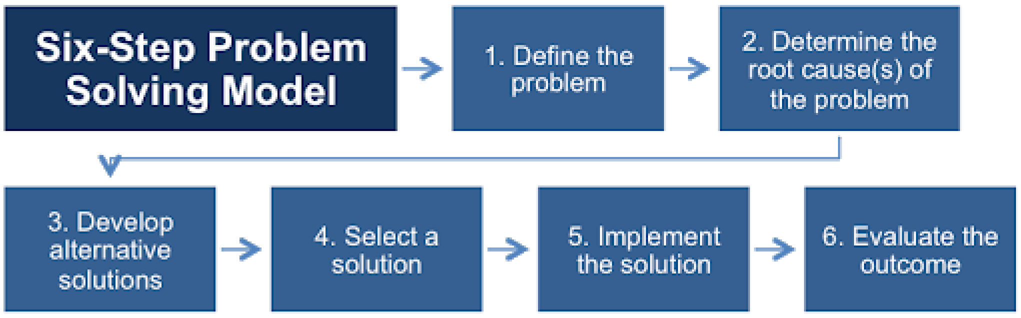
Problem Solving Using C
KCA 102: Session 2023-24

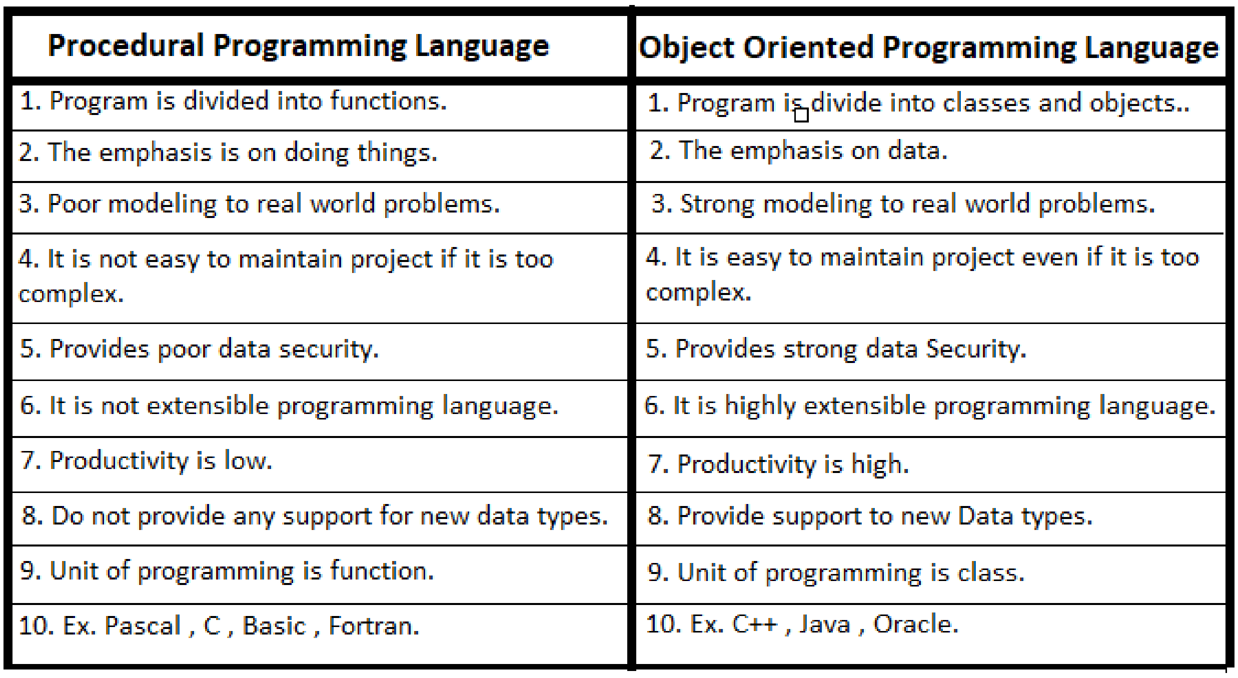
Unit-1	**Lecture-1**	**Topic: Approaches to problem solving and Use of high-level programming language**

Approaches to problem solving



Six-Step Problem Solving Model
1. Define the problem
2. Determine the root cause(s) of the problem
3. Develop alternative solutions
4. Select a solution
5. Implement the solution
6. Evaluate the outcome

Procedure Programming language vs Object Oriented Programming Languages



Procedural Programming Language	Object Oriented Programming Language
1. Program is divided into functions.	1. Program is divide into classes and objects..
2. The emphasis is on doing things.	2. The emphasis on data.
3. Poor modeling to real world problems.	3. Strong modeling to real world problems.
4. It is not easy to maintain project if it is too complex.	4. It is easy to maintain project even if it is too complex.
5. Provides poor data security.	5. Provides strong data Security.
6. It is not extensible programming language.	6. It is highly extensible programming language.
7. Productivity is low.	7. Productivity is high.
8. Do not provide any support for new data types.	8. Provide support to new Data types.
9. Unit of programming is function.	9. Unit of programming is class.
10. Ex. Pascal , C , Basic , Fortran.	10. Ex. C++ , Java , Oracle.

COMPUTER LANGUAGES

To write a program (tells what to do) for a computer, we must use a computer language.

Over the years computer languages have evolved from machine languages to natural languages. The following is the summary of computer languages

1940‘s	--	Machine Languages
1950‘s	--	Symbolic Languages
1960‘s	--	High Level Languages

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language which is made of streams of 0‘s and 1‘s. The instructions in machine language must be in streams of 0‘s and 1‘s. This is also referred as binary digits. These are so named as the machine can directly understood the programs

Advantages

1) High speed execution
2) The computer can understand instructions immediately
3) No translation is needed.

Disadvantages

1) Machine dependent
2) Programming is very difficult
3) Difficult to understand
4) Difficult to write bug free programs
5) Difficult to isolate an error

Example

```
  2      0010
 +3      0011
 ---     ---------------
  5            0101
 ---     ---------------
```

Symbolic Languages (or) Assembly Language

In the early 1950‘s Admiral Grace Hopper, a mathematician and naval officer, developed the concept of a special computer program that would convert programs into machine language. These early programming languages simply mirrored the machine languages using symbols or mnemonics to represent the various language instructions. These languages were known as symbolic languages. Because a computer does not understand symbolic language it must be translated into the machine language. A special program called an **Assembler** translates symbolic code into the machine language. Hence they are called as Assembly language.

Advantages:

1) Easy to understand and use

2) Easy to modify and isolate error
3) High efficiency
4) More control on hardware

Disadvantages:

1) Machine Dependent Language
2) Requires translator
3) Difficult to learn and write programs
4) Slow development time
5) Less efficient

Example:

	2	PUSH 2,A
	3	PUSH 3,B
+		ADD A,B
	5	PRINT C

High-Level Languages

The symbolic languages greatly improved programming efficiency they still required programmers to concentrate on the hardware that they were using working with symbolic languages was also very tedious because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problems being solved led to the development of high-level languages.

High-level languages are portable to many different computers allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer

C	A systems implementation Language
C++	C with object-oriented enhancements
JAVA	Object oriented language for internet and general applications using basic C syntax

Advantages:

1) Easy to write and understand
2) Easy to isolate an error
3) Machine independent language
4) Easy to maintain
5) Better readability
6) Low Development cost

7) Easier to document
8) Portable

Disadvantages:

1) Needs translator
2) Requires high execution time
3) Poor control on hardware
4) Less efficient

Example: C language

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

Difference between Machine, Assembly, High Level Languages

Feature	**Machine**	**Assembly**	**High Level**
Form	0‘s and 1‘s	Mnemonic codes	Normal English
Machine Dependent	Dependent	Dependent	Independent
Translator	Not Needed	Needed(Assembler)	Needed(Compiler)
Execution Time	Less	Less	High
Languages	Only one	Different Manufacturers	Different Languages
Nature	Difficult	Difficult	Easy
Memory Space	Less	Less	More

Unit-1	Lecture-2	Topic: Concept of compiler interpreter, algorithm, flow chart and structured programming

Language Translators

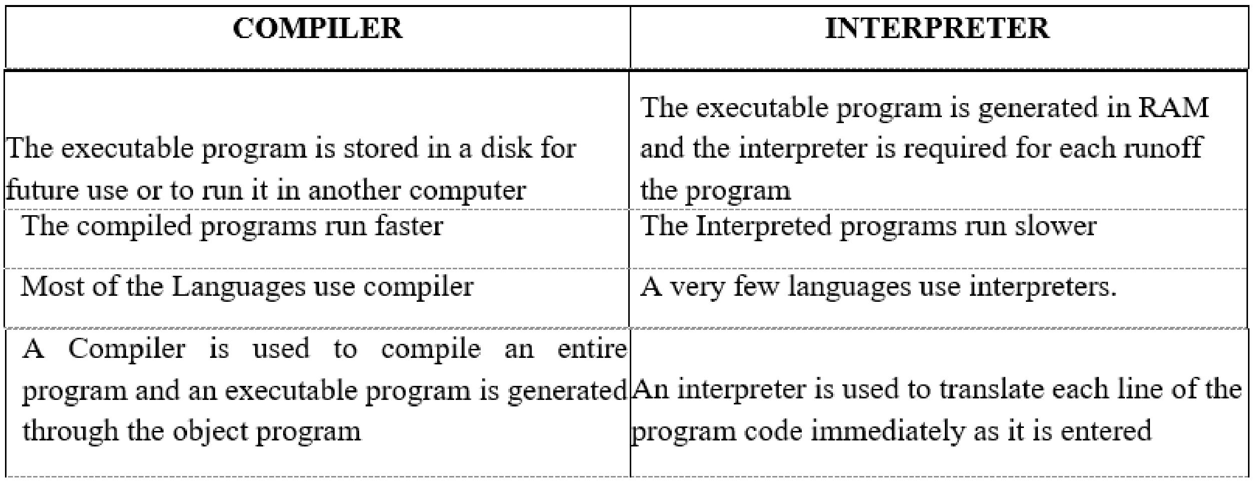
These are the programs which are used for converting the programs in one language into machine language instructions, so that they can be executed by the computer.

1) Compiler: It is a program which is used to convert the high-level language programs into machine language

2) Assembler: It is a program which is used to convert the assembly level language programs into machine language

3) Interpreter: It is a program, it takes one statement of a high-level language program, translates it into machine language instruction and then immediately executes the resulting machine language instruction and so on.

Comparison between a Compiler and Interpreter



COMPILER	INTERPRETER
The executable program is stored in a disk for future use or to run it in another computer	The executable program is generated in RAM and the interpreter is required for each runoff the program
The compiled programs run faster	The Interpreted programs run slower
Most of the Languages use compiler	A very few languages use interpreters.
A Compiler is used to compile an entire program and an executable program is generated through the object program	An interpreter is used to translate each line of the program code immediately as it is entered

ALGORITHM

Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.
We represent an algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.
The ordered set of instructions required to solve a problem is known as an *algorithm*.

The characteristics of a good algorithm are:

- Precision – the steps are precisely stated (defined).
- Uniqueness – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- Finiteness – the algorithm stops after a finite number of instructions are executed.
- Input – the algorithm receives input.
- Output – the algorithm produces output.
- Generality – the algorithm applies to a set of inputs.

Computational Algorithm

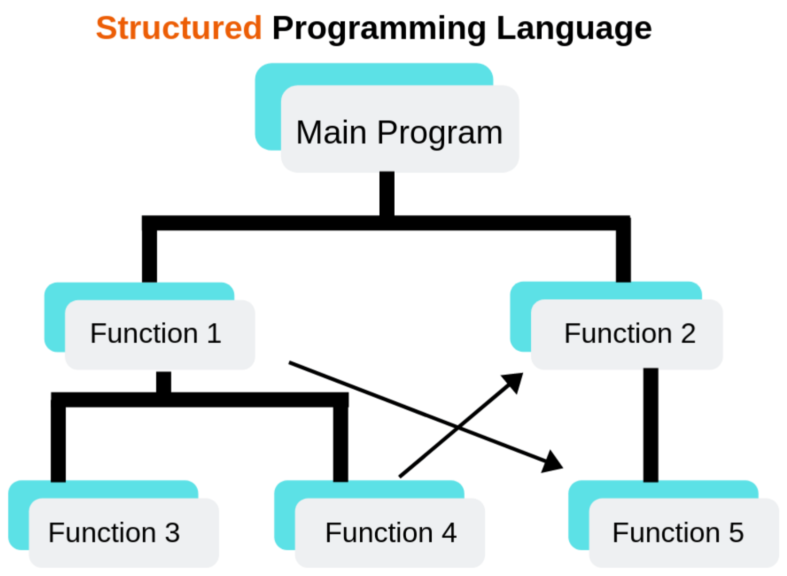
"A set of steps to accomplish or complete a task that is described precisely enough that a computer can run it".

Example

Write an algorithm to find out number is odd or even?

Ans.

```
step 1 : start
step 2 : input number
step 3 : rem=number mod 2
step 4 : if rem=0 then
            print "number even"
         else
            print "number odd"
         endif
step 5 : stop
```

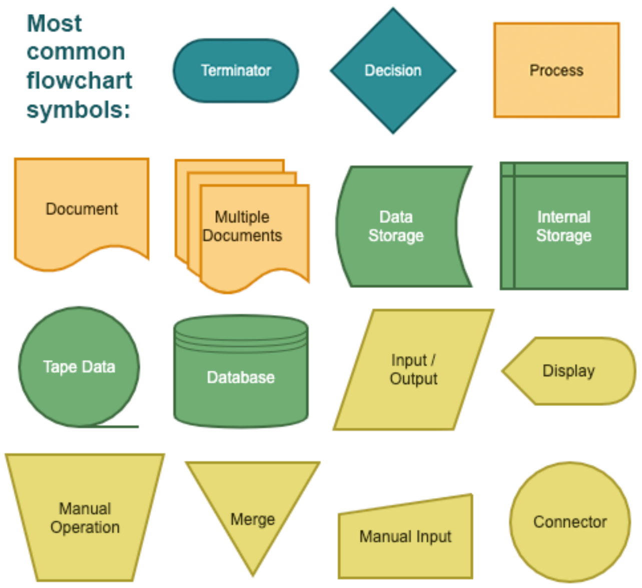


FLOWCHART

Flowchart is a diagrammatic representation of an algorithm. Flowchart is very helpful in writing program and explaining program to others.

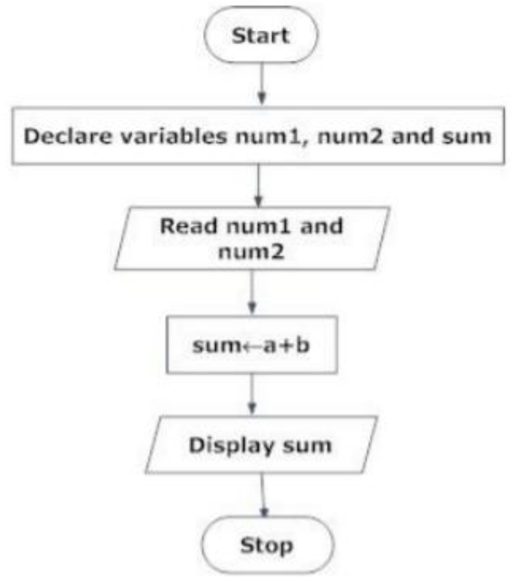
Symbols Used in Flowchart

Different symbols are used for different states in flowchart, For example: Input/Output and decision making has different symbols. The table below describes all the symbols that are used in making flowchart

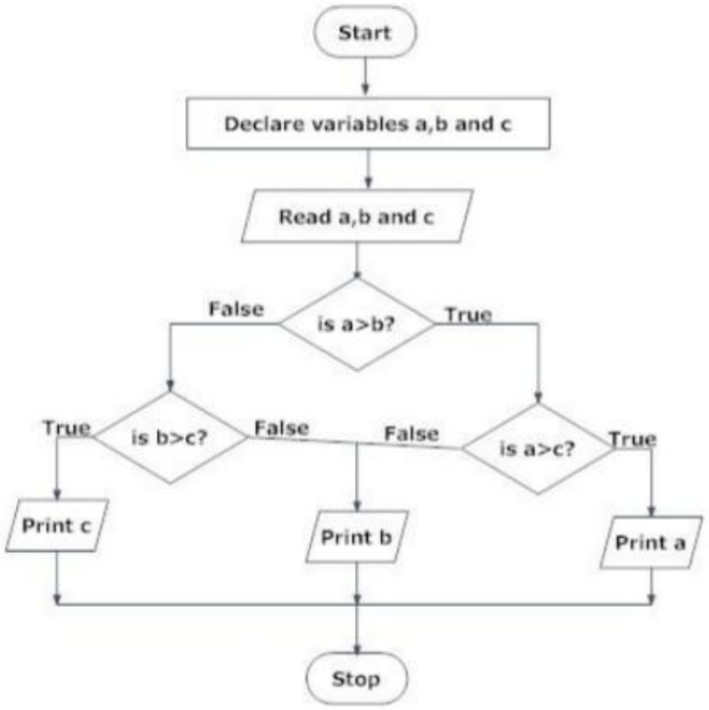


Examples

Draw a flowchart to add two numbers entered by user.



Draw flowchart to find the largest among three different numbers entered by user.



Unit-1	**Lecture-3**	**Topic: C history, salient features of C, Structure of C program and compiling of C program**

HISTORY TO C LANGUAGE

C is a general-purpose language which has been closely associated with the**UNIX**operatingsystem for which it was developed - since the system and most of the programs that run it are written in C.
Many of the important ideas of C stem from the language **BCPL,** developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language **B**, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a **DEC**PDP-7. **BCPL** and **B** are "type less" languages whereas C provides a variety of data types.
In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C ProgrammingLanguage by Kernighan & Ritchie caused a revolution in the computing world.
In 1983, the American National Standards Institute (ANSI) established a committee to provide amodern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.
The Unix operating system and virtually all Unix applications are written in the C language. C has now become a widely used professional language for various reasons.

- Easy to learn
- Structured language
- It produces efficient programs.
- It can handle low-level activities.
- It can be compiled on a variety of computers.

Facts about C

- C was invented to write an operating system called UNIX.
- C is a successor of B language which was introduced around 1970
- The language was formalized in 1988 by the American National Standard Institute (ANSI).
- By 1973 UNIX OS almost totally written in C.
- Today C is the most widely used System Programming Language.
- Most of the state-of-the-art software have been implemented using C

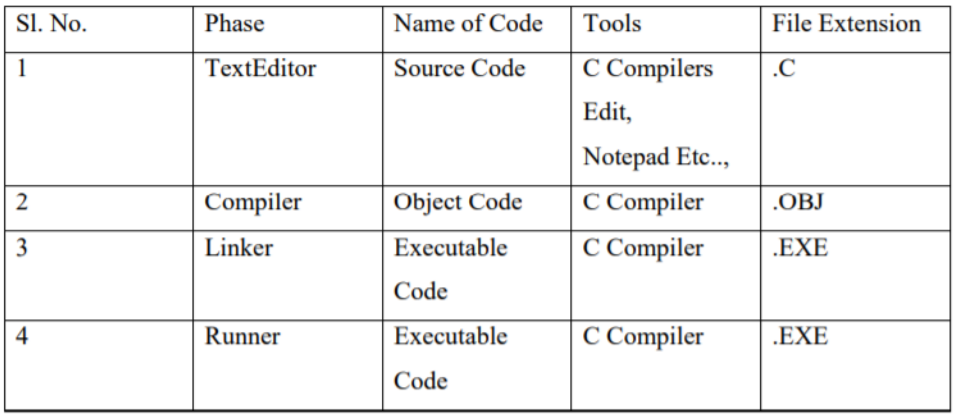
Some examples of the use of C might be:

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Print Spoolers
- Network Drivers
- Modern Programs
- Data Bases
- Language Interpreters
- Utilities

CREATING AND RUNNING PROGRAMS

The procedure for turning a program written in C into machine Language. The process is presented in a straightforward, linear fashion but you should recognize that these steps are repeated many times during development to correct errors and make improvements to the code. The following are the four steps in this process

1) Writing and Editing the program
2) Compiling the program
3) Linking the program with the required modules
4) Executing the program



Sl. No.	Phase	Name of Code	Tools	File Extension
1	TextEditor	Source Code	C Compilers Edit, Notepad Etc..,	.C
2	Compiler	Object Code	C Compiler	.OBJ
3	Linker	Executable Code	C Compiler	.EXE
4	Runner	Executable Code	C Compiler	.EXE

Writing and Editing Programs

The software used to write programs is known as a text editor. A text editor helps us enter, change and store character data. Once we write the program in the text editor we save it using a filename stored with an extension of .C. This file is referred as source code file.

Compiling Programs

The code in a source file stored on the disk must be translated into machine language. This is the job of the compiler. The Compiler is a computer program that translates the source code written in a high-level language into the corresponding object code of the low-level language. This translation process is called *compilation.* The entire high level program is converted into the executable machine code file. The Compiler which executes C programs is called as C Compiler. Example Turbo C, Borland C, GC etc.,

The C Compiler is actually two separate programs:

The Preprocessor
The Translator

The Preprocessor reads the source code and prepares it for the translator. While preparing the code, it scans for special instructions known as preprocessor commands. These commands tell the preprocessor to look for special code libraries. The result of preprocessing is called the translation unit.

After the preprocessor has prepared the code for compilation, the translator does the actual work of converting the program into machine language. The translator reads the translation unit and writes the resulting object module to a file that can then be combined with other precompiled units to form the final program. An object module is the code in the machine language.

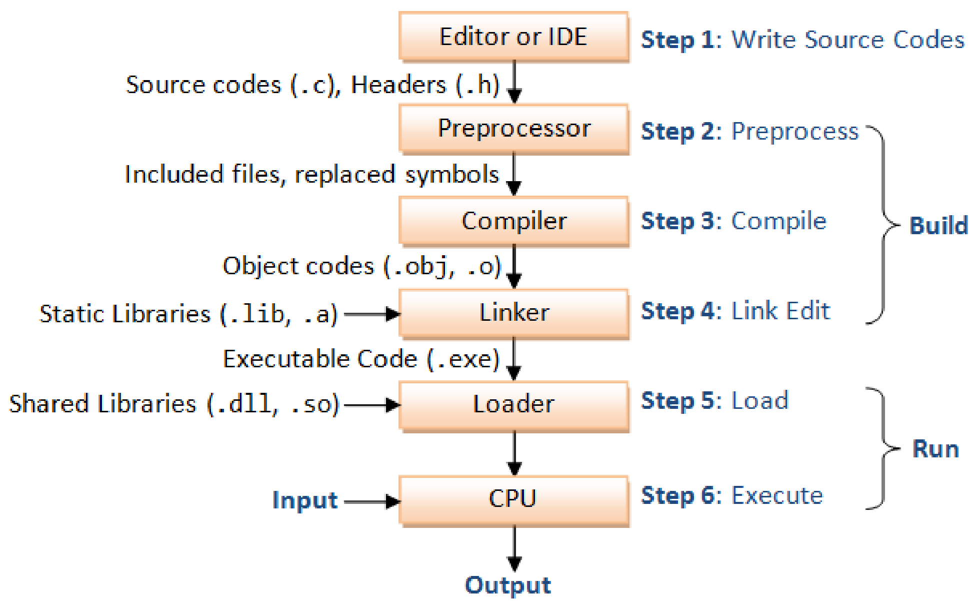
Linking Programs

The Linker assembles all functions, the program‘s functions and system‘s functions into one executable program.

Executing Programs

To execute a program, we use an operating system command, such as run, to load the program into primary memory and execute it. Getting the program into memory is the function of an operating system program known as the **loader.** It locates the executable program andreads it into memory. When everything is loaded the program takes control and it begin execution

PROCESS OF COMPILING AND RUNNING C PROGRAM



BASIC STRUCTURE OF C PROGRAMMING

Documentation section
Link section
Definition section
Global declaration section
main () Function section
{
Declaration part
Executable part
}
Subprogram section
Function 1
Function 2
(User defined functions)

1. **Documentation section:** The documentation section consists of a set of comment linesgiving the name of the program, the author and other details, which the programmer would like to use later.

2. **Link section:** The link section provides instructions to the compiler to link functionsfrom the system library such as using the *#include directive.*

3. **Definition section:** The definition section defines all symbolic constants such usingthe *#define directive.*

4. **Global declaration section:** There are some variables that are used in more than onefunction. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the *user-defined functions.*

5. **main () function section:** Every C program must have one main function section. Thissection contains two parts; declaration part and executable part

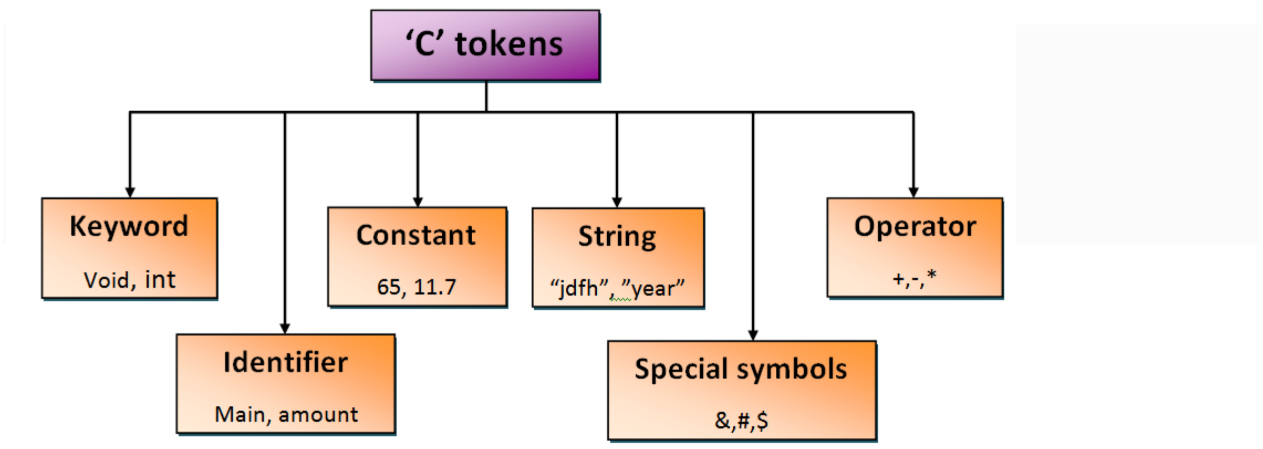
   1. **Declaration part:** The declaration part declares all the*variables*usedin theexecutable part.

   2. **Executable part:** There is at least one statement in the executable part. These twoparts must appear between the opening and closing braces. The *programexecution*beginsat the opening brace and ends at the closing brace. The closingbrace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.

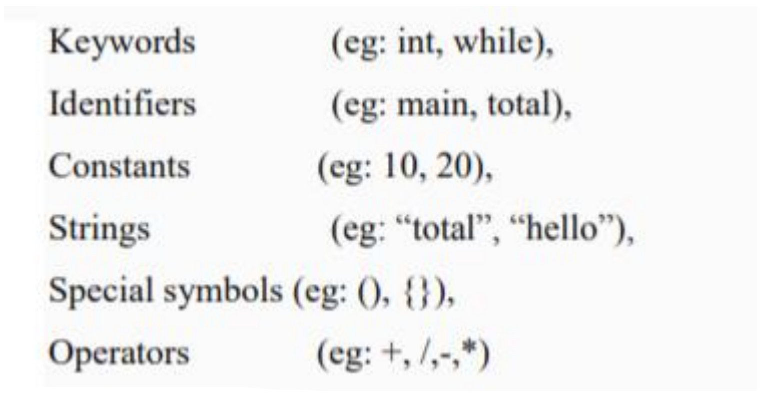
6. **Subprogram section:** If the program is a*multi-function program*thenthe subprogramsection contains all the *user-defined functions*that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

Unit-1	Lecture-4	Topic: Character Set, Token, Keywords, identifiers,constants

C TOKENS



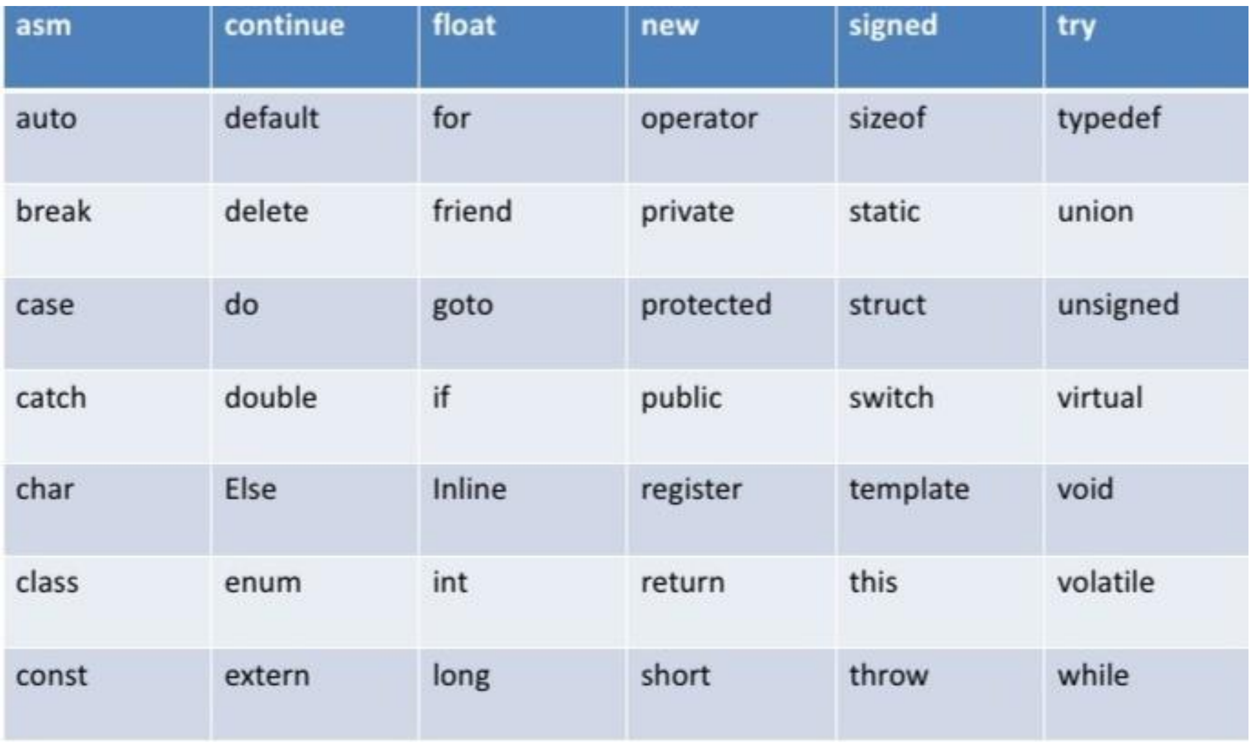
C tokens are the basic buildings blocks in C language which are constructed together to write a C program.Each and every smallest individual unit in a C program is known as C tokens.C tokens are of six types. They are



Keywords	(eg: int, while),
Identifiers	(eg: main, total),
Constants	(eg: 10, 20),
Strings	(eg: "total", "hello"),
Special symbols	(eg: (), {}),
Operators	(eg: +, /,-,*)

C KEYWORDS

C keywords are the words that convey a special meaning to the c compiler. The keywords cannot be used as variable names. The list of C keywords is given below:



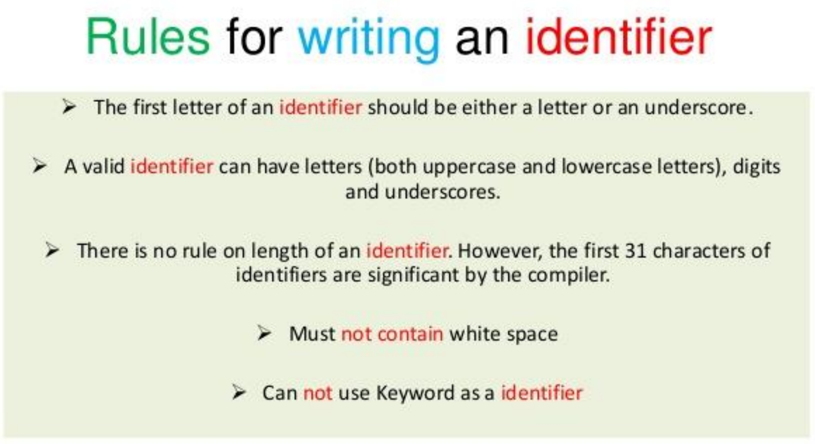
asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	Else	Inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

C IDENTIFIERS

Identifiers are used as the general terminology for the names of variables, functions and arrays. These are user defined names consisting of arbitrarily long sequence of letters and digits with either a letter or the underscore(__) as a first character.

- 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 first 31 characters are significant.

Some examples of c identifiers:



Rules for writing an identifier

- The first letter of an identifier should be either a letter or an underscore.
- A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
- There is no rule on length of an identifier. However, the first 31 characters of identifiers are significant by the compiler.
- Must not contain white space
- Can not use Keyword as a identifier

Valid/Invalid Identifiers

Valid	Invalid
sum	7of9
c4_5	x-name
A_NUMBER	name with spaces
longnamewithmanychars	1234a
TRUE	int
_split_name	AXYZ&

CONSTANTS

A C constant refers to the data items that do not change their value during the program execution. Several types of C constants that are allowed in C are:

Integer Constants

Integer constants are whole numbers without any fractional part. It must have at least one digit and may contain either + or – sign. A number with no sign is assumed to be positive. There are three types of integer constants:

Decimal Integer Constants

Integer constants consisting of a set of digits, 0 through 9, preceded by an optional – or + sign.
Example of valid decimal integer constants
341, -341, 0, 8972

Octal Integer Constants

Integer constants consisting of sequence of digits from the set 0 through 7 starting with 0 is said to be octal integer constants.
Example of valid octal integer constant
  10, 0424, 0, 0540

Hexadecimal Integer Constants

Hexadecimal integer constants are integer constants having sequence of digits preceded by 0x or 0X. They may also include alphabets from A to F representing numbers 10 to 15. Example of valid hexadecimal integer constants

0xD, 0X8d, 0X, 0xbD

It should be noted that, octal and hexadecimal integer constants are rarely used in programming.

Real Constants

The numbers having fractional parts are called real or floating-point constants. These may be represented in one of the two forms called *fractional form* or the *exponent form* and may also have either + or – sign preceding it.

Example of valid real constants in fractional form or decimal notation
0.05, -0.905, 562.05, 0.015

Representing a real constant in exponent form

The general format in which a real number may be represented in exponential or scientific form is

mantissa e exponent

The mantissa must be either an integer or a real number expressed in decimal notation. The letter e separating the mantissa and the exponent can also be written in uppercase i.e. E And, the exponent must be an integer.

Examples of valid real constants in exponent form are:

252E85, 0.15E-10, -3e+8

Character Constants

A character constant contains one single character enclosed within single quotes.
Examples of valid character constants

‗a‘ , ‗Z‘, ‗5‘

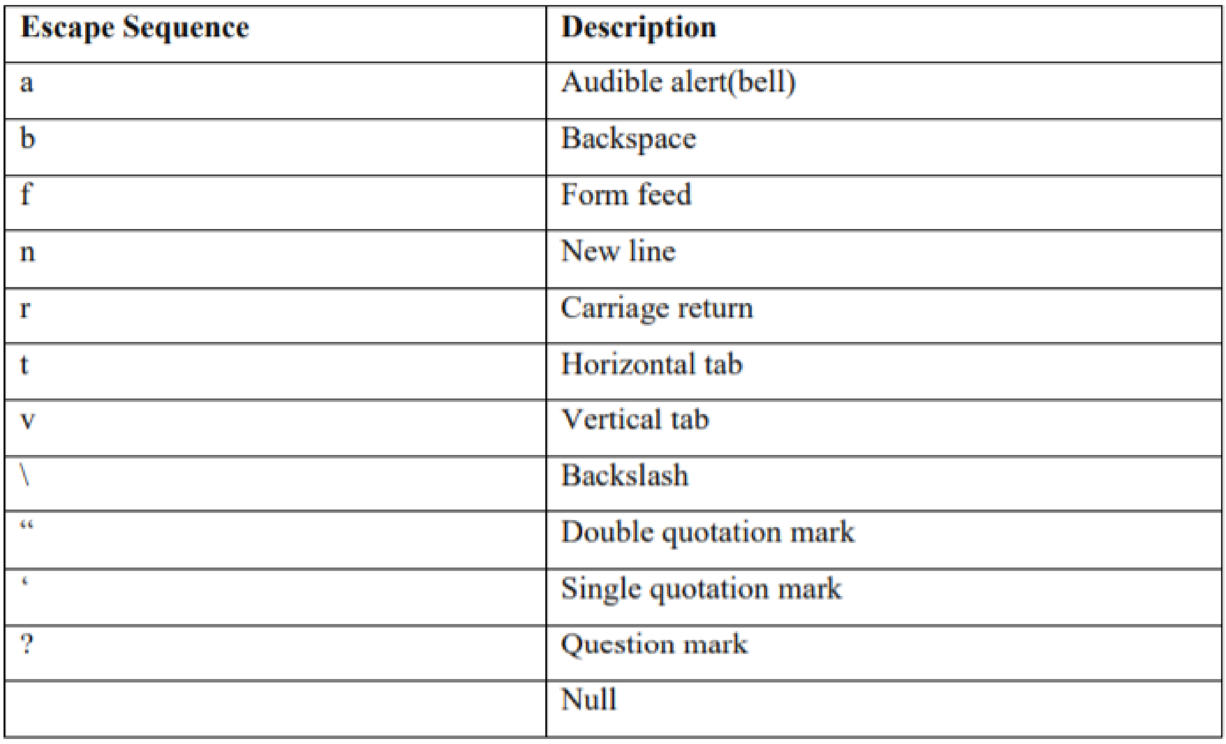
It should be noted that character constants have numerical values known as ASCII values, for example, the value of ‗A‘ is 65 which is its ASCII value.

Escape Characters/ Escape Sequences

C allows us to have certain non graphic characters in character constants. Non graphic characters are those characters that cannot be typed directly from keyboard, for example, tabs, carriage return, etc.

These non graphic characters can be represented by using escape sequences represented by a backslash() followed by one or more characters.

NOTE: An escape sequence consumes only one byte of space as it represents a single character.



Escape Sequence	**Description**
a	Audible alert(bell)
b	Backspace
f	Form feed
n	New line
r	Carriage return
t	Horizontal tab
v	Vertical tab
\	Backslash
“	Double quotation mark
‘	Single quotation mark
?	Question mark
	Null

STRING CONSTANTS

String constants are sequence of characters enclosed within double quotes. For example,

—hello‖

—abc‖

—hello911‖

Every sting constant is automatically terminated with a special character **„‟** called the**nullcharacter** which represents the end of the string.

For example, —hello‖ will represent —hello‖ in the memory.

Thus, the size of the string is the total number of characters plus one for the null character.

Special Symbols

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.

[](){},;:*…=#

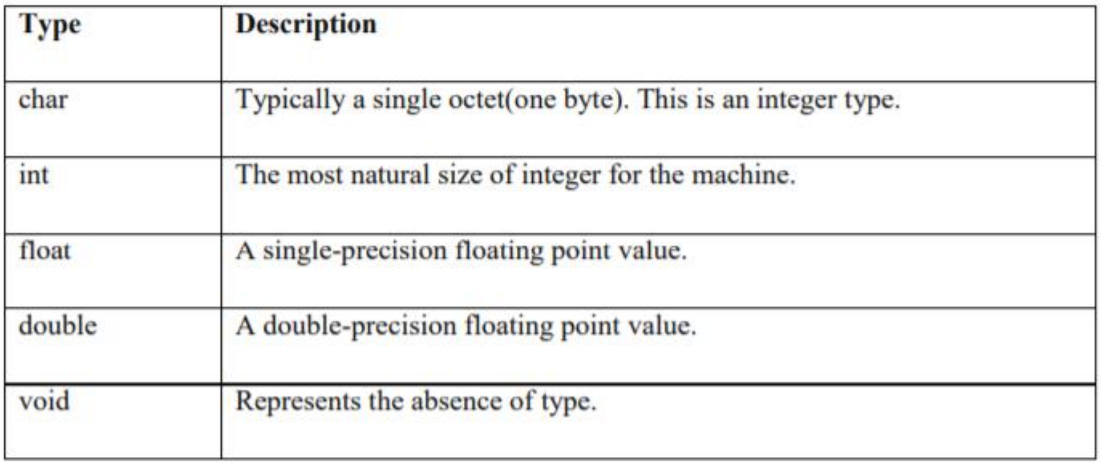
Braces{}: These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.
Parentheses(): These special symbols are used to indicate function calls and function parameters.
Brackets[]: Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.

Unit-1	**Lecture-5**	**Topic: Variable and Data types**

VARIABLES

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.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types explained in the previous chapter, there will be the following basic variable types –



Type	Description
char	Typically a single octet(one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.

C programming language also allows defining various other types of variables like Enumeration, Pointer, Array, Structure, Union, etc.

Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows −

type variable_list;

Here, **type** must be a valid C data type including char, int, float, double, or any user-defined object; and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

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

The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.
Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −
type variable_name = value;
Some examples are −
int d = 3, f = 5;   // declaration of d and f.

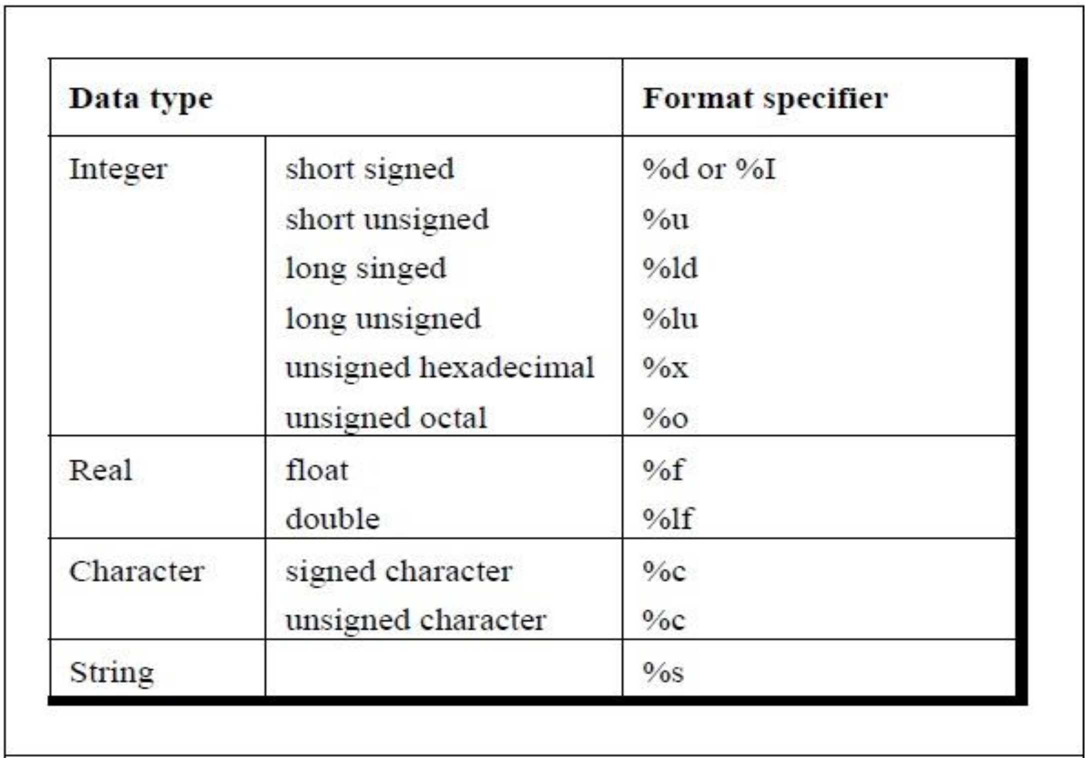
```
int d = 3, f = 5;          // definition and initializing d and f.
```

```
float z = 3.16;          // definition and initializes z.
char x = 'x';            // the variable x has the value 'x'.
```

Fundamental/Primary DataTypes:

C has the following simple data types (16-bit implementation):

Type	Size	Range	Precision for real numbers
char	1 byte	-128 to 127	
unsigned char	1 byte	0 to 255	
signed char	1 byte	-128 to 127	
short int or short	2 bytes	-32,768 to 32,767	
unsigned short or unsigned short int	2 bytes	0 to 65535	
int	2 bytes	-32,768 to 32,767	
unsigned int	2 bytes	0 to 65535	
Long or long int	4 bytes	-2147483648 to 2147483647 (2.1 billion)	
unsigned long or unsigned long int	4 bytes	0 to 4294967295	
float	4 bytes	3.4 E−38 to 3.4 E+38	6 digits of precision
double	8 bytes	1.7 E-308 to 1.7 E+308	15 digits of precision
long double	10 bytes	+3.4 E-4932 to 1.1 E+4932	provides between 16 and 30 decimal places



Data type		Format specifier
Integer	short signed	%d or %I
	short unsigned	%u
	long singed	%ld
	long unsigned	%lu
	unsigned hexadecimal	%x
	unsigned octal	%o
Real	float	%f
	double	%lf
Character	signed character	%c
	unsigned character	%c
String		%s

DeclaringVariables:

Variables are declared in three basic places: inside functions, in the definition of function parameters and outside of the functions. If they are declared inside functions, they are called as *local variables*, if in the definition of functions then formal parameters and out side of all functions, then global variables.

Globalvariables:

Global variables are known throughout the program. The variables hold their values throughout the programs execution. Global variables are created by declaring them outside of any function.

Global variables are defined above main () in the following way:

```
short number, sum;
int bignumber, bigsum;
char letter;
main()
{

}
```

It is also possible to pre-initialize global variables using the = operator for assignment.

Example:

```
float sum =0.0;
int bigsum = 0;
char letter =`A';
main()
{
}
```

This is the same as:

```
        float sum;
        int bigsum;
        char letter;
        main()
        {sum =0.0;
            bigsum = 0;
            letter=`A';
}
```

is more efficient.

C also allows multiple assignment statements using =, for example: a =

```
b = c = d = 3;
```

which is the same as, but more efficient than:

```
a =3;
```

```
b =3;
c =3;
d =3;
```

This kind of assignment is only possible if all the variable types in the statement are the same.

Localvariables:

Variable that are declared inside a function are called "local variables". These variables are referred to as "automatic variables". Local variables may be referenced only by statements that are inside the block in which the variables are declared.
Local variables exist only while the block of code in which they are declared is executing, i.e. a local variable is created upon entry into its block & destroyed upon exit.

Example:

Consider the following two functions:

```
void func1 (void)
{
        int x;
        x = 10;
}

void func2 (void)
{
        int x;
        x = -199;
}
```

The integer variable x is declared twice, once in func1 () & once in func2 (). The x in func1 () has no bearing on or relationship to the x in func2 (). This is because each x is only known to the code with in the same block as variable declaration.

Unit-1	Lecture-6	Topic: Standard I/O

ConsoleI/O:

Console I/O refers to operations that occur at the keyboard and screen of your computer.

Reading and writingCharacters:

The simplest of the console I/O functions are getche (), which reads a character from the keyboard, and putchar (), which prints a character to the screen. The getche () function waits until a key is pressed and then returns its value. The key pressed is also echoed to the screen automatically. The putchar () function will write its character argument to the screen at the current cursor position. The prototypes for getche () and putchar () are shown here:

```
int getche (void);
int putchar (int c);
```

The header file for getche () and putchar () is in CONIO.H.

The following programs inputs characters from the keyboard and prints them in reverse case. That is, uppercase prints as lowercase, the lowercase prints as uppercase. The program halts when a period is typed. The header file CTYPE.H is required by the islower() library function, which returns true if its argument is lowercase and false if it is not.

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

main(void)
{
        char ch;
        printf (“enter chars, enter a period to stop\n”); do
        {
                ch = getche (); if
                ( islower (ch))
                        putchar (toupper (ch));
                else
                        putchar (tolower (ch));
        } while (ch!=‘.’);                      /* use a period to stop */
        return 0;
}
```

There are two important variations on getche().

- The first is getchar(), which is the original, UNIX-based character inputfunction.
  - The trouble with getchar() is that it buffers input until a carriage return is entered. The reason for this is that the original UNIX systems line-buffered terminal input, i.e., you had to hit a carriage return for anything you had just typed to actually be sent to thecomputer.
  - The getchar() function uses the STDIO.H headerfile.
- A second, more useful, variation on getche() is getch(), which operates precisely like getche () except that the character you type is not echoed to the screen. It uses the CONIO.Hheader.

Reading and writingStrings:

On the next step, the functions gets() and puts() enable us to read and write strings of characters at the console.

The gets() function reads a string of characters entered at the keyboard and places them at the address pointed to by its argument. We can type characters at the keyboard until we strike a carriage return. The carriage return does not become part of the string; instead a null terminator is placed at the end and gets() returns. Typing mistakes can be corrected prior to striking ENTER. The prototype for gets() is:

```
char* gets (char *str);
```

Where, str is a character array that receives the characters input by the user. Its prototype is found in STDIO.H. The following program reads a string into the array str and prints its length.

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

main(void)
{
        char str[80];
        gets (str);
        printf (“length is %d”, strlen(str));
        return 0;
}
```

The puts() function writes its string argument to the screen followed by a newline. Its prototype is.

int puts (char str);

It recognizes the same backslash codes as printf(), such as “\t” for tab. It cannot output numbers or do format conversions. Therefore, puts() takes up less space and runs faster than printf(). Hence, the puts() function is often used when it is important to have highly optimized code. The puts() function returns a non negative value if successful, EOF otherwise. The following statement writes “hello” on the screen.

puts (“hello”);

The puts() function uses the STDIO.H header file.

Basic console I/O functions:

Function	**Operation**
getchar()	Reads a character from the keyboard and waits for carriage return
getche()	Reads a character with echo and does not waits for carriage return
getch()	Reads a character from the keyboard with out echo and not waits for carriage return
Putchar()	Writes a character to the screen
gets()	Reads a string from the keyboard
puts()	Writes a string to the screen

Distinguishion between getchar() and gets() functions:

getchar()	gets()
Used to receive a single character.	Used to receive a single string, white spaces and blanks.
Does not require any argument.	It requires a single argument.

Reading Character Data in a CProgram

All data that is entered into a C program by using the scanf function enters the computer through a special storage area in memory called the **standard input buffer** (or **stdin**). A user's keystrokes (including the new line character \n generated when the Enter key is pressed) are stored in this buffer, which can then be read using functions such as scanf. When numbers are read from this area, the function converts the keystrokes stored in the buffer (except for the \n) into the type of data specified by the control string (such as "%f") and stores it in the memory location indicated by the second parameter in the scanf function call (the variable's address). The \n remains in the buffer. This can cause a problem if the next scanf statement is intended to read a *character* from the buffer. The program will mistakenly read the remaining \n as the character pressed by the user and then proceed to the next statement, never even allowing the user to enter a character of theirown.

You can solve this problem when writing C statements to read *character* data from the keyboard by adding a call to a special function named **fflush**that clears all characters (including \n's) from the given input buffer. The statement would be place ahead of each statement in your program used to input characters, such as:

```
fflush(stdin); scanf("%c", &A);
          or
fflush(stdin); A=getchar();
```

Formatted Console I/O (printf() andscanf()):

The pre-written function printf() is used to output most types of data to the screen and to other devices such as disks. The C statement to display the word "Hello" wouldbe:

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

The printf () function can be found in the stdio.h header file.

DisplayingPrompts:

The printf() function can be used to display "prompts" (messages urging a user to enter some data) as in:

```
printf ("How old are you? ");
```

When executed, this statement would leave the cursor on the same line as the prompt and allow the user to enter a response following it on the same line. A space was included at the end of the prompt (before the closing quote mark) to separate the upcoming response from the prompt and make the prompt easier to read during data entry.

CarriageReturns:

If you want a carriage return to occur after displaying a message, simply include the special *escape sequence* \n at the end of the of the message before the terminating quote mark, as in thestatement.

```
printf ("This is my program.\n");
```

ConversionSpecifiers:

All data output by the printf() function is actually produced as a string of characters (one symbol after another). This is because display screens are character-based devices. In order tosend any other type of data (such as integers and floats) to the screen, you must add special symbols called conversion specifiers to the output command to convert data from its stored data format into a string of characters. The C statement to display the floating point number 123.456 wouldbe:

```
printf ("%f",123.456);
```

The "%f" is a conversion specifier. It tells the printf() function to convert the floating point data into a string of characters so that it can be sent to the screen. When outputting non-string data with the printf() function you must include two parameters (items of data) within the parentheses following printf(). The first parameter is a quoted control string containing the appropriate conversion specifier for the type of non-string data being displayed *and optionally* any text that should be displayed with it. The second parameter (following a comma) should be the value or variable containing the value. The C statement to display the floating point number 123.456 preceded by the string "The answer is: " and followed by a carriage return would be:

```
printf ("The answer is: %f\n",123.456);
```

Notice that the value does not get typed inside of the quoted control string, but rather as a separate item following it and separated by a comma. The conversion specifier acts as a place holder for the data within the outputstring.

The following table lists the most common conversion specifiers and the types of data that they convert:

Specifier	**Data Type**
%c	char
%f	float
%d or %i	signed int (decimal)
%h	short int
%p	Pointer (Address)
%s	String of Characters
Qualified Data Types	
%lf	long float or double
%o	unsigned int (octal)
%u	unsigned int (decimal)
%x	unsigned int (hexadecimal)
%X	Unsigned Hexadecimal (Upper Case)
%e	Scientific Notation (Lower case e)
%E	Scientific Notation (Upper case E)
%g	Uses %e or %fwhich ever is shorter
%ho	short unsigned int (octal)
%hu	short unsigned int (decimal)
%hx	short unsigned int (hexadecimal)
%lo	long unsigned int (octal)
%lu	long unsigned int (decimal)
%lx	long unsigned int (hexadecimal)
%Lf	long double
%n	The associated argument is an integer pointer into which the number of characters written so far is placed.
%%	Prints a % sign

Output Field Width andRounding:

When displaying the contents of a variable, we seldom know what the value will be. And yet, we can still control the format of the output field (area), including the:

amount of space provided for output (referred to as the output's "*field width*")
alignment of output (left or right) within a specified field and rounding *of floating point numbers* to a fixed number of places right of the decimal point

Output formatting is used to define specific alignment and rounding of output, and can be performed in the printf() function by including information within the conversion specifier. For example, to display the floating point number 123.456 right-aligned in an eight character wide output field and rounded to two decimal places, you would expand basic conversion specifier from "%f" to "%8.2f". The revised statement wouldread:

```
printf ("The answer is:%8.2f\n",123.456);
```

The 8 would indicate the width, and the .2 would indicating the rounding. *Keep in mind that the addition of these specifiers as no effect on the value itself, only on the appearance of the output characters.*

The value 123.456 in the statement above could be replaced by a variable or a symbolic constant of the same data type. The 8 in the statement above specifies the width of the output field. If you include a width in the conversion specifier, the function will attempt to display the number in that width, aligning it against the rightmost character position. Unused positions will appear as blank spaces (padding) on the left of the output field. If you want the value to be left-aligned within the field, precede the width with a minus sign (-). If no width is specified, the number will be displayed using only as many characters as are necessary without padding. When a values is too large to fit in a specified field width, the function will expand the field to use as many characters as necessary.

The .2 in the statement above specifies the decimal precision (i.e., the number of place that you want to appear to the right of the decimal point) *and would be used only in situations where you are outputting floating point values*. The function will round the output to use the specified number of places (adding zeros to the end if necessary). Any specified field width *includes* the decimal point and the digits to its right. The default decimal precision (if none is specified) is 6 places.

In situations where you want floating point values displayed in scientific notation, formatting also is used to define specific alignment and rounding of output in the printf function by including information within the conversion specifier %e. For example, to display the floating point number 123.456 in scientific notation in a twelve character wide output field with its mantissa (significant digits) rounded to two decimal places, you would expand basic conversion specifier from "%e" to "%12.2e". The resulting output wouldbe:

```
The answer is: 1.23e+002
```

Notice that values displayed in scientific notation always place the decimal point after the first significant digit and use the exponent (digits shown following the letter e) to express the power of the number. The C statement to produce the output above would be:

```
printf ("The answer is:%12.2e\n",123.456);
```

The 12 would indicate the *overall* field width (following any message) *including* the decimal point and the exponential information. The .2 would indicating the rounding of the digits following the decimal point.

C command using printf () with various conversion specifiers:	Output Produced on Screen																			
	Position:									1	1	1	1	1	1	1	1	1	1	2
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
printf ("%3c",'A');			**A**	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%-3c",'A');	**A**			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%8s","ABCD");					**A**	**B**	**C**	**D**	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%d",52);	**5**	**2**	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%8d",52);							**5**	**2**	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%-8d",52);	**5**	**2**							X	X	X	X	X	X	X	X	X	X	X	X
printf ("%f",123.456);	**1**	**2**	**3**	**.**	**4**	**5**	**6**	**0**	**0**	**0**	X	X	X	X	X	X	X	X	X	X
printf ("%10f",123.456);	**1**	**2**	**3**	**.**	**4**	**5**	**6**	**0**	**0**	**0**	X	X	X	X	X	X	X	X	X	X
printf ("%10.2f",123.456);					**1**	**2**	**3**	**.**	**4**	**6**	X	X	X	X	X	X	X	X	X	X
printf ("%-10.2f",123.456);	**1**	**2**	**3**	**.**	**4**	**6**					X	X	X	X	X	X	X	X	X	X
printf ("%.2f",123.456);	**1**	**2**	**3**	**.**	**4**	**6**	X	X	X	X	X	X	X	X	X	X	X	X	X	X
printf ("%10.3f",-45.8);				**-**	**4**	**5**	**.**	**8**	**0**	**0**	X	X	X	X	X	X	X	X	X	X
printf ("%10f",0.00085);			**0**	**.**	**0**	**0**	**0**	**8**	**5**	**0**	X	X	X	X	X	X	X	X	X	X
printf ("%10.2e",123.89);		**1**	**.**	**2**	**4**	**e**	**+**	**0**	**0**	**2**	X	X	X	X	X	X	X	X	X	X

Distinguishing between printf() and puts() functions:

puts()	printf()
They can display only one string at a time.	They can display any number of characters, integers or strings a time.
All data types of considered as characters.	Each data type is considered separately depending upon the conversion specifications.

scanf () function:

scanf() is the general purpose console input routine. It can read all the built-in data types of automatically convert numbers into the proper internal format. It is much like the reverse of printf(). The f in scanf stands forformatted.

```
# include <stdio.h>
int main()
{
        int num;
        printf ("Enter a number: ");
        scanf("%d", &num);
        printf("The number you have entered was %d\n", num);
        return 0;
}
```

The above program requires a variable in which to store the value for num. The declaration, int num; provides a temporary storage area called num that has a data type of integer (whole number). The scanf function requires a Format String, which is provided by the %d character pair. The percent sign is an introducer and is followed by a conversion character, d, which specifies that the input is to be an integer. The input is stored in the variable, num. The scanf function requires the address of the variable, and this is achieved by prefixing the variable name with an ampersand (eg.&num).

The printf statement also uses the format string to relay the information. The printf function does not require the address of the variable and so the ampersand is not required. The prototype for scanf() is in STDIO.H.

The format specifiers for scanf () are as follows:

Code	Meaning
%c	Read a single character
%d	Read a decimal integer
%i	Read a decimal integer, hexa decimal or octal integer
%h	Read a short integer
%e	Read a floating-point number
%f	Read a floating-point number
%g	Read a floating-point number
%o	Read an octal number
%s	Read a string
%x	Read a hexadecimal number
%p	Read a pointer
%n	Receives an integer value equal to the number of characters read so far
%u	Read an unsigned integer
%[..]	Scan for a string of words

Distinguishing between scanf() and gets() functions:

scanf()	gets()
Strings with spaces cannot be accessed until ENTER key is pressed.	Strings with any number of spaces can be accessed.
All data types can be accessed.	Only character data type can be accessed.
Spaces and tabs are not acceptable as a part of the input string.	Spaces and tabs are perfectly acceptable of the input string as a part.
Any number of characters, integers.	Only one string can be received at a time. Strings, floats can be received at a time.

OPERATORS AND EXPRESSIONS

C language offers many types of operators. They are,

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bit wise operators
6. Conditional operators (ternary operators)
7. Increment/decrement operators
8. Special operators

Operators:

There are three general classes of operators: arithmetic, relational and logical and bitwise.

ArithmeticOperators:

Operation	Symbol	Meaning
Add	+	
Subtract	-	
Multiply	*	
Division	/	Remainder lost
Modulus	%	Gives the remainder on integer division, so 7 % 3 is 1.
--	Decrement	
++	Increment	

As well as the standard arithmetic operators (+ - * /) found in most languages, C provides some more operators.

Assignment is = ***i.e.****i*= 4; ch = `y';

Increment ++, Decrement -- which are more efficient than their long hand equivalents.

For example, X = X + 1 can be written as ++X or as X++. There is however a difference when they are used in expression.

The ++ and -- operators can be either in post-fixed or pre-fixed. A pre-increment operation such as ++a, increments the value of a by 1, before a is used for computation, while a post increment operation such as a++, uses the current value of a in the calculation and then increments the value of a by 1. Consider the following:

```
X = 10;
Y = ++X;
```

In this case, Y will be set to 11 because X is first incremented and then assigned to Y. However if the code had been written as

```
X = 10;
Y = X++;
```

Y would have been set to 10 and then X incremented. In both the cases, X is set to 11; the difference is when it happens.

The % (modulus) operator only works with integers.

Division / is for both integer and float division. So be careful.

The answer to:*x* = 3 / 2 is 1 even if *x* is declared a float!!

RULE: If both arguments of / are integer then do integer division. So make sure you do this. The correct (for division) answer to the above is *x* = 3.0 / 2 or *x*= 3 / 2.0 or (better) *x* = 3.0 /2.0.

There is also a convenient **shorthand** way to express computations in C. It is

very common to have expressions like: *i*= *i*+ 3 or *x* = *x* * (*y* + 2) This can

written in C (generally) in a ***shorthand*** form like this:

We can rewrite *i*= *i*+ 3 as *i*+= 3 and

x = *x* * (*y* + 2) as *x* *= *y* + 2.

NOTE: that *x* *= *y* + 2 means *x* = *x* * (*y* + 2) and <u>NOT</u>*x* = *x* * *y* + 2.

RelationalOperators:

The relational operators are used to determine the relationship of one quantity to another. They always return 1 or 0 depending upon the outcome of the test. The relational operators are as follows:

Operator	Action
= =	equal to
! =	not equal to
<	Less than
<=	less than or equal to
>	Greater than
>=	Greater than or equal to

To test for equality is ==

If the values of x and y, are 1 and 2 respectively then the various expressions and their results are:

Expression	**Result**	**Value**
X != 2	False	0
X == 2	False	0
X == 1	True	1
Y != 3	True	1

A warning: Beware of using "=" instead of "= =", such as writing accidentally

```
if (i = j) .....
```

This is a perfectly **LEGAL** C statement (syntactically speaking) which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if j is non- zero. This is called **assignment by value** -- a key feature of C.

Not equals is: !=

Other operators < (less than), > (greater than), <= (less than or equals), >= (greater than or equals) are as usual.

Logical (Comparison)Operators:

Logical operators are usually used with conditional statements. The three basic logical operators are && for logical AND, || for logical OR and ! for not.

The truth table for the logical operators is shown here using one's and zero's. (the idea of true and false under lies the concepts of relational and logical operators). In C true is any value other than zero, false is zero. Expressions that use relational or logical operators return zero for false and one fortrue.

P	Q	P && q	P || q	! p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Example:

(i) x == 6 && y ==7

This while expression will be TRUE (1) if both x equals 6 and y equals 7, and FALSE (0) otherwise.

(ii) x < 5 || x >8

This whole expression will be TRUE (1) if either x is less than 5 or x is greater than 8 and FALSE (0) otherwise.

Bit wiseOperators:

The ***bit wise*** operators of C are summarized in the following table:

Bitwise operators	
&	AND
|	OR
^	XOR
~	One's Compliment
<<	Left shift
>>	Right Shift

The truth table for Bitwise operators AND, OR, and XOR is shown below. The table uses 1 for true and 0 forfalse.

P	Q	P AND q	P OR q	P XOR q
0	0	0	0	0
0	1	0	1	1

1	0	0	1	1
1	1	1	1	0

DO NOT confuse & with &&: & is bit wise AND, &&logical AND. Similarly for | and ||.

if X = 00000010 (binary) or 2 (decimal)

~ is a unary operator: it only operates on one argument to right of the operator. It finds 1's compliment (unary). It translates all the 1 bits into O's and all O's into 1's

Example:

12 = 00001100
~12 = 11110011 = 243

The shift operators perform appropriate shift by operator on the right to the operator on the left. The right operator must be positive. The vacated bits are filled with zero (***i.e.***when shift operation takes places any bits shifted off are lost).
Example:

X << 2 shifts the bits in X by 2 places to the left.
X >>= 2 implies X = 00000000 or 0 (decimal)

Also: if X = 00000010 (binary) or 2 (decimal) X

<<= 2 implies X = 00001000 or 8(decimal)

Therefore, a shift left is equivalent to a multiplication by 2.

Similarly, a shift right is equal to division by2.

NOTE: Shifting is much faster than actual multiplication (*) or division (/) by 2. So if you want fast multiplications or division by 2 ***use shifts***.

The bit wise AND operator (&) returns 1 if both the operands are one, otherwise it returns zero. For example, if y = 29 and z = 83, x = y & z the resultis:

```
0  0  0  1  1  1  0  1    29 inbinary
                                    &
0  1  0  1  0  0  1  1             83
                             inbinary
0  0  0  1  0  0  0  1

                          Result
```

The bit wise or operator (|) returns 1 if one or more bits have a value of 1, otherwise it returns zero. For example if, y = 29 and z = 83, x = y | z the result is:

```
0  0  0  1  1  1  0  1    29 inbinary
                                    |
0  1  0  1  0  0  1  1    83 inbinary

0  1  0  1  1  1  1  1
```

Result

The bit wise XOR operator (^) returns 1 if one of the operand is 1 and the other is zero, otherwise if returns zero. For example, if y = 29 and z = 83, x = y ^ z the result is:

```
0  0  0  1  1  1  0  1    29 inbinary
                                    ^
0  1  0  1  0  0  1  1    83 inbinary

0  1  0  0  1  1  1  0   Result
```

Unit-1	Lecture-8	Topic: operators and expressions cont..

ConditionalOperator:

Conditional expression use the operator symbols question mark

(?) (x > 7) ?2 : 3

What this says is that if x is greater than 7 then the expression value is 2. Otherwise the expression value is 3.

In general, the format of a conditional expression
is: a ? b : c

Where, a, b & c can be any C expressions.

Evaluation of this expression begins with the evaluation of the sub-expression 'a'. If the value of 'a' is true then the while condition expression evaluates to the value of the sub-expression 'b'. If the value of 'a' is FALSE then the conditional expression returns the value of the sub-expression 'C'.

sizeofOperator:

In situation where you need to incorporate the size of some object into an expression and also for the code to be portable across different machines the size of unary operator will be useful. The size of operator computes the size of any object at compile time. This can be used for dynamic memoryallocation.

Usage: sizeof(object)
The object itself can be the name of any sort of variable or the name of a basic type (like int, float, char etc).

Example:

sizeof (char) =
1 sizeof (int) =
2 sizeof (float)
= 4 sizeof
(double) = 8

SpecialOperators:

Some of the special operators used in C are listed below. These are reffered as separators or punctuators.

Ampersand (&)	Comma (,)	Asterick(*)
Ellipsis (…)	Braces ({ })	Hash (#)
Brackets ([ ])	Parenthesis (())	Colon (:)
Semicolon (;)		

Ampersand:
Ampersand (&) also referred as address of operator usually precedes the identifier name, which indicates the memory allocation (address) of the identifier.

Comma:

Comma (,) operator is used to link the related expressions together. Comma used expressions are linked from left to right and the value of the right most expression is the value of the combined expression. The comma operator has the lowest precedence of all operators. For example:

Sum = (x = 12, y = 8, x +

y); The result will be sum = 20.

The comma operator is also used to separate variables during declaration. For example:

int a, b, c;

Asterick:

Asterick(*) also referred as an indirection operator usually precedes the identifier name, which identifies the creation of the pointer operator. It is also used as an unary operator.

Hash:

Hash (#) also referred as pound sign is used to indicate preprocessor directives, which is discussed in detail already.

Parenthesis:

Parenthesis () also referred as function call operator is used to indicate the opening and closing of function prototypes, function calls, function parameters, etc., Parenthesis are also used to group expressions, and there by changing the order of evaluation of expressions.

Semicolon:

Semicolon (;) is a statement terminator. It is used to end a C statement. All valid C statements must end with a semicolon, which the C compiler interprets as the end of the statement. For example:

```
c = a
+ b; b
= 5;
```

Order of Precedence of COperators:

It is necessary to be careful of the meaning of such expressions as a + b * c.

We may want the effect as

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

All operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right, so that:

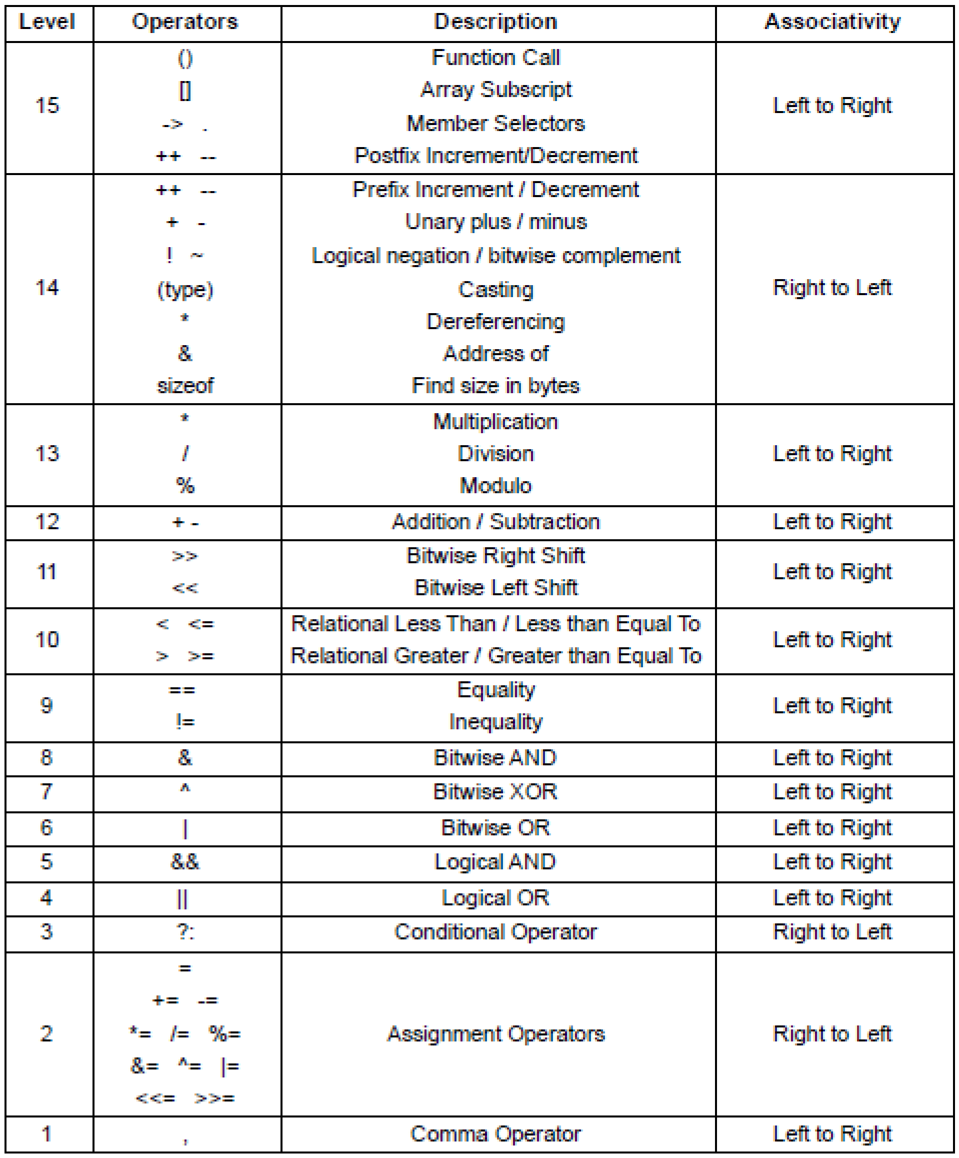
```
a - b - c
```

is evaluated as: (a - b)

- c as you would

expect.

From high priority to low priority the order for all C operators (we have not met all of them yet) is:



Level	Operators	Description	Associativity
15	()	Function Call	Left to Right
	[]	Array Subscript	
	-> .	Member Selectors	
	++ --	Postfix Increment/Decrement	
14	++ --	Prefix Increment / Decrement	Right to Left
	+ -	Unary plus / minus	
	! ~	Logical negation / bitwise complement	
	(type)	Casting	
	*	Dereferencing	
	&	Address of	
	sizeof	Find size in bytes	
13	*	Multiplication	Left to Right
	/	Division	
	%	Modulo	
12	+ -	Addition / Subtraction	Left to Right
11	>>	Bitwise Right Shift	Left to Right
	<<	Bitwise Left Shift	
10	< <=	Relational Less Than / Less than Equal To	Left to Right
	> >=	Relational Greater / Greater than Equal To	
9	==	Equality	Left to Right
	!=	Inequality	
8	&	Bitwise AND	Left to Right
7	^	Bitwise XOR	Left to Right
6	|	Bitwise OR	Left to Right
5	&&	Logical AND	Left to Right
4	||	Logical OR	Left to Right
3	?:	Conditional Operator	Right to Left
2	=	Assignment Operators	Right to Left
	+= -=		
	*= /= %=		
	&= ^= |=		
	<<= >>=		
1	,	Comma Operator	Left to Right

Thus: a < 10 && 2 * b < c is interpreted as: (a < 10) && ((2 * b) < c)

PairMatching:

Turbo C++ editor will find the companion delimiters for the following pairs:

{ }, [ ], ( ), <>, /* */, " " and ' '.

To find the matching delimiter, place the cursor on the delimiter you wish to match and press CTRL – Q [for a forward match and CTRL – Q] for a reverse match. The editor will move the cursor to the matching delimiter.

Casting betweentypes:

Mixing *types* can cause problems. For

```
example: int a = 3;
int b
= 2;
float
c;
c = b * (a / b);
printf ("2 * (3 / 2) = %f\n", c);
```

doesn't behave as you might expect. Because the first (a/b) is performed with integer arithmetic it gets the answer 1 not 1.5. Therefore, the program prints:

```
2 * (3/2) = 2.000
```

The best way round this is what is known as a *cast*. We can *cast* a variable of one type to another type like so:

```
int a = 3;
int b = 2;
float c;
c = b * ( (float) a / b);
```

The (float) a construct tells the compiler to switch the type of variable to be a float. The value of the expression is automatically cast to float. The main use of *casting* is when you have written a routine which takes a variable of one type and you want to call it with a variable of another type. For example, say we have written a power function with a prototype like so:

```
int pow (int n, int m);
```

We might well have a float that we want to find an approximate power of a given number ***n***. Your compiler should complain bitterly about your writing:

```
float n= 4.0;
int squared;
squared= pow (n, 2);
```

The compiler will not like this because it expects ***n*** to be of type int but not float.

However, in this case, we want to tell the compiler that we do know what we're doing and have good reason for passing it a float when it expects an int (whatever that reason might be). Again, a cast can rescueus:

```
float n = 4.0;
int squared;
squared = pow ((int)n,2);          /* We cast the float down to anint*/
```

IMPORTANT RULE: To move a variable from one type to another then we use a *cast* which has the form (*variable_type*) *variable_name*.

CAUTION: It can be a problem when we *downcast* – that is cast to a type which has less precision than the type we are casting from. For example, if we cast a double to a float we will lose some bits of precision. If we cast an int to a char it is likely to overflow [recall that a char is basically an int which fits into 8 binarybits].