



/ and *****
both have the same precedence
but Left to Right (**LTR**) associativity

+ and **-**
both have the same precedence
but Left to Right (**LTR**) associativity

/ and *****
have the higher precedence
than **+** and **-**

C Operator Precedence and Associativity XV

1. Associativity is only used when there are two or more operators of same precedence.
   - ✓ The point to note is associativity doesn't define the order in which operands of a single operator are evaluated.
   - ✓ For example, consider the following program, associativity of the + operator is left to right, but it doesn't mean f1() is always called before f2().

C Operator Precedence and Associativity XVI

```c
// Associativity is not used in the below program.
// Output is compiler dependent.

#include <stdio.h>
int x = 0;
int f1()
{
          x = 5;
          return x;
}
int f2()
{
          x = 10;
          return x;
}
int main()
{
          int p = f1() + f2();
          printf("%d ", x);
          return 0;
}
```

C Operator Precedence and Associativity XVII

2. All operators with the same precedence have same associativity.
   - ✓ This is necessary, otherwise, there won't be any way for the compiler to decide evaluation order of expressions which have two operators of same precedence and different associativity.
   - ✓ For example + and – have the same associativity.
3. Precedence and associativity of postfix ++ and prefix ++ are different.
   - ✓ Precedence of postfix ++ is more than prefix ++, their associativity is also different.
   - ✓ Associativity of postfix ++ is left to right and associativity of prefix ++ is right to left.

C Operator Precedence and Associativity XVIII

4. Comma has the least precedence among all operators and should be used carefully For example consider the following program, the output is 1.

```c
#include <stdio.h>
int main()
{
        int a;
        a = 1, 2, 3; // Evaluated as (a = 1), 2, 3
        printf("%d", a);
        return 0;
}
```

The above program fails in compilation, but the following program compiles fine and prints 1.

C Operator Precedence and Associativity XIX

```c
#include<stdio.h>

int main(void)
{
        int a;
        a = 1, 2, 3;
        printf("%d", a);
        return 0;
}
```

```c
#include<stdio.h>

int main(void)
{
        int a;
        a = (1, 2, 3);
        printf("%d", a);
        return 0;
}
```

C Operator Precedence and Associativity XX

In a C program, comma is used in two contexts: (1) A separator (2) An Operator.

Comma works just as a separator in PROGRAM 1 and we get compilation error in this program.

```c
/* comma as an operator */
int i = (5, 10); /* 10 is assigned to i*/
int j = (f1(), f2()); /* f1() is called (evaluated) first followed by f2().
  The returned value of f2() is assigned to j */
```

```c
/* comma as a separator */
int a = 1, b = 2;
void fun(x, y);
```

```c
/* Comma acts as a separator here and doesn't enforce any sequence.
    Therefore, either f1() or f2() can be called first */
void fun(f1(), f2());
```

C Operator Precedence and Associativity XXI

```c
#include <stdio.h>
int main()
{
        int x = 10;
        int y = 15;

        printf("%d", (x, y));
        getchar();
        return 0;
}
```

```c
#include <stdio.h>
int main()
{
        int x = 10;
        int y = (x++, ++x);
        printf("%d", y);
        getchar();
        return 0;
}
```

C Operator Precedence and Associativity XXII

```c
#include <stdio.h>
int main()
{
        int x = 10, y;

        // The following is equivalent
        // to y = x + 2 and x += 3,
        // with two printings
        y = (x++,
        printf("x = %d\n", x),
        ++x,
        printf("x = %d\n", x),
        x++);

        // Note that last expression is evaluated
        // but side effect is not updated to y
        printf("y = %d\n", y);
        printf("x = %d\n", x);

        return 0;
}
```

C Operator Precedence and Associativity XXIII

5. There is no chaining of comparison operators in C
   In Python, expression like "c > b > a" is treated as "c > b and b > a", but this type of chaining doesn't happen in C. For example consider the following program. The output of following program is "FALSE".

```
1
2  #include <stdio.h>
3  int main()
4  {
5            int a = 10, b = 20, c = 30;
6
7            // (c > b > a) is treated as ((c  > b) > a), associativity of '>'
8            // is left to right. Therefore the value becomes ((30 > 20) > 10)
9            // which becomes (1 > 20)
10           if (c > b > a)
11          printf("TRUE");
12           else
13          printf("FALSE");
14           return 0;
15 }
```

▶ Increment ++ and Decrement - - Operator as Prefix and Postfix

✓ Precedence of Postfix increment/Decrement operator is greater than Prefix increment/Decrement.

✓ Associativity of Postfix is also different from Prefix. Associativity of postfix operators is from left-to-right and that of prefix operators is from right-to-left.

✓ Operators with some precedence have same associativity as well.

- If you use the ++ operator as prefix like: ++var. The value of var is incremented by 1 then, it returns the value. or means first increment then assign it to another variable.

C Operator Precedence and Associativity XXV

- If you use the ++ operator as postfix like: var++. The original value of var is returned first then, var is incremented by 1. or means first assign it to another variable then increment.
- The - - operator works in a similar way like the ++ operator except it decreases the value by 1.

✓ you cannot use rvalue before or after increment/decrement operator.

Example:

(a+b)++; Error

++(a+b); Error.

Error: lvalue required as increment operator(compiler is expecting a variable as an increment operand but we are providing an expression

C Operator Precedence and Associativity XXVI

(a+b) which does not have the capability to store data). Because (a+b) is rvalue. (a+b) is an expression or you can say it is value not an operator.

```
1  int main()
2  {
3          int x = 1;
4          int y=0;
5          x=y++;
6        //  (x=y) = (x=y) +1;
7          scanf("%d",&y);
8          printf("%d\n%d",x,y);
9         // return 0;
10 }
```

✓ Unary operator must be associated with a valid operand.

C Operator Precedence and Associativity XXVII

```c
int main()
{
        int a = 10, b = 5, c = 1;
        printf("%d",a+++b);
        printf("%d",a +++ b);
        printf("%d",a++ + b);
        printf("%d",a +++b);
        printf("%d",a + ++b);
        printf("%d",a+ ++b);
}
```

C Operator Precedence and Associativity XXVIII

```
a+++b;
//Valid tokens in line number 5:
|a| |++| |+| |b| |; |

//Make valid syntax for post-increment and pre-increment
// Unary operator must be associated with a valid operand.
//++ will be associated with a
a++
+
b
-------------
a++ + b;
```

```
int main()
{
        int a = 10, b = 5, c = 1, result;
        result = a-++c-++b;
        printf("%d",result);
}
```

C Operator Precedence and Associativity XXIX

```
result = a-++c-++b;
//Valid tokens in line number 7:
|result|, |=|, |a|, |-|, |++|, |-|, |++| and |b|

//Make valid syntax for post-increment
```

```
1  int main()
2  {
3            int a = 10, b = 15, c = 20, d=25;
4            //int a = 17, b = 15, c = 20, d=25;
5         if(a<= b == d > c)
6         {
7               printf("TRUE");
8         }
9         else
10        {
11              printf("FALSE");
12        }
13 }
```

C Operator Precedence and Associativity XXX

```
1
2  |a| |<| |=| |b| |=| |=| |d| |>| |c|
3  OR
4  |a| |<=| |b| |==| |d| |>| |c|
5
6  |<=| --> Precedence 9
7  |==| --> Precedence 8
8  |>|  --> Precedence 9
9
10 ((a<=b) == (d>c))
11 (1 == 1)
```

Variables in C I

✓ A variable in simple terms is a storage place which has some memory allocated to it.

✓ Basically, a variable used to store some form of data.

✓ Different types of variables require different amounts of memory, and have some specific set of operations which can be applied on them.

A variable is nothing but a name given to a storage area that our programs can manipulate.

✓ Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Variables in C II

Sr. No.	Type & Description
1	char Typically a single octet(one byte). It is an integer type.
2	int The most natural size of integer for the machine.
3	float A single-precision floating point value.
4	double A double-precision floating point value.
5	void Represents the absence of type.

Variables in C III

Variable Declaration:

A typical variable declaration is of the form:

```
  type variable_name; or
  For multiple variables:
  type variable1_name, variable2_name, variable3_name;

```

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

Variables in C IV

Rules for defining variables:

1. A variable can have alphabets (both upper and lower case), digits, and underscore.
2. A variable name can start with the alphabet, and underscore (_) only.
3. It can't start with a digit.
4. No whitespace is allowed within the variable name.
5. A variable name must not be any reserved word or keyword, e.g. int, goto , etc.

Variables in C V

Difference between variable declaration and definition:

✓ Variable declaration refers to the part where a variable is first declared or introduced before its first use.

✓ Variable definition is the part where the variable is assigned a memory location and a value.

✓ Most of the times, variable declaration and definition are done together.

Variables in C VI

Declaration	Definition
A variable or a function can be declared any number of times	A variable or a function can be defined only once
Memory will not be allocated during declaration	Memory will be allocated
int f(int); The above is a function declaration. This declaration is just for informing the compiler that a function named f with return type and argument as int will be used in the function.	int f(int a){ return a; }. The system allocates memory by seeing the above function definition.

Variables in C VII

See the following C program for better clarification:

```c
#include <stdio.h>
int main()
{
          // declaration and definition of variable 'a123'
          char a123 = 'a';
          // This is also both declaration and definition as 'b' is allocated memory and assigned some garbage
       value.
          float b;
          // multiple declarations and definitions
          int _c, _d45, e;
          // Let us print a variable
          printf("%c \n", a123);
          return 0;
         ------------------------
          Output: ???
}

```

Variables in C VIII

Types of Variables in C :

1. Local Variable :
   - ✓ A variable that is declared and used inside the function or block is called local variable.
   - ✓ It's scope is limited to function or block. It cannot be used outside the block.
   - ✓ Local variables need to be initialized before use.

   Example: -

Variables in C IX

```c
#include <stdio.h>
void function()
{
        int x = 10; // local variable
}

int main()
{
        function();
}
```

2. Global Variable :

✓ A variable that is declared outside the function or block is called a global variable.

✓ It is declared at the starting of program. It is available to all the functions.

Example: -

Variables in C X

```c
#include <stdio.h>
int x = 20; //global variable
void function1()
{
        printf("%d\n" , x);
}
void function2()
{
        printf("%d\n" , x);
}
int main()
{
        function1();
        function2();
          return 0;
}
```

In the above code both the functions can use global variable x as we already global variables are accessible by all the functions.

Variables in C XI

3. Static Variable:

✓ A variable that retains its value between multiple function calls is known as static variable.

✓ It is declared with the static keyword.

```
static data_type var_name = var_value;
```

Example:-

```
 #include <stdio.h>
void function()
{
      int x = 20; //local variable
      static int y = 30; //static variable
      x = x + 10;
      y = y + 10;
      printf("\n%d,%d",x,y);
}
int main()
{

        function();
```

Variables in C XII

```
14          function();
15          function();
16          return 0;
17  }
18
```

✓ In C, static variables can only be initialized using constant literals.

For example, following program fails in compilation.

```
1   #include<stdio.h>
2   int initializer(void)
3   {
4               return 50;
5   }
6
7   int main()
8   {
9               static int i = initializer();
10              printf(" value of i = %d", i);
11              getchar();
12              return 0;
13  }
```

Variables in C XIII

If we change the program to following, then it works without any error.

```
#include<stdio.h>
int main()
{
            static int i = 50;
            printf(" value of i = %d", i);
            getchar();
            return 0;
}

```

In the above example , local variable will always print same value whenever function will be called whereas static variable will print the incremented value in each function call.

Variables in C XIV

4. 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.

✓ Automatic variables are similar as local variables.

Example:-

Variables in C XV

```
1   #include <stdio.h>
2  void function()
3  {
4          int x=10; //local variable (also automatic)
5          auto int y=20; //automatic variable
6  }
7  int main()
8  {
9
10           function();
11           return 0;
12 }
13
```

In the above example both x and y are automatic variables .The only difference is that variable y is explicitly declared with auto keyword.

Variables in C XVI

5. External Variable:
   - ✓ External variable can be shared between multiple C files.
   - ✓ We can declare external variable using extern keyword.

Syntax:

```
extern data_type var_name = var_value;

```

Example:

```
 myfile.h
  extern int x=10; //external variable (also global)
----------------------------------------
  program1.c
  #include "myfile.h"
  #include <stdio.h>
  void printValue()
{
        printf("Global variable: %d", global_variable);

}

```

Variables in C XVII

In the above example x is an external variable which is used in multiple files.

6. void:

   void is a special data type. But what makes it so special? void, as it literally means, is an empty data type.

   ✓ It means it has nothing or it holds no value. For example, when it is used as the return data type for a function it simply represents that the function returns no value.

   ✓ Similarly, when its added to a function heading, it represents that the function takes no arguments.

Variables in C XVIII

- 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,

```
1 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,

```
1 int rand(void);
2
```

- Pointers to void A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function

```
1 void *malloc( size_t size);
```

returns a pointer to void which can be casted to any data type.

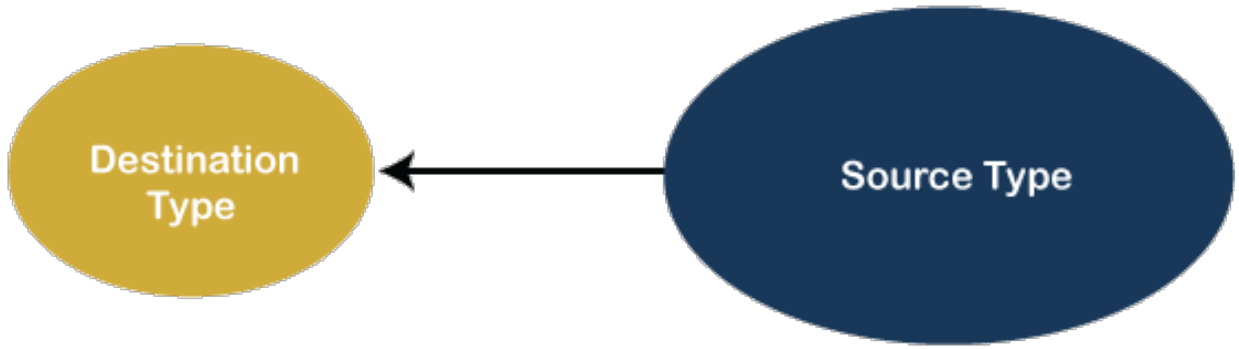
7. typedef:
   typedef is used to give a new name to an already existing or even a custom data type (like a structure).
   ✓ It comes in very handy at times, for example in a case when the name of the structure defined by you is very long or you just need a short-hand notation of a per-existing data type.
   Let's implement the keywords which we have discussed above.
   Take a look at the following code which is a working example to demonstrate these keywords:

Variables in C XX

```
typedef <existing_name> <alias_name>
//Example
typedef unsigned char BYTE;
BYTE  b1, b2;

typedef unsigned int unit;
unit a, b;

```

Type Casting and Type Conversion

1. Type Casting: In typing casting, a data type is converted into another data type by the programmer using the casting operator during the program design. In typing casting, the destination data type may be smaller than the source data type when converting the data type to another data type, that's why it is also called narrowing conversion.



Variables in C XXII

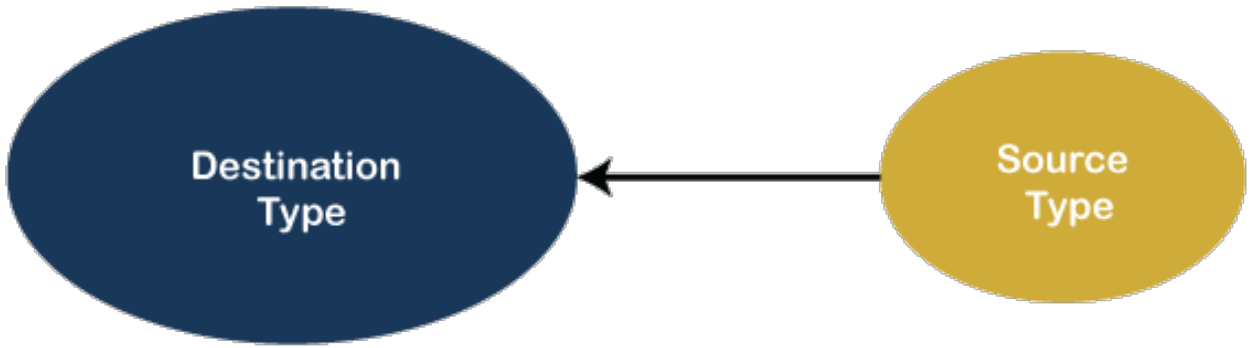
```
Destination_datatype = (target_datatype) variable;
(data_type) it is known as casting operator
(): is a casting operator.
```

```
float b = 3.0;
int a = (int) b; // converting a float value into integer
```

```
#include <stdio.h>

main()
{

        int sum = 17, count = 5;
        double mean;

        mean = (double) sum / count;
        printf("Value of mean : %f\n", mean );
}
//It should be noted here that the cast operator has precedence over division, so the value of sum is
      first converted to type double and finally it gets divided by count yielding a double value.
```

Variables in C XXIII

2. Type conversion : In type conversion, a data type is automatically converted into another data type by a compiler at the compiler time. In type conversion, the destination data type cannot be smaller than the source data type, that's why it is also called widening conversion. One more important thing is that it can only be applied to compatible data types.



Variables in C XXIV

```
1 int a = 20;
2 float b;
3 b = a; // Now the value of variable b is 20.000   /* It defines the conversion of int data type to
      float data type without losing the information. */
```

✓ Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the cast operator. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

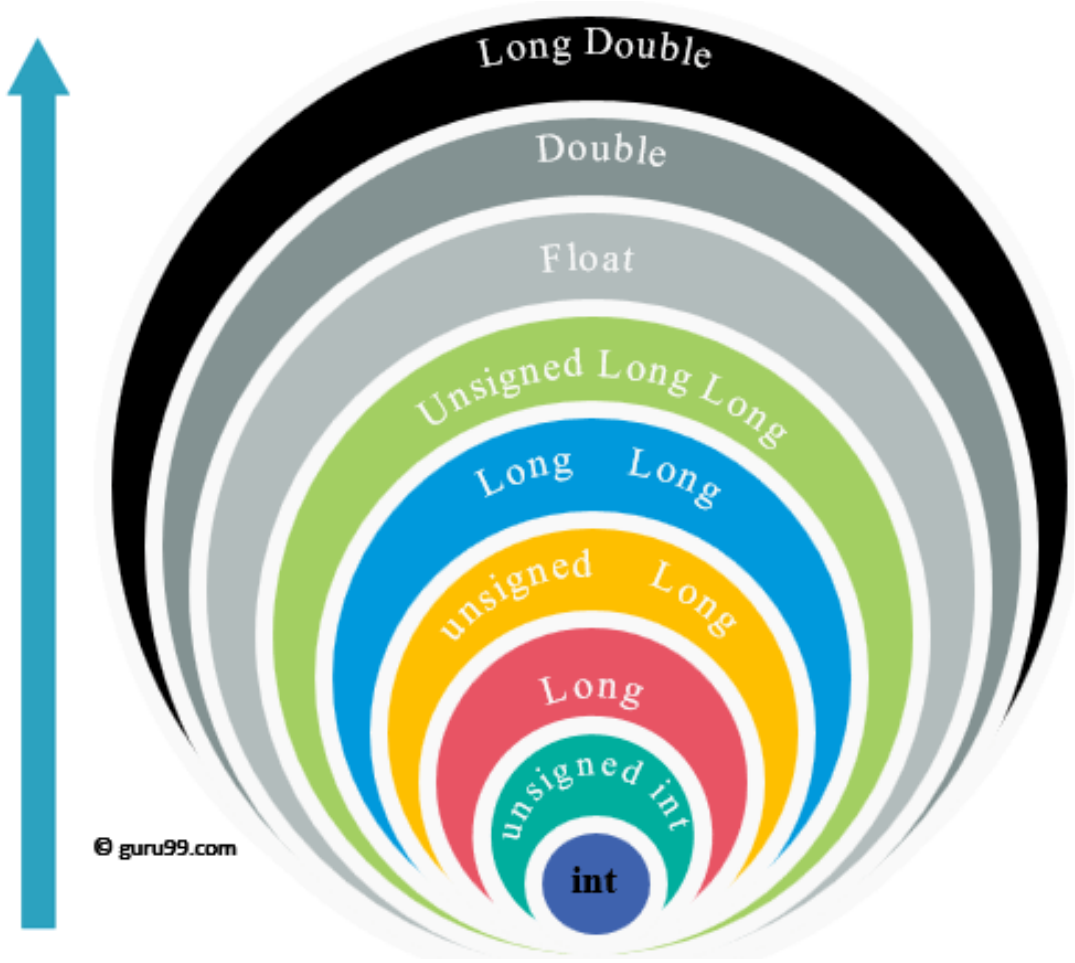
Integer Promotion

Integer promotion is the process by which values of integer type "smaller" than int or unsigned int are converted either to int or unsigned int. Consider an example of adding a character with an integer.

Usual Arithmetic Conversion

The usual arithmetic conversions are implicitly performed to cast their values to a common type. The compiler first performs integer promotion; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy.

Variables in C XXVI



Variables in C XXVII

```c
#include <stdio.h>
main()
{
        int  i = 17;
        char c = 'c'; /* ascii value is 99 */
        int sum;
        sum = i + c;
        printf("Value of sum : %d\n", sum );
}
```

✓ Note: The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && and ||.

S.NO	TYPE CASTING	TYPE CONVERSION
1.	In type casting, a data type is converted into another data type by a programmer using casting operator.	Whereas in type conversion, a data type is converted into another data type by a compiler.
2.	Type casting can be applied to compatible data types as well as incompatible data types.	Whereas type conversion can only be applied to compatible datatypes.
3.	In type casting, casting operator is needed in order to cast the a data type to another data type.	Whereas in type conversion, there is no need for a casting operator.
4.	In typing casting, the destination data type may be smaller than the source data type, when converting the data type to another data type.	Whereas in type conversion, the destination data type can't be smaller than source data type.
5.	Type casting takes place during the program design by programmer.	Whereas type conversion is done at the compile time.