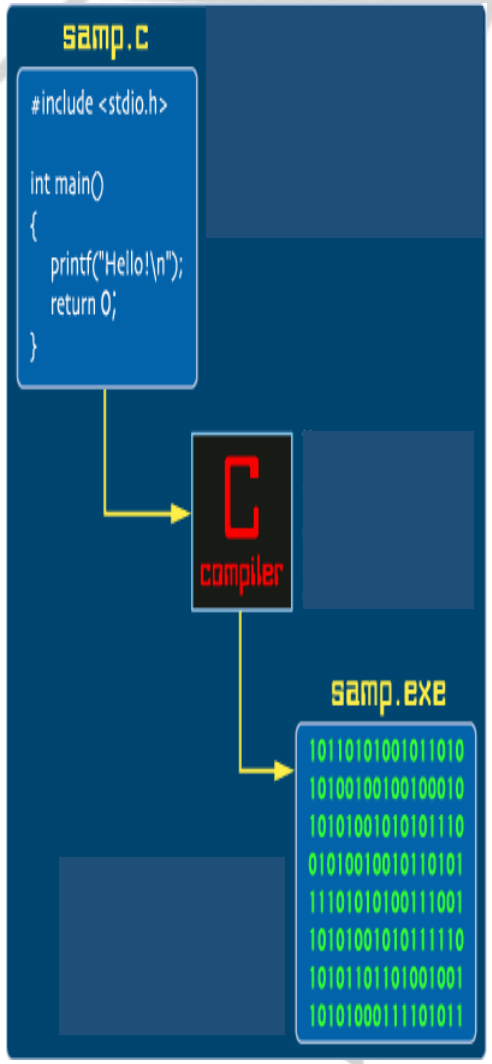
UNIVERSITY of SAN CARLOS
SCIENTIA • VIRTUS • DEVOTIO

DCISM
DEPARTMENT OF COMPUTER, INFORMATION SCIENCES AND MATHEMATICS

CIS 1101 – PROGRAMMING 1

INTRODUCTION TO C

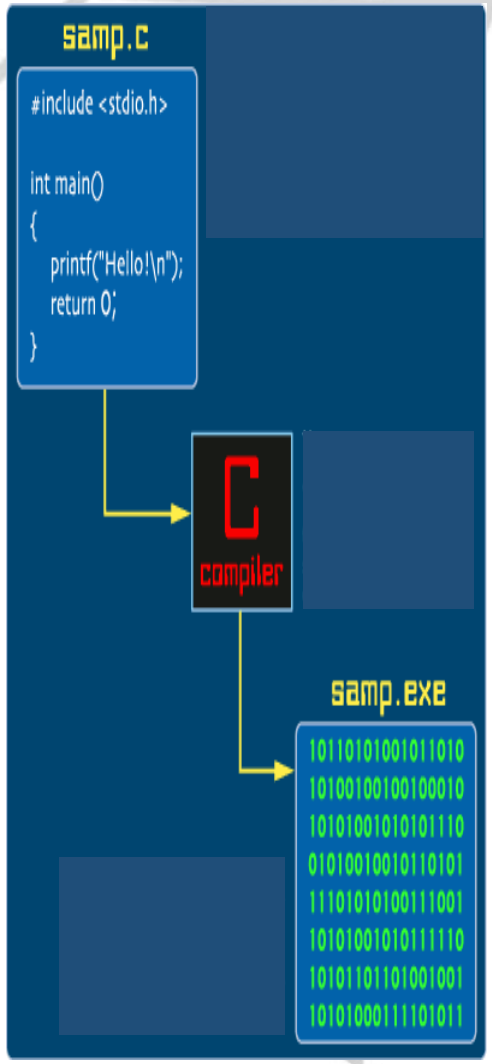
WHAT IS C?



- **C** Programming Language:

  A high-level structured oriented programming language, used in general purpose programming, developed by Dennis Ritchie at AT&T Bell Labs, in the USA in 1972.

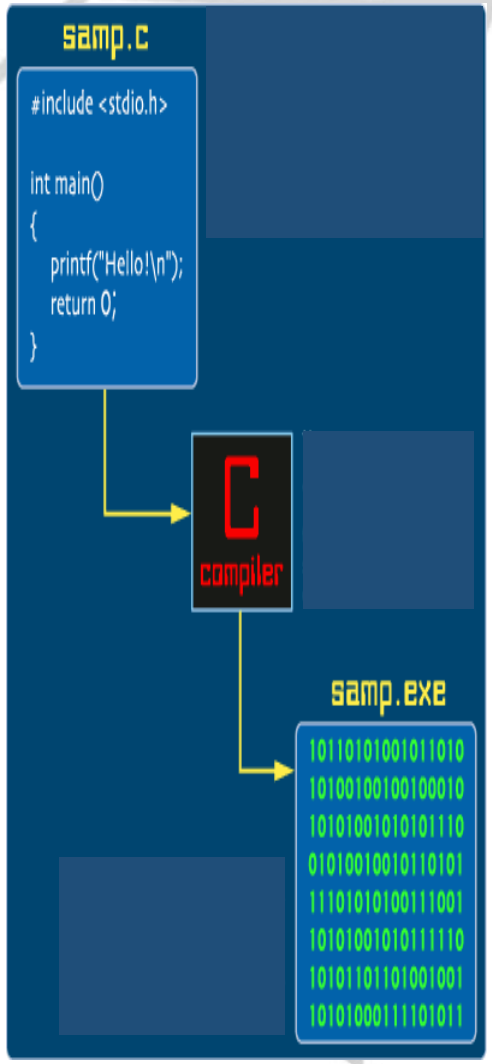
WHAT IS C?



- **C can be defined by the following ways:**

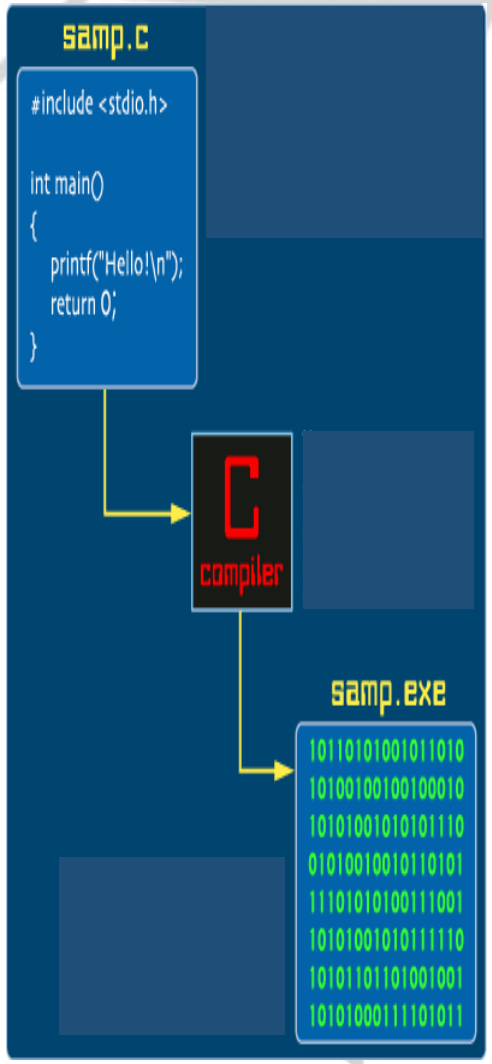
1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

C: MOTHER LANGUAGE



- **Most** of the **compilers, JVMs, Kernels,** and others are **written** in **C** language
- **Most** of the programming languages follow **C** syntax, such as: C++, Java, C#, and others
- **Provides** the core concepts like the array, strings, functions, file handling, and others that are being used in many languages like C++, Java, C#, and others

C: SYSTEM PROGRAMMING LANGUAGE

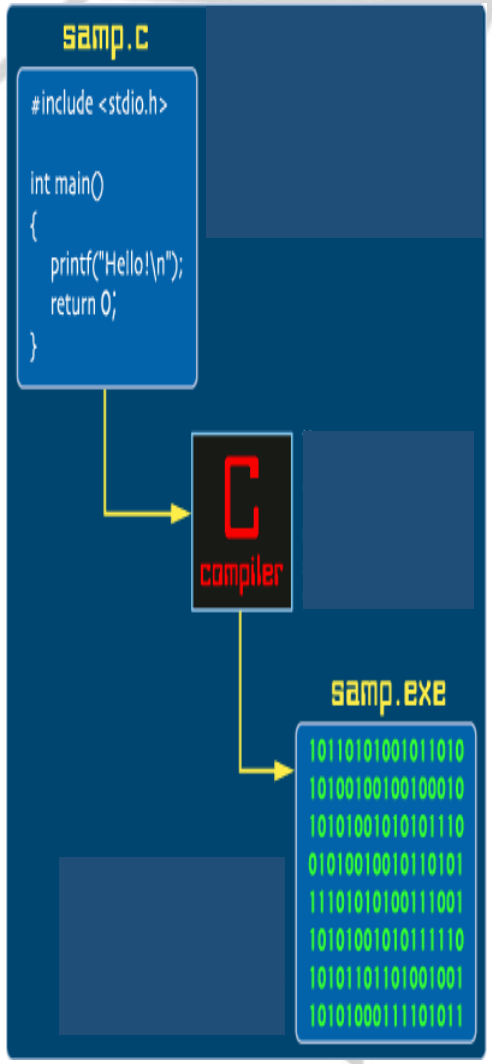


- **System Programming Language:**
  - **used** to **create system software**

- **C** language is a system programming language because it can be **used** to **do low-level programming** such as **drivers** and **kernels.**

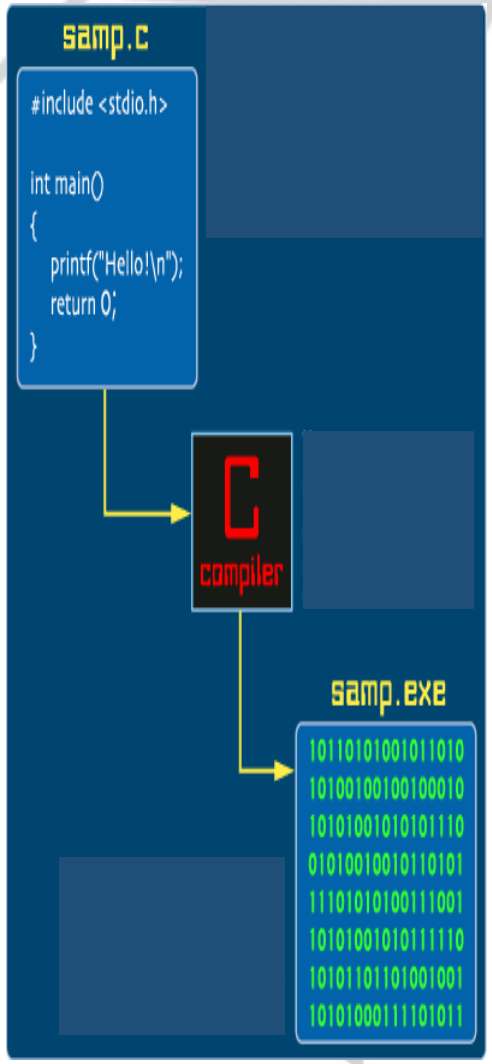
- C is generally used to create hardware devices, OS, drivers, kernels, and others.
  - Example: Linux kernel is written in C.

C: PROCEDURE-ORIENTED PROGRAMMING LANGUAGE



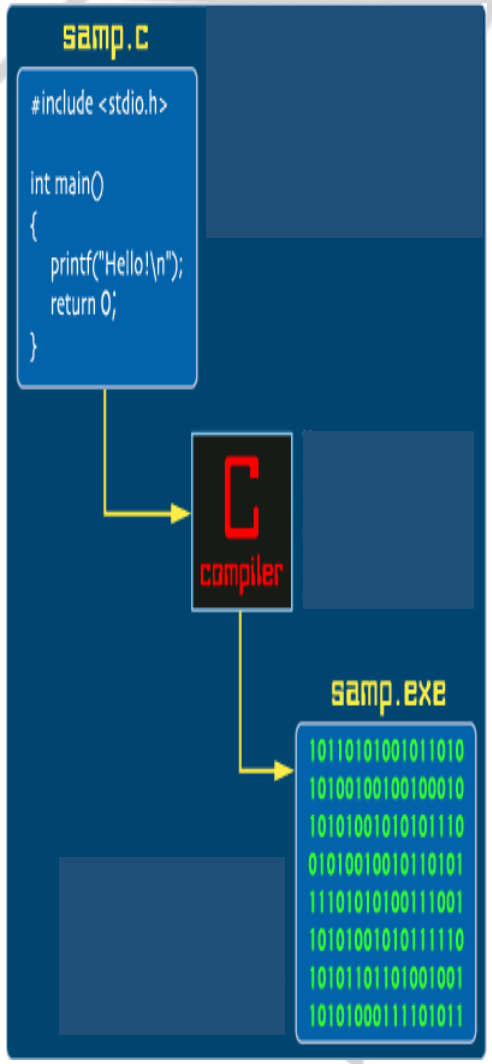
- Procedure:
  - known as a function, method, routine, subroutine, and others.
- Procedural language:
  - specifies a **series of steps** for the **program** to **solve the problem**.
  - breaks the program into functions, data structures, and others such as in **C**
- **Note:**
  - In C, **variables and function prototypes** must be **declared before being used.**

C: STRUCTURED PROGRAMMING LANGUAGE



- **Structured Programming Language**:
  - a **subset** of the procedural language.
- **Structure:**
  - means to **break a program into parts** or **blocks** so that it may be easy to understand.
- In **C** language, programs are **broken into parts** using functions to make the program easier to understand and modify.

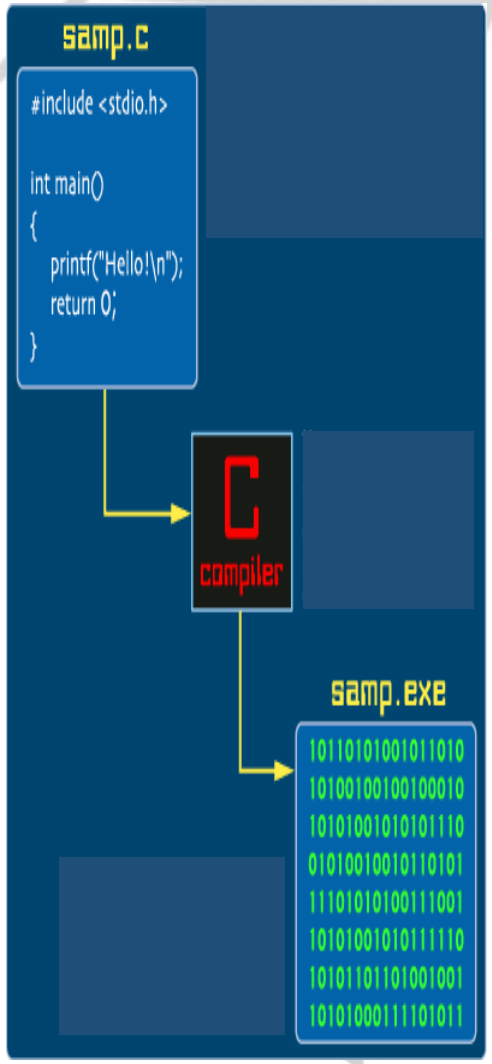
C: MID-LEVEL PROGRAMMING LANGUAGE



- **C** is considered as a **middle-level language** because it **supports** the feature of **both low-level** and **high-level** languages.

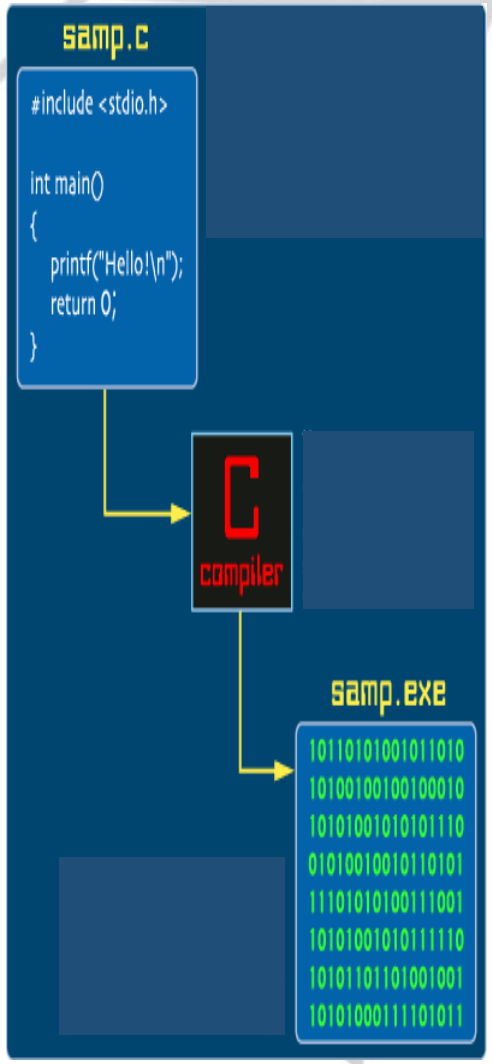
- **C** language program is **converted** into assembly code and it supports pointer arithmetic (low-level) but it is machine independent (a feature of high-level).

WHY LEARN C PROGRAMMING?



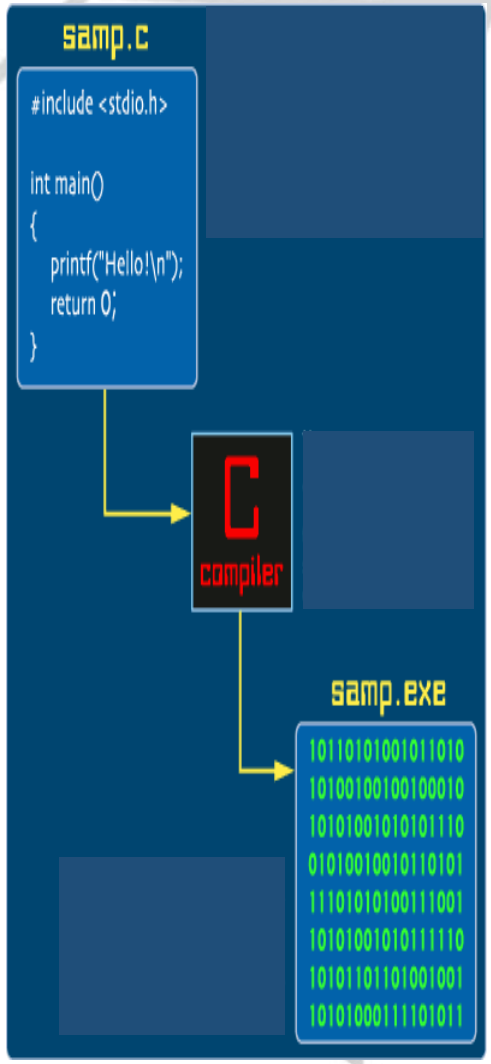
- **Key Advantages** of **Learning C Programming:**
  - Easy to learn
  - Structured language
  - Produces efficient programs
  - Can handle low-level activities
  - Can be compiled on a variety of computer platforms

APPLICATIONS OF C



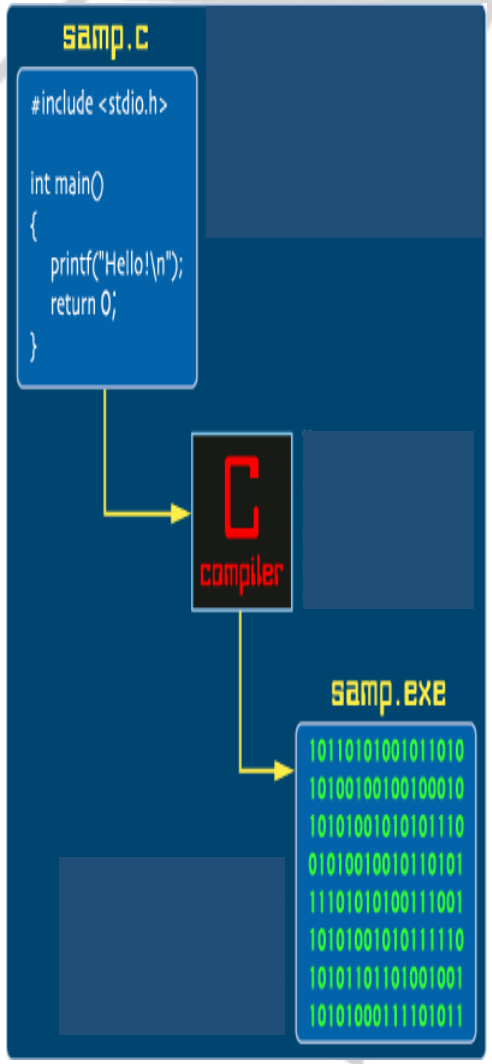
- Operating Systems
- Language Compilers
- Language Interpreters
- Assemblers
- Text Editors
- Network Drivers
- Modern Programs
- Browsers and Extensions
- Database Systems
- Print Spoolers
- Utilities
- Graphics packages
- Word processors
- Spreadsheets
- IoT Applications
- Embedded Systems

FACTS ABOUT C



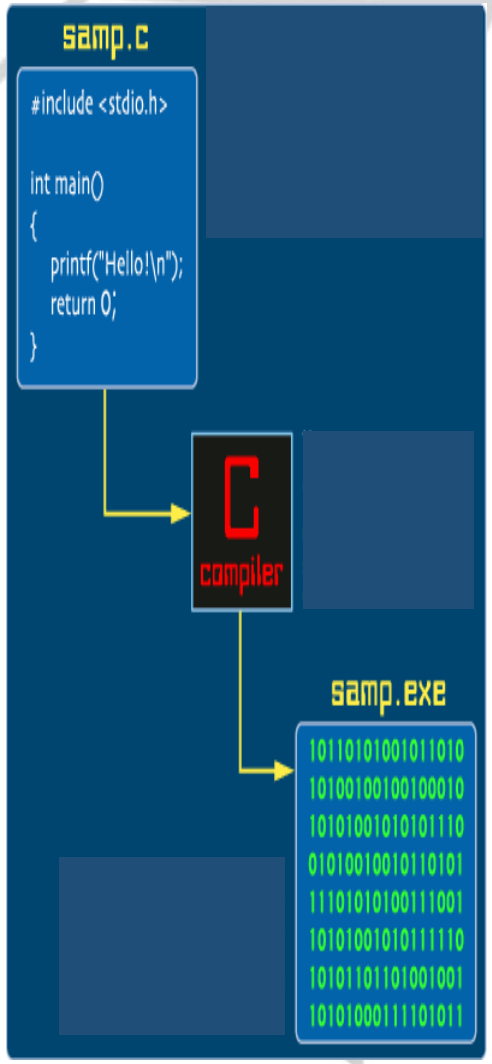
- C was **invented to write** an operating system called **UNIX**.
- C is a **successor of B language** and was introduced and created by Dennis Ritchie in 1972.
- The language was **formalized in 1988** by the **A**merican **N**ational **S**tandard **I**nstitute (**ANSI**).
- The **UNIX OS** was totally written in C.

FACTS ABOUT C



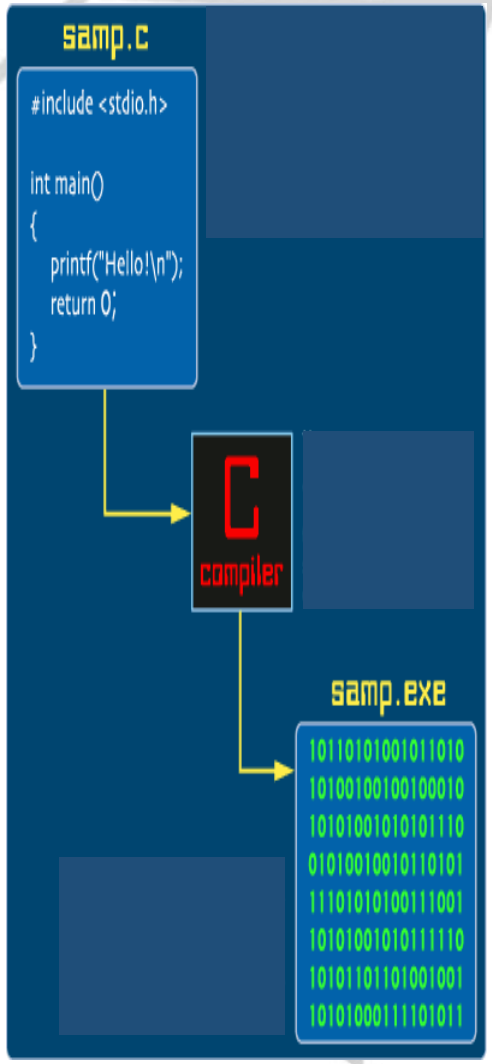
- Today, C is the most widely used and popular System Programming Language.
- **Most** of the state-of-the-art software have been implemented using C.
- Today's most popular Linux OS and RDBMS MySQL have been written in C.

C PROGRAM LANGUAGE: OTHER INFO



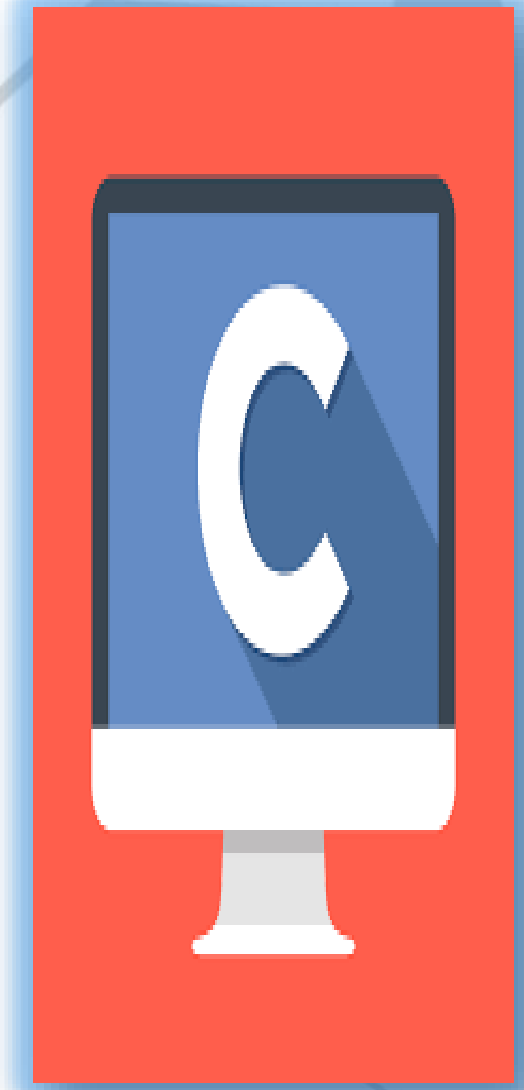
- C is a compiled language and it must run through a C compiler to turn it into an executable file that the computer can run.
- One must have access to a C compiler in order to be able to write and run a C program.
- If one is working at home on a Windows machine, there is a need to download a free C compiler or purchase a commercial compiler.

C PROGRAM LANGUAGE: OTHER INFO



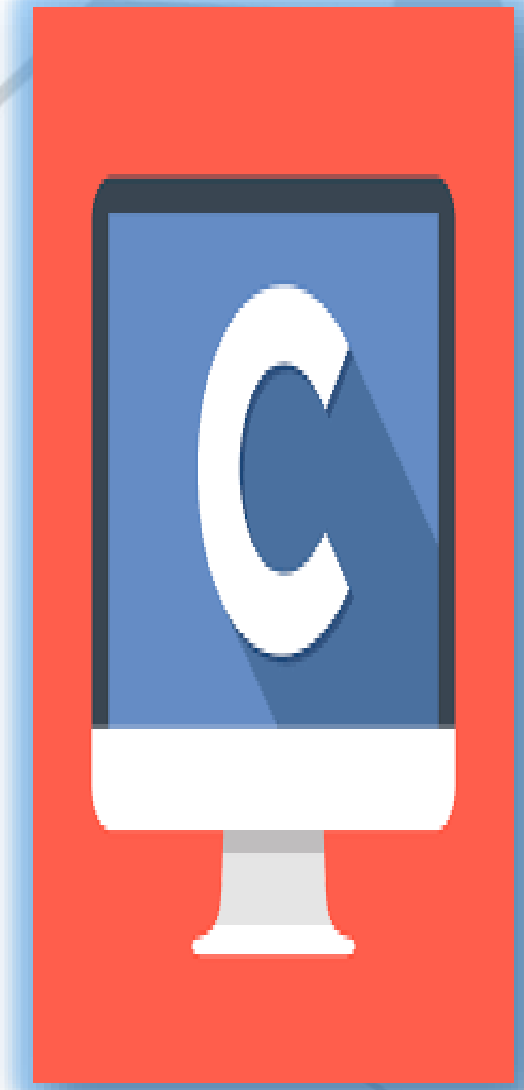
- Many later languages have borrowed syntax/features directly or indirectly from C language.
- Syntax of Java, PHP, JavaScript and many other languages is mainly based on C language.
- C++ is nearly a superset of C language (there are few programs that may compile in C, but not in C++).

COMPONENTS OF C LANGUAGE: VARIABLE



- **Variable:**
  - A named **memory location** which acts as **placeholder** or **container** for **storing data** whose **value can change during program execution**.

TYPES OF VARIABLES IN C



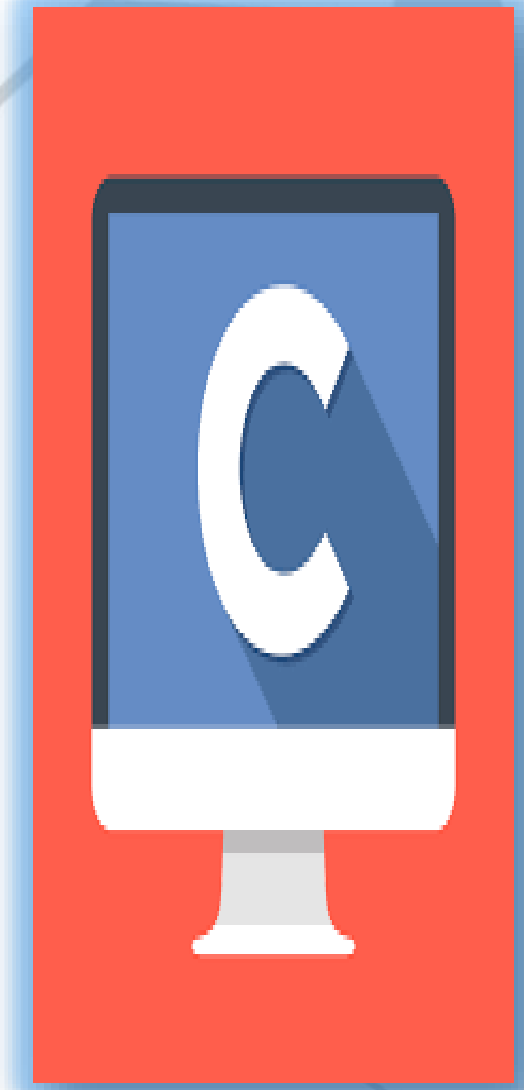
1. Local Variable

- A variable that is declared inside the function or block, must be declared at the start of the block, and must be initialized before it is used.

- **Example:**

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

TYPES OF VARIABLES IN C



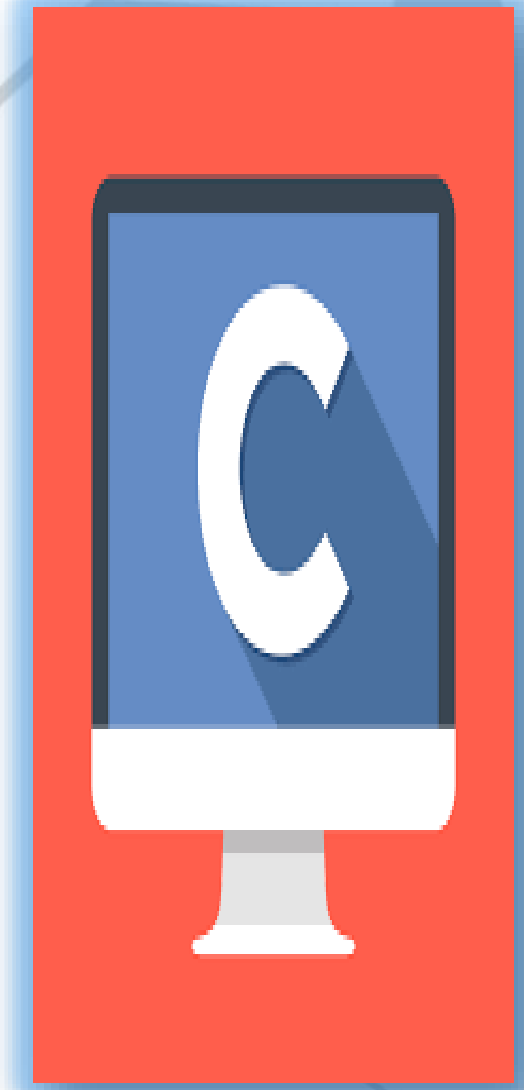
2. Global Variable

- A variable that is declared outside the function or block and must be declared at the start of the block.

- **Example:**

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

TYPES OF VARIABLES IN C

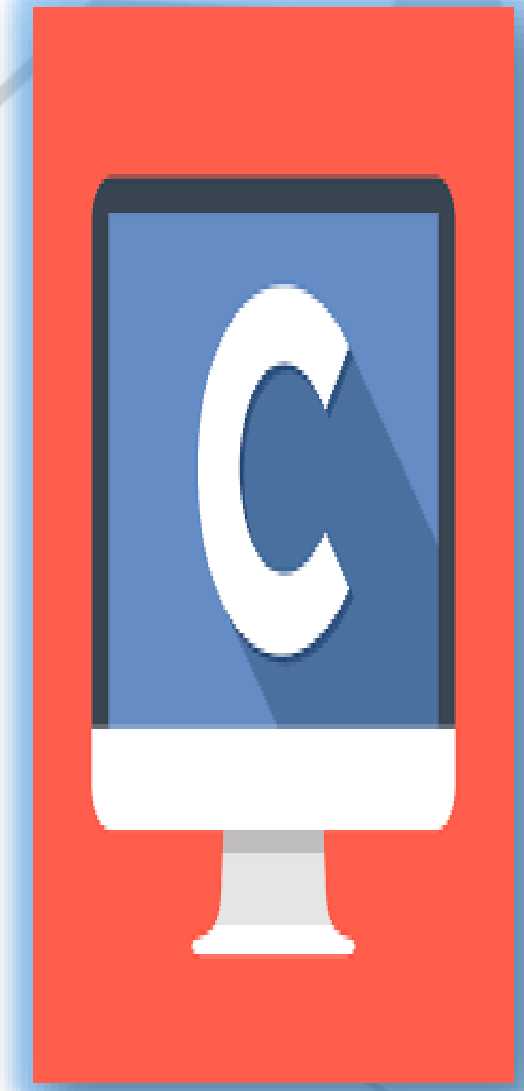


3. Static Variable

- A variable that is declared with the static keyword and it retains its value between multiple function calls.
- **Example:**

```
void function1(){
        int x=10;    // local variable
        static int y=10;   // static variable
        x=x+1;
        y=y+1;
        printf("%d,%d", x,y);
}
```

TYPES OF VARIABLES IN C



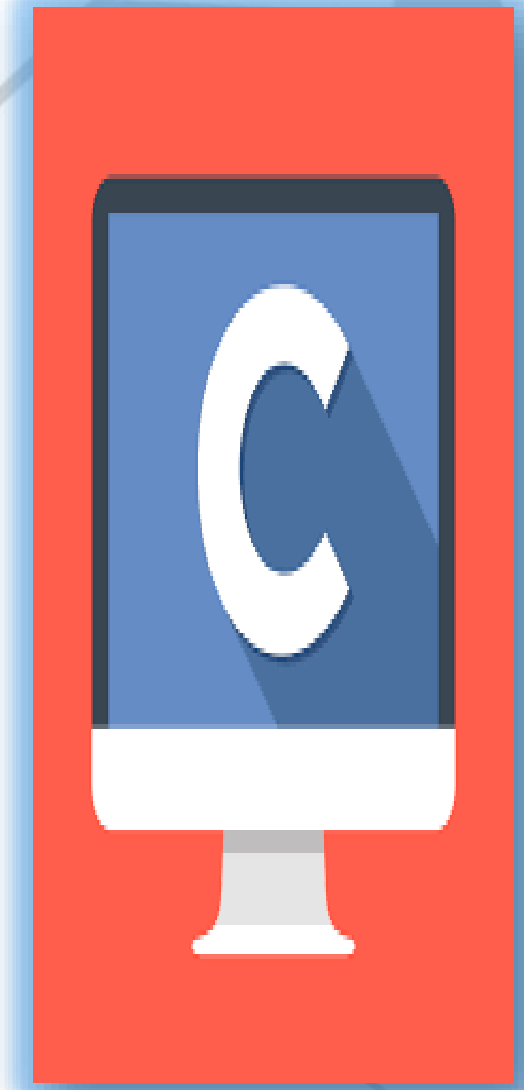
4. Automatic Variable

- All variables in C that are declared inside the block by default and can be explicitly declared using **auto keyword**.

- **Example:**

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

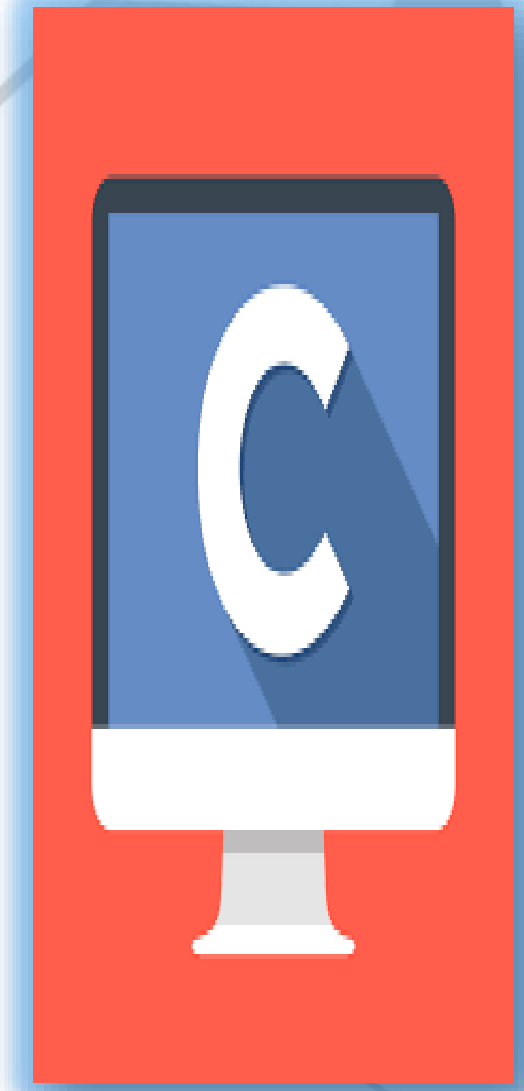
TYPES OF VARIABLES IN C



5. External Variable

- Used to share a variable in multiple C source files
- Can be declared using **extern keyword**

TYPES OF VARIABLES IN C



- **Example:**

myfile.h

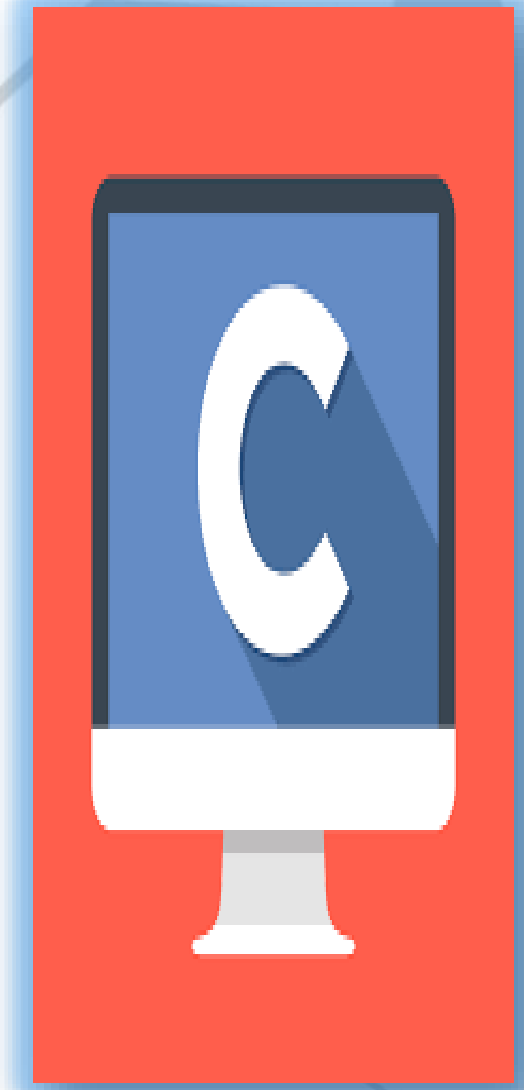
```
extern int global_var = 10; // external variable (also global)
```

program1.c

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

void printValue()
{
        printf("Global variable: %d", global_var);
}
```

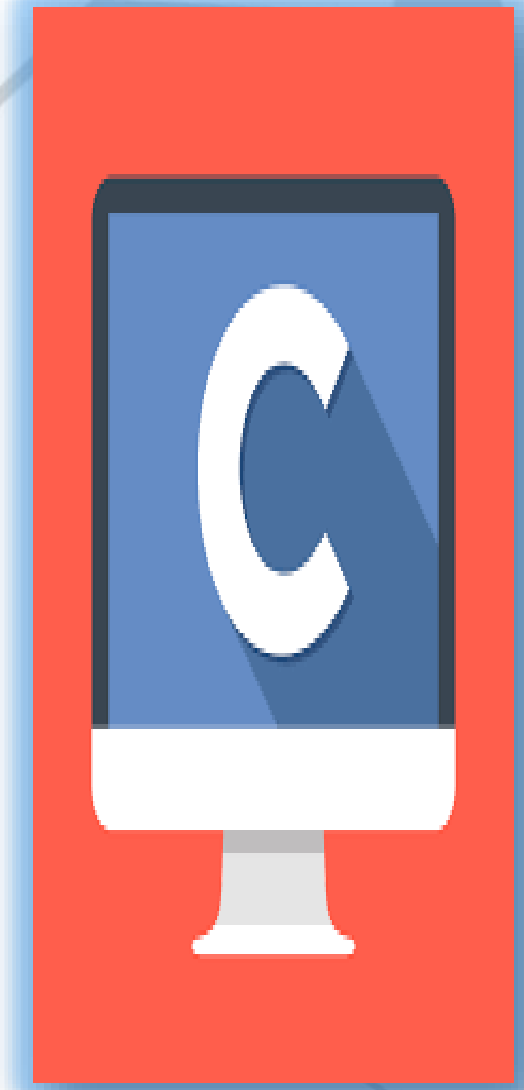
VARIABLE DECLARATION IN C



- Declaring some space for a variable which will be used to store data

- **SYNTAX:**
  - **Simple Declaration:**
    - **data_type variableName;**
  - **Declaration with Initialization:**
    - **data_type variableName =** {variable | literal | constant | expression}**;**
  - **Single-line Multiple Variable Declaration of Same Type:**
    - **data_type variable1Name, variable2Name, variable3Name;**

VARIABLE DEFINITION IN C



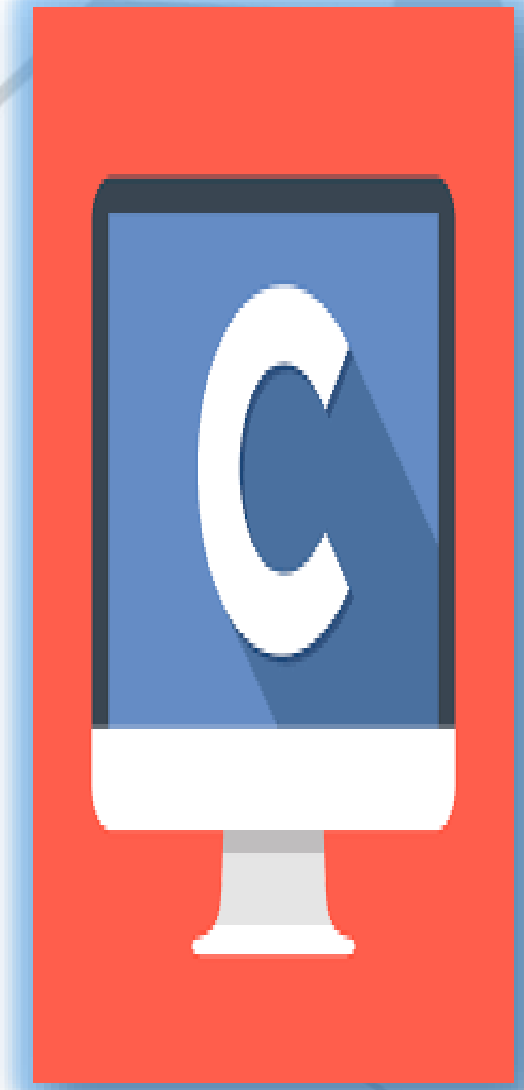
- **int** width, height = 5;
  width = 8;

- **char** letter = 'A';

- **float** age, area;
  age = 51;
  area = l * w;

- **double** d;
  d = 10;

VARIABLE ASSIGNMENT



- A process of assigning a value to a variable.

- **Example:**

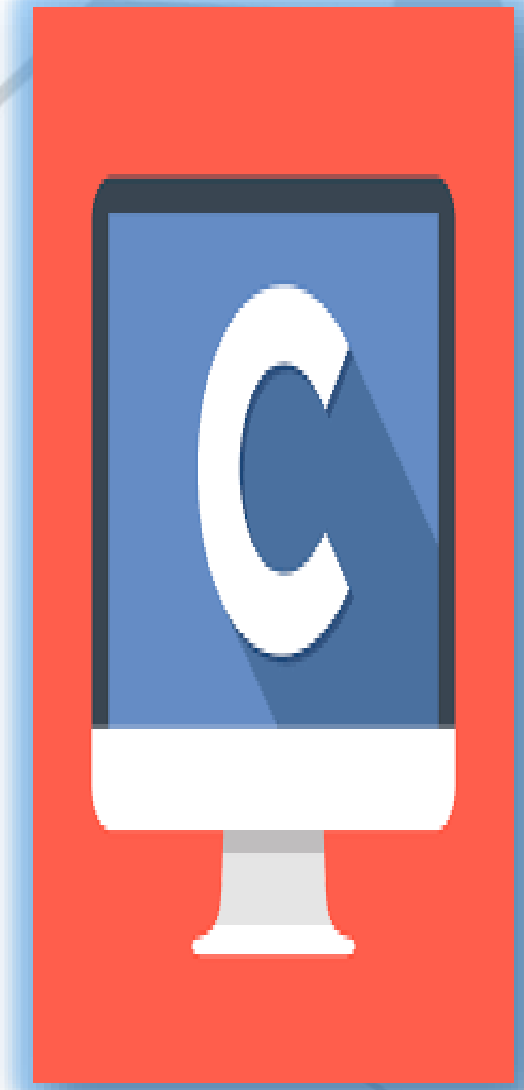
```
int width = 60;
int age = 31;
```

COMPONENTS OF C LANGUAGE: IDENTIFIER

❑ Identifier:

- A **name** given to entities such as:
  - **constants**
  - **variables**
  - **structures**
  - **functions**
  - **others**

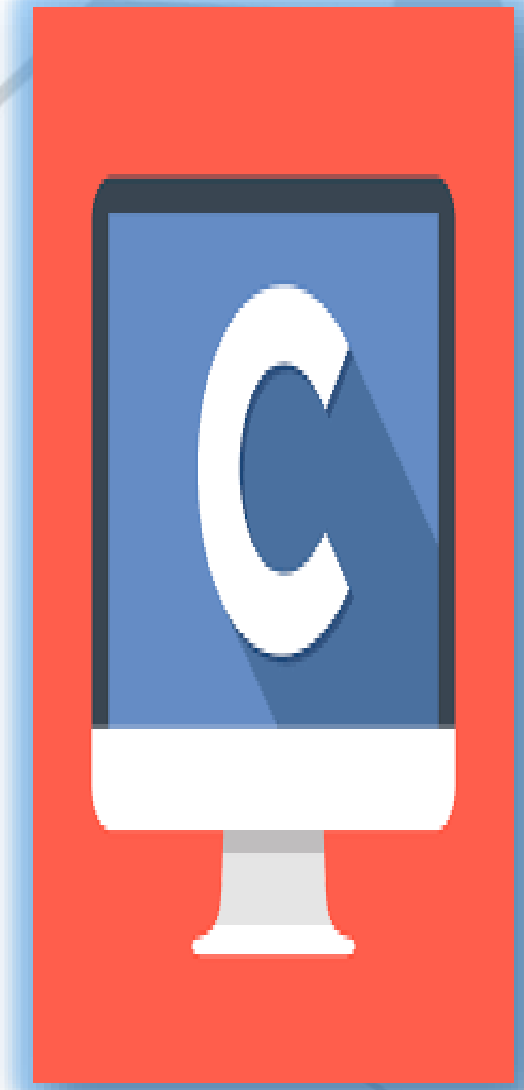
COMPONENTS OF C LANGUAGE: IDENTIFIERS



❑ Rules for Naming Identifiers:

- An **identifier** may contain capital letters A-Z, lowercase letters a-z, digits 0-9, and the underscore character.
- The **first character** must be an **alphabetic character** (upper case or lower case letters) or an **underscore**.
- **Cannot** be a reserved word or keyword.
- **C** is **case-sensitive** (it distinguishes between upper case and lower case letters in identifiers).

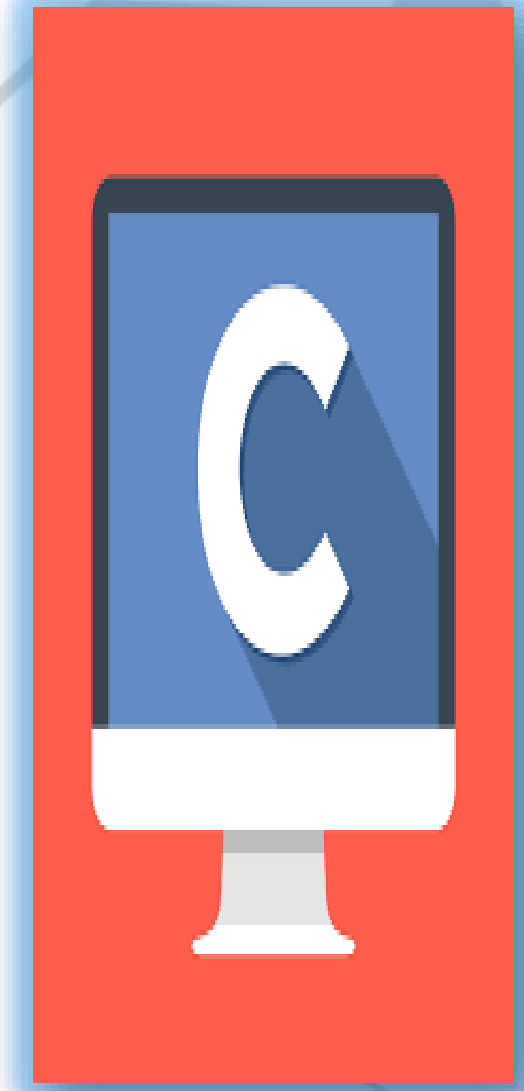
COMPONENTS OF C LANGUAGE: IDENTIFIERS



❑ Rules for Naming Identifiers:

- Identifier names must be **unique**.
- **Only** the **first thirty-one (31) characters** are **significant**.
- **Must not** contain white spaces or blank spaces.
- **Special characters** such as # and **$** are **not allowed**.

COMPONENTS OF C LANGUAGE: KEYWORDS

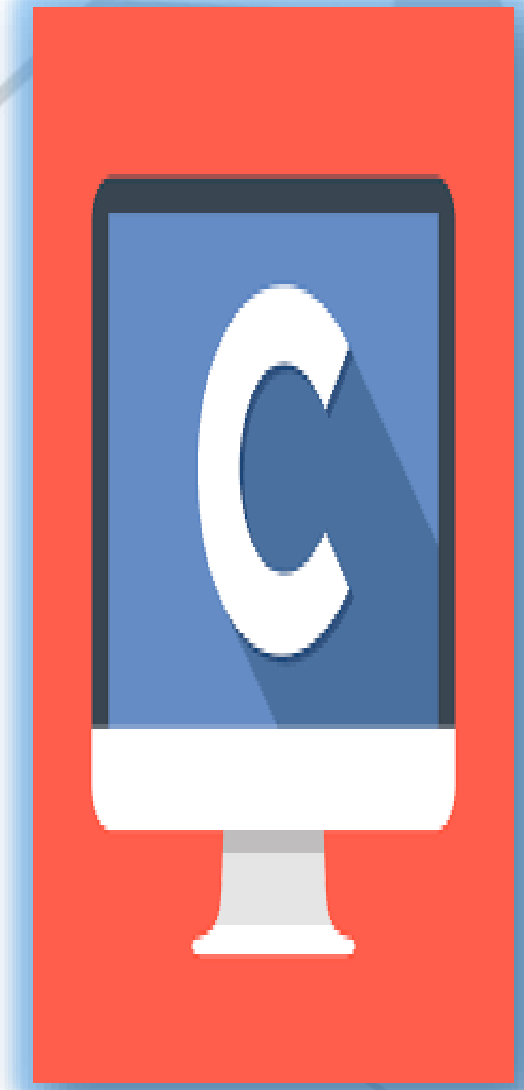


- **Keywords:**
  - **Reserved words** which have been **assigned** with **specific meanings** in the **C language**.
  - **Cannot** be used as variable name, constant name, and others.

COMPONENTS OF C LANGUAGE: 32 KEYWORDS

C LANGUAGE 32 KEYWORDS			
auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

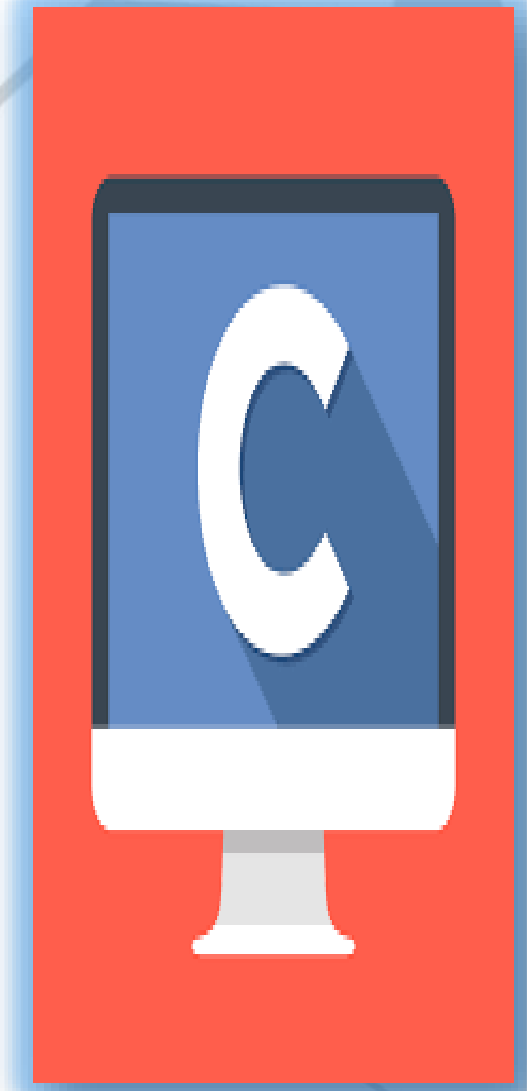
COMPONENTS OF C LANGUAGE: DATA TYPES



Data Types

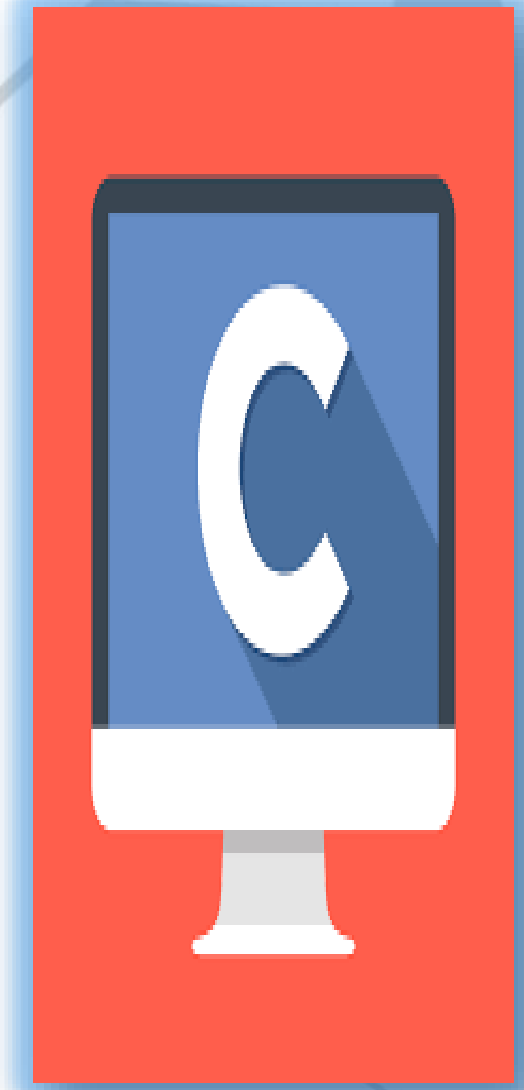
- **Specify** the **type of data** that a **variable** can **store**.

COMPONENTS OF C LANGUAGE: DATA TYPES (ANSI C)



Types	Data Types
Primary (Built-In)	char, int, float, double, void
Derived	array, references, pointers
User-Defined	structure, union, enum

COMPONENTS OF C LANGUAGE: GENERIC DATA TYPES

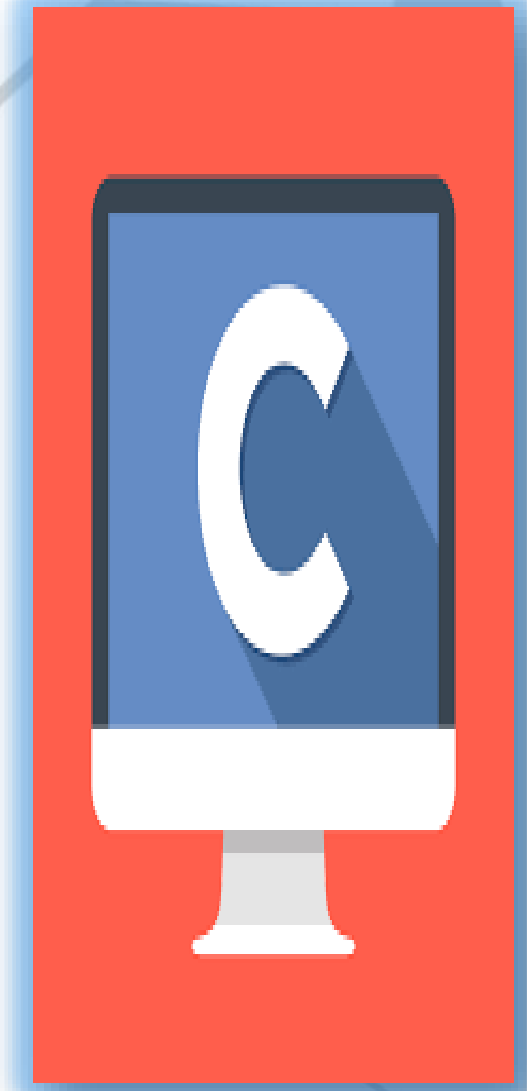


Types	Data Types
Basic (Standard)	char, int, float, double
Derived	array, pointer, structure, union
Enumeration	enum
Void	void

COMPONENTS OF C LANGUAGE: PRIMARY DATA TYPES

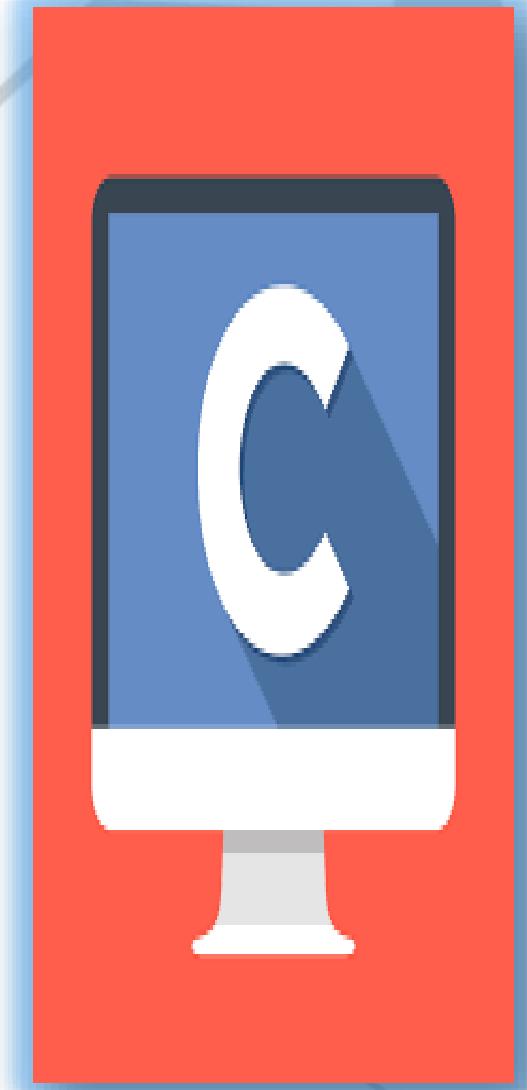
Types	Function
int	Used to declare variables that store numeric or integer values It **does not have a fractional part**
float	Used to declare variables that can take on numeric values with a fractional part Its storage space is less
double	Used for variables that can take on numeric values with a fractional part Its size is more than float
char	Used to declare variables that can **store one character** as a value
void	Holds **no value** **Void type function: will not return any value**

COMPONENTS OF C LANGUAGE: DERIVED DATA TYPES



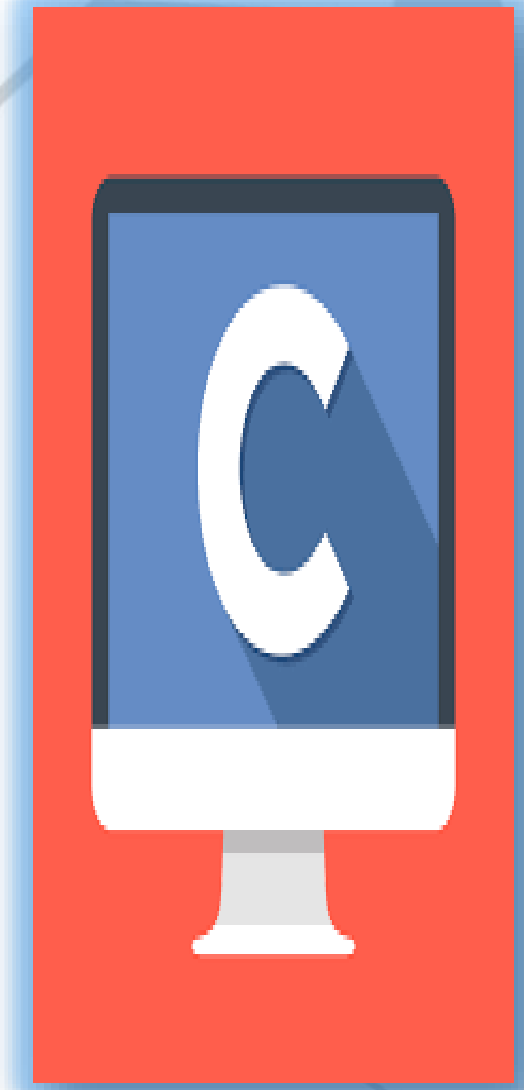
Types	Description
Arrays	Arrays are sequences of data items having homogeneous values. They have adjacent memory locations to store values.
References	Function pointers allow referencing functions with a particular signature.
Pointers	These are powerful C features which are used to access the memory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

COMPONENTS OF C LANGUAGE: USER-DEFINED DATA TYPES



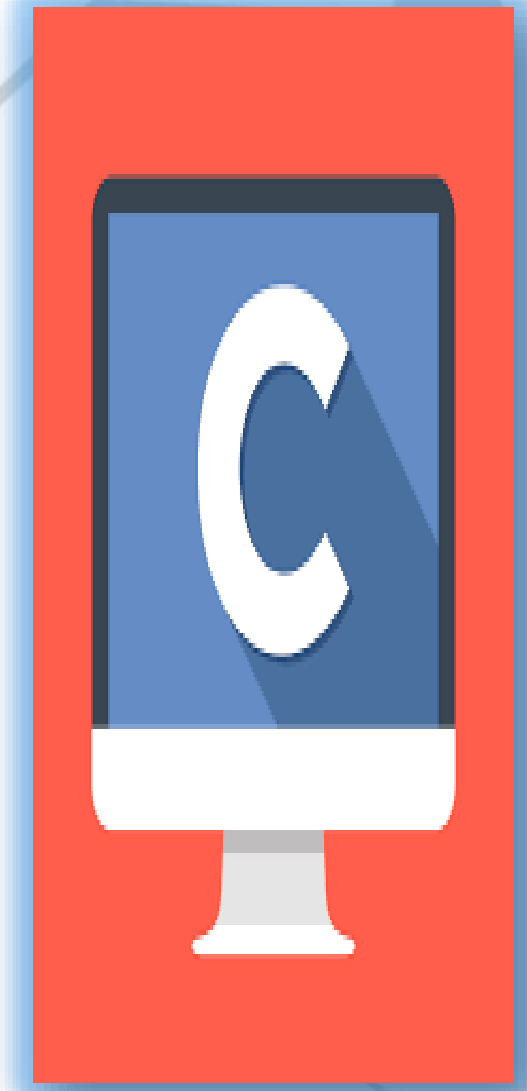
Types	Description
Structure	It is a package of variables of different types under a single name. This is done to handle data efficiently. "struct" keyword is used to define a structure.
Union	These allow storing various data types in the same memory location. Programmers can define a union with different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

COMPONENTS OF C LANGUAGE: DATA TYPES



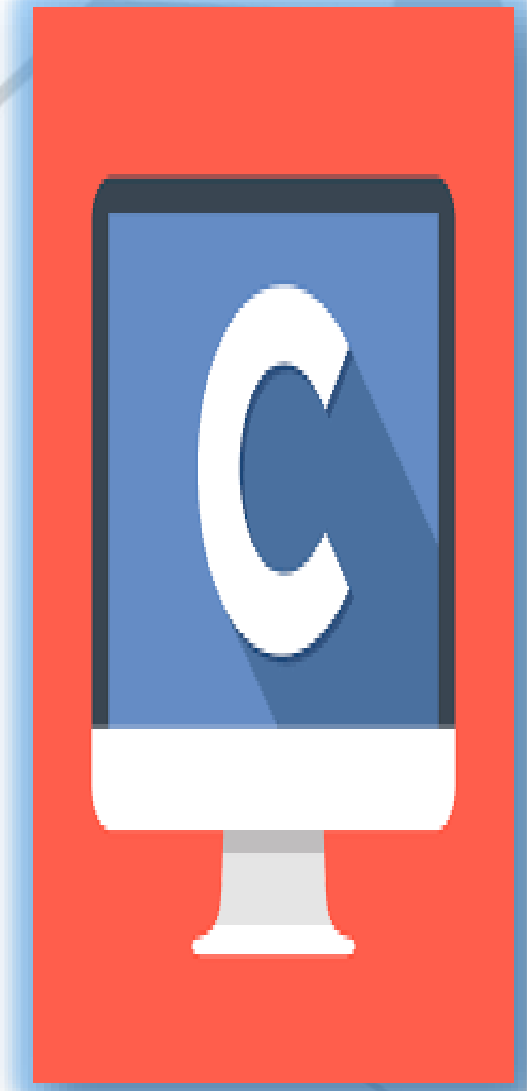
DATA TYPE	MEMORY (BYTES)	RANGE	FORMAT SPECIFIER
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu

COMPONENTS OF C LANGUAGE: DATA TYPES



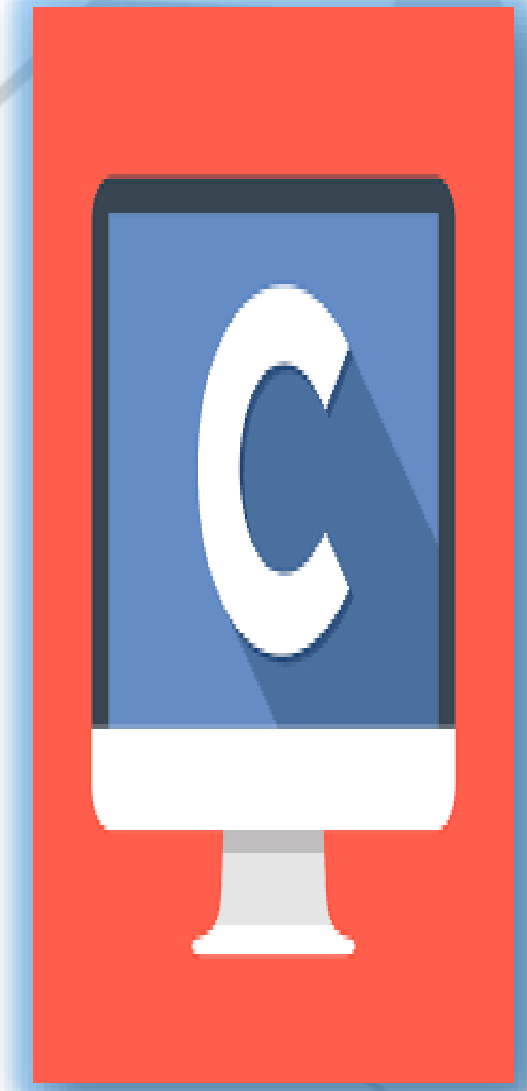
DATA TYPE	MEMORY (BYTES)	RANGE	FORMAT SPECIFIER
char	1	-128 to 127	%c
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	10 / 12 / 16		%Lf

COMPONENTS OF C LANGUAGE: DATA TYPES



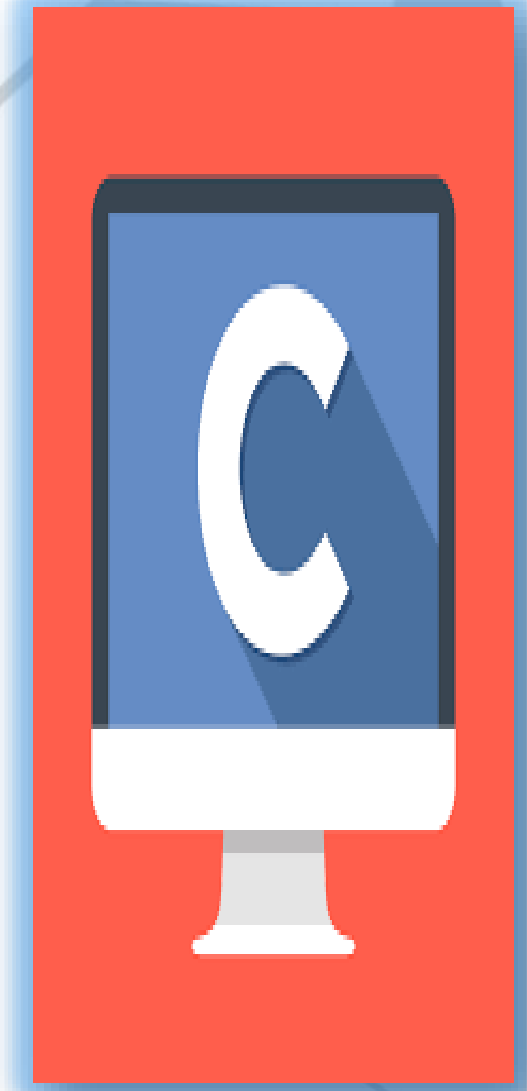
DATA TYPE	MEMORY (BYTES)	RANGE	FORMAT SPECIFIER
char	1	-128 to 127	%c
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	10 / 12 / 16		%Lf

OPERATORS : ORDER OF PRECEDENCE



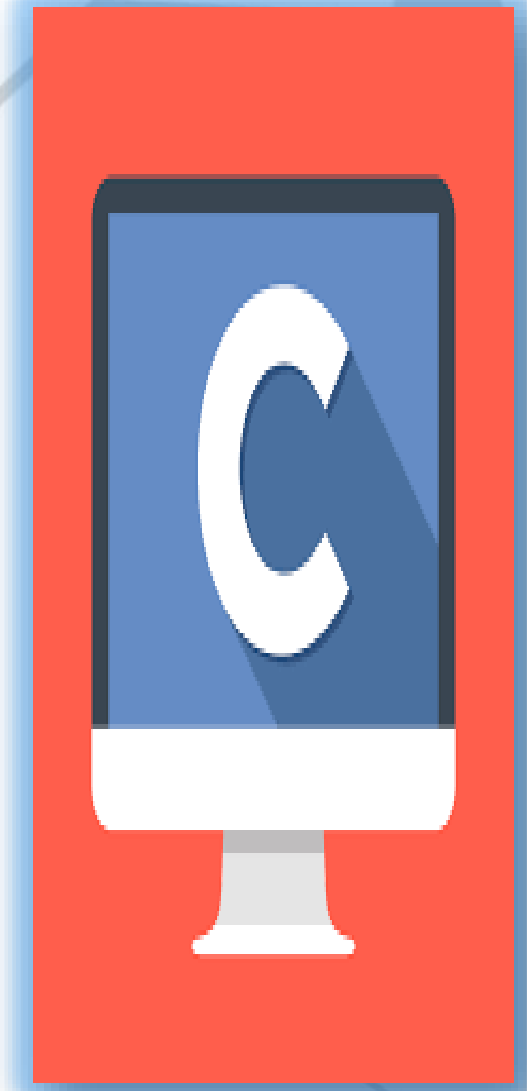
OPERATOR	DESCRIPTION	ASSOCIATIVITY
( )	Parentheses (function call)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ —	Postfix increment/decrement	
++ —	Prefix increment/decrement	right-to-left
+ –	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(type)	Cast (convert value to temporary value of type)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	

OPERATORS



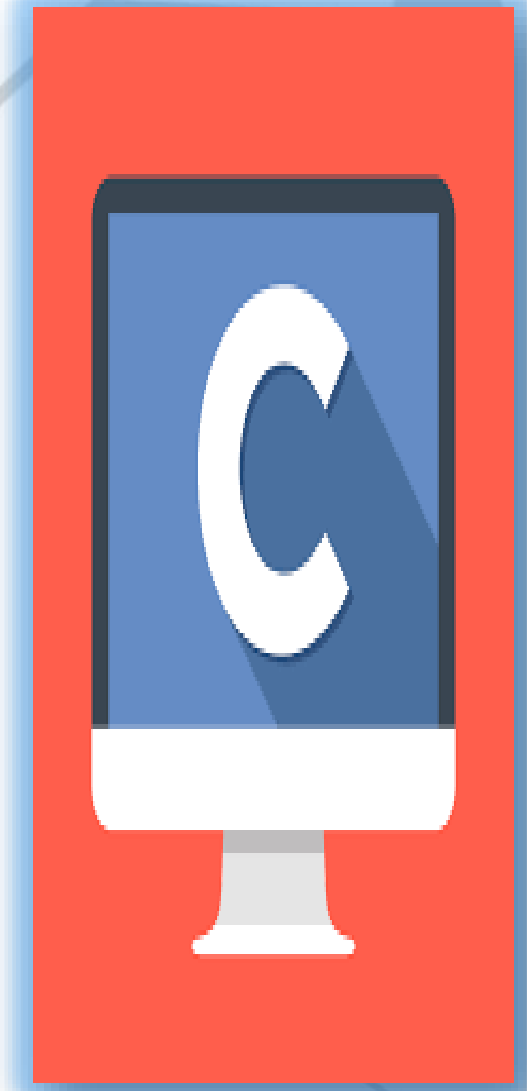
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	
<< >>	Bitwise shift left, Bitwise shift right	
< <=	Relational less than/less than or equal to	
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	
&	Bitwise AND	
^	Bitwise exclusive OR	
|	Bitwise inclusive OR	
&&	Logical AND	
||	Logical OR	

OPERATORS



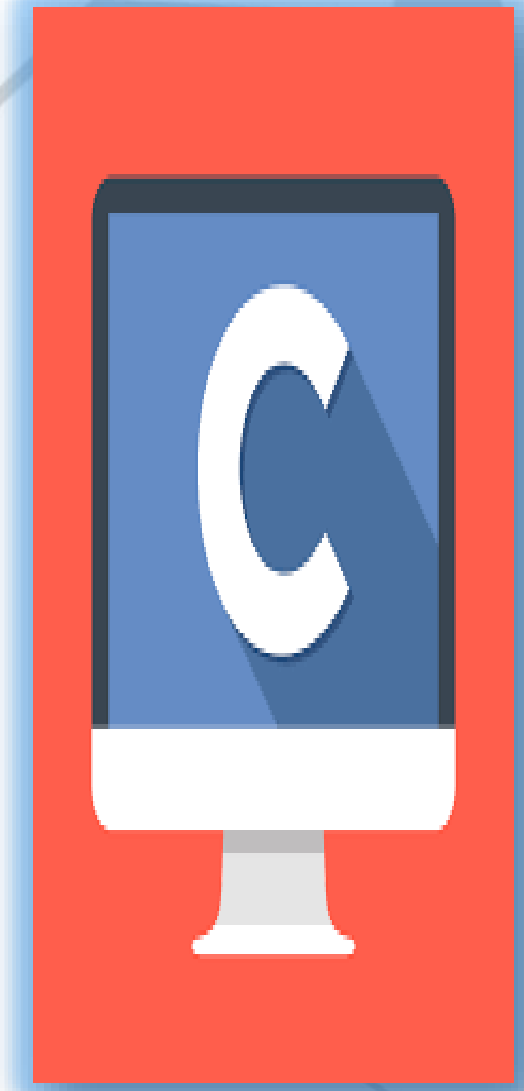
? :	Ternary conditional	right-to-left
=	Assignment	
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

OPERATORS : UNARY



Unary Opertors	operators which require single operand to perform any action	
sizeof()	returns the byte size of an operand	
increment (++)	adds 1 to the operand	
	post-increment	increments value by 1 after assigning the value to the variable
	pre-increment	increments value by 1 before assigning the value to the variable
decrement (--)	deducts 1 to the operand	
	post-decrement	decrements value by 1 after assigning the value to the variable
	pre-decrement	decrements value by 1 before assigning the value to the variable
unary minus (-)	negates the value	

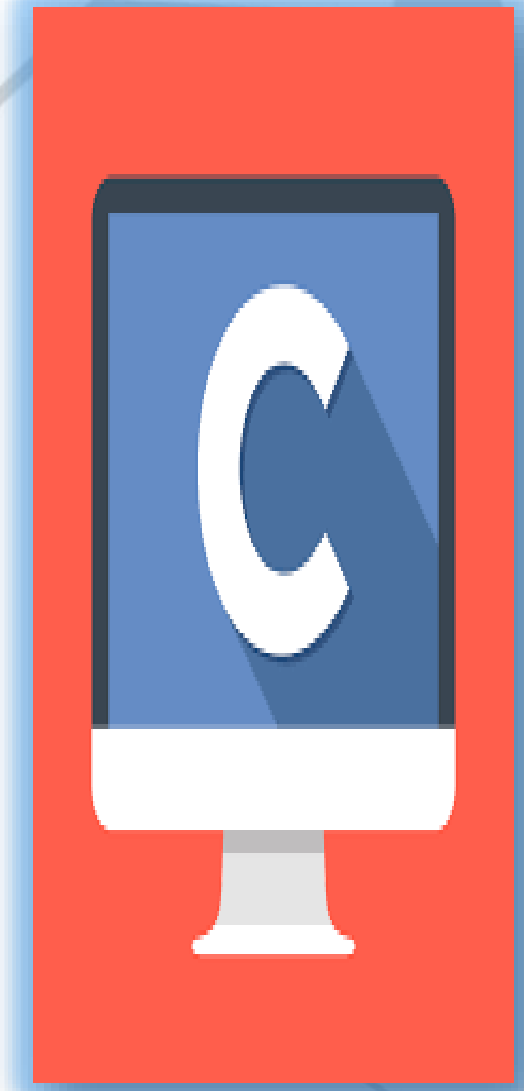
OPERATORS : BINARY



Binary Operators	operators which require two operands to perform any action

Arithmetic Operator	Name	Description
+	Addition	Adds two operands
–	Subtraction	Subtracts two operands
*	Multiplication	Multiplies two operands
/	Division	Divides first operand by second
%	Modulus	Returns the remainder of division

OPERATORS : BINARY



```
>>>1 + 3
4
>>>10 – 4
6
>>>4 * 2
8
>>>10 / 2
5
>>>17 % 5
2
```

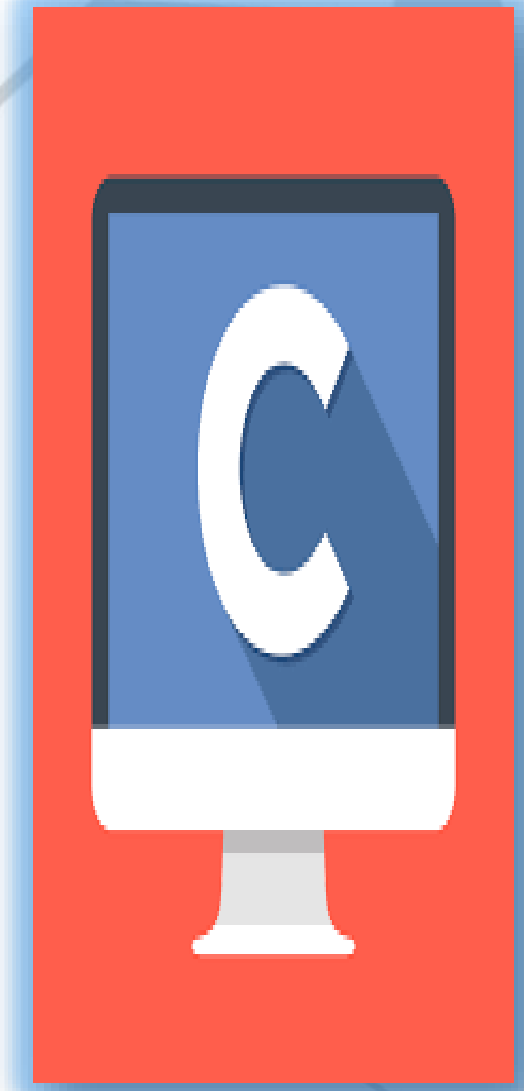
OPERATORS : BINARY

Relational Operator	Name	Description
= =	Is equal to	Checks whether the two operands are equal or not. If so, it returns true. Otherwise, it returns false.
!=	Is not equal to	Checks whether the two operands are equal or not. If so, it returns true. Otherwise, it returns false.
>	Greater than	Checks whether the first operand is greater than the second operand. If so, it returns true. Otherwise it returns false

OPERATORS : BINARY

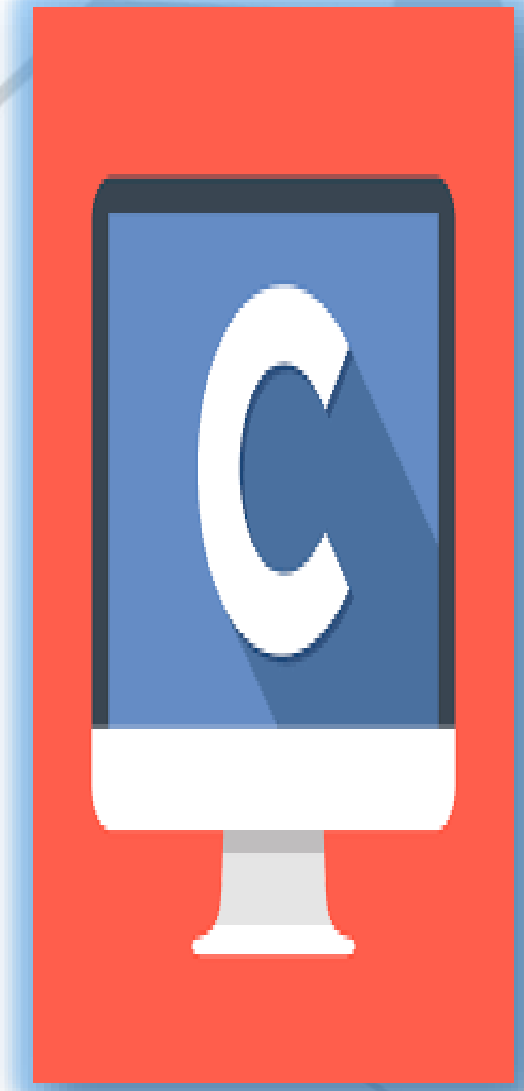
<	Less than	Checks whether the first operand is lesser than the second operand. If so, it returns true. Otherwise, it returns false.
>=	Greater than or equal to	Checks whether the first operand is greater than or equal to the second operand. If so, it returns true. Otherwise, it returns false.
< =	Less than or equal to	Checks whether the first operand is lesser than or equal to the second operand. If so, it returns true. Otherwise, it returns false.

OPERATORS : BINARY



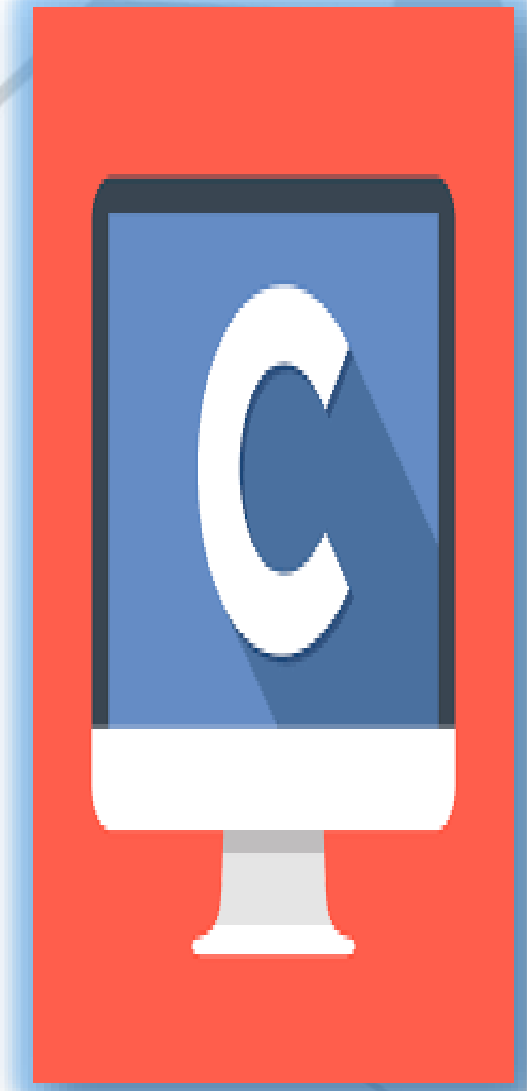
```
>>> 8 == 6+2
        True
>>> 6 != 6
        False
>>> -1 > 0
        False
>>> 7 >= 5
        True
```

OPERATORS : BINARY



Logical Operator	Name	Description
&&	logical AND	It returns true when both conditions are true
||	logical OR	It returns true when at-least one of the condition is true
!	logical NOT	It is used to reverse state of its operand. If a condition is true, then Logical NOT operator will make it false.

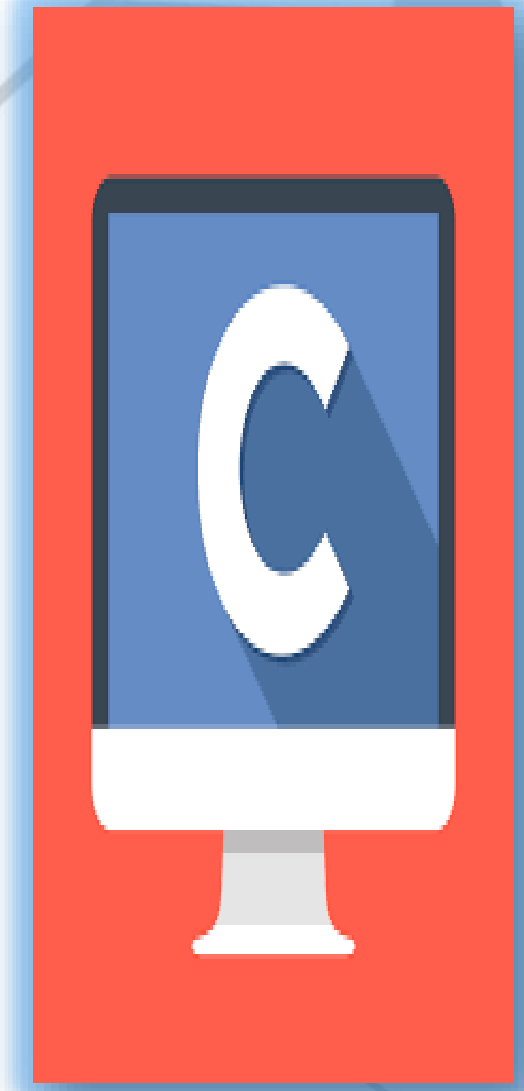
OPERATORS : BINARY



TRUTH TABLE

A	B	(A and B)	(A or B)	not(A and B)	not(A or B)
True	True	True	True	False	False
True	False	False	True	True	False
False	True	False	True	True	False
False	False	False	False	True	True

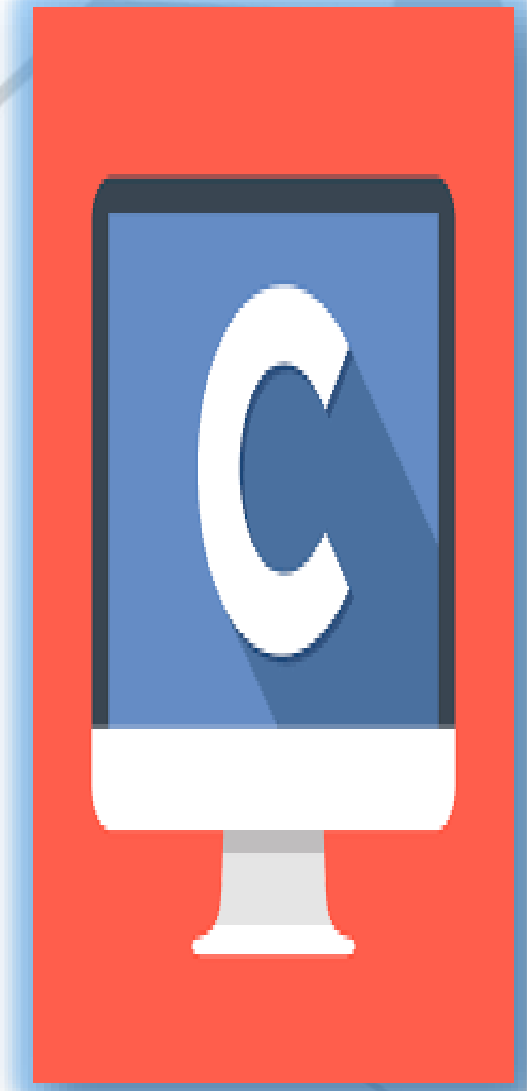
OPERATORS : BINARY



```
>>> (8>9) and (2<9)
False
>>> (2>1) and (2>9)
False
>>> (2==2) or (9<20)
True
>>> (3!=3) or (9>20)
False
```

```
>>> not (8 > 2)
False
>>> not (2 > 10)
True
```

OPERATORS : BINARY

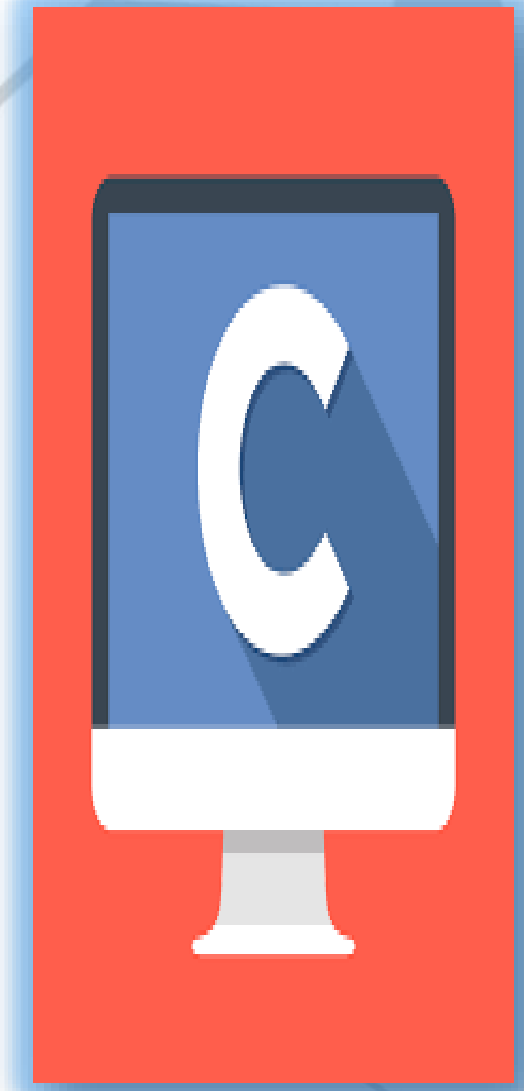


Operator	Name	Description
&	Bitwise AND	The result of a&b is 1 only if both bits are 1.
|	Bitwise OR	The result of OR is 1 any of the two bits is 1
^	Bitwise XOR	The result of XOR is 1 if the two bits are different.
>>	Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.

OPERATORS : BINARY

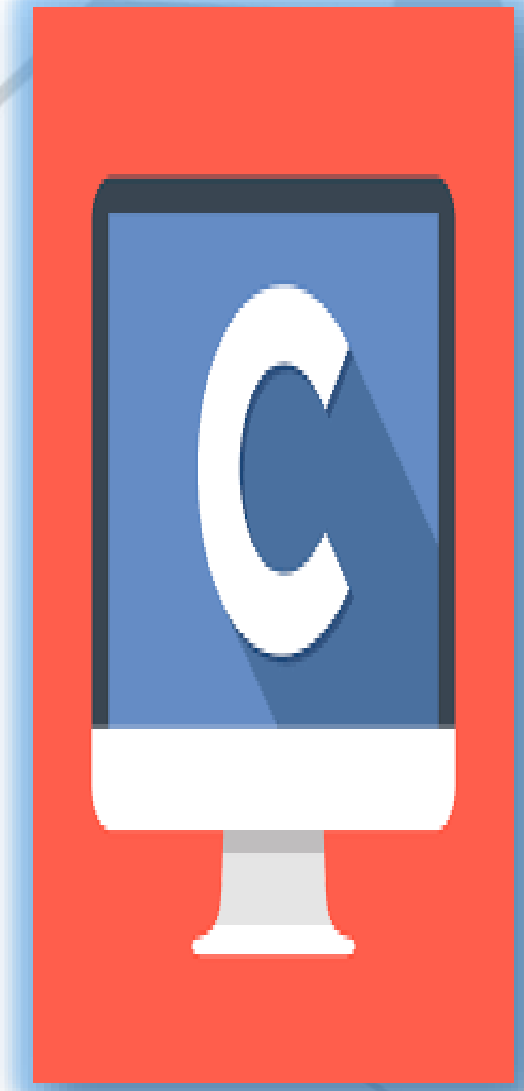
<<	Binary Left Shift	The left operands value is moved left by the number of bits specified by left operand
~	Binary One's Complement	Inverts all bits.

OPERATORS : BINARY



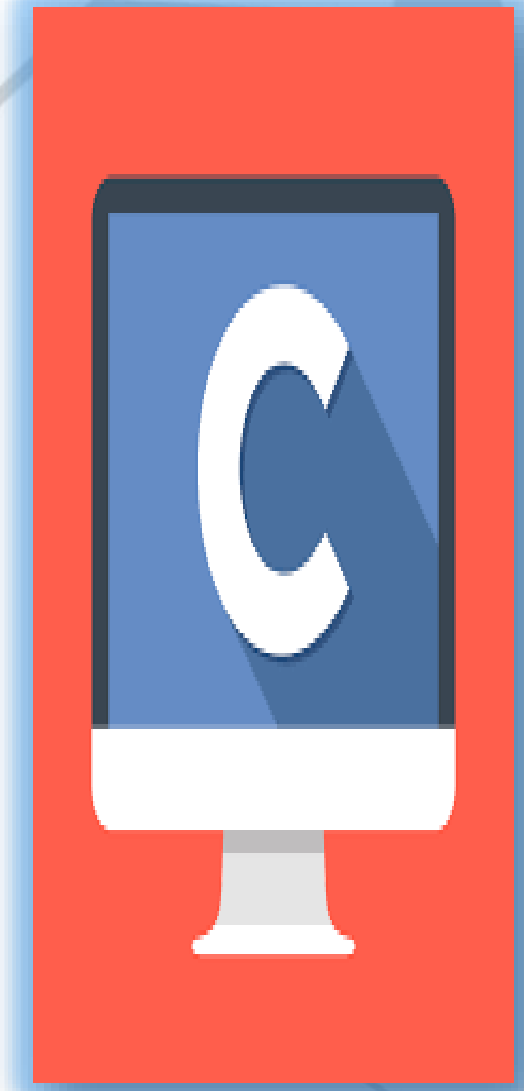
Assignment Operator	Name
=	Assignment operator
+=	Increment, then assign
– =	Decrement, then assign
* =	Multiply, then assign
/=	Divide, then assign
%=	Modulus, then assigns

OPERATORS : BINARY



Special Operator	Description
&	It returns the address of a variable
*	It is used as pointer to a variable

OPERATORS : TERNARY



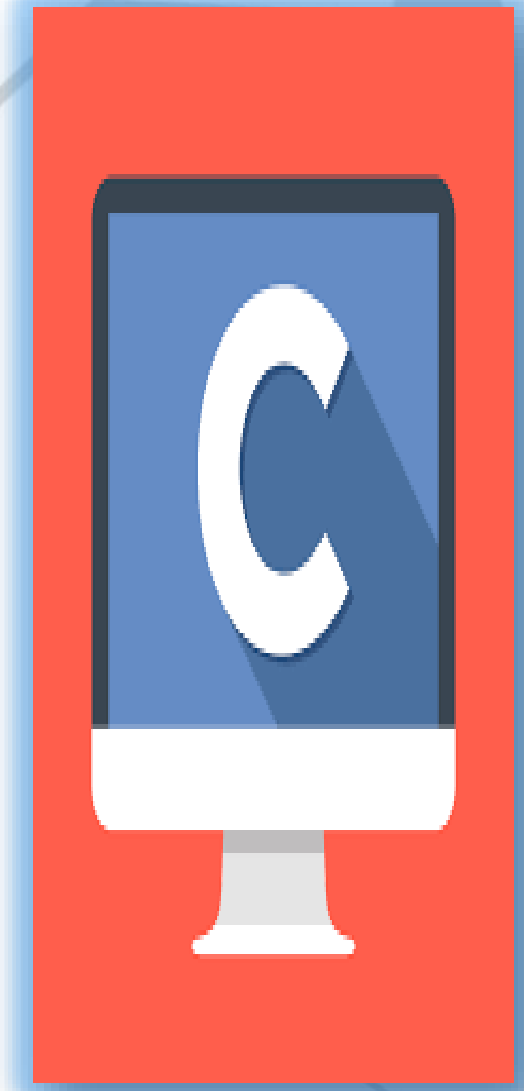
Ternary Operators	operators which is also called as “Conditional Operator”

Syntax: Condition ? expression1 : expression2

Sample:
int val = 100 < 99 ? 100 : 200;
printf(“%d”, val);

Output:
200

INPUT FUNCTION : scanf( )



- **scanf() function:**
  - used for input
  - used to read a character, string, numeric data from keyboard
  - an inbuilt library function defined in stdio.h (header file)
  - **standard input:** keyboard
  - **Syntax:**

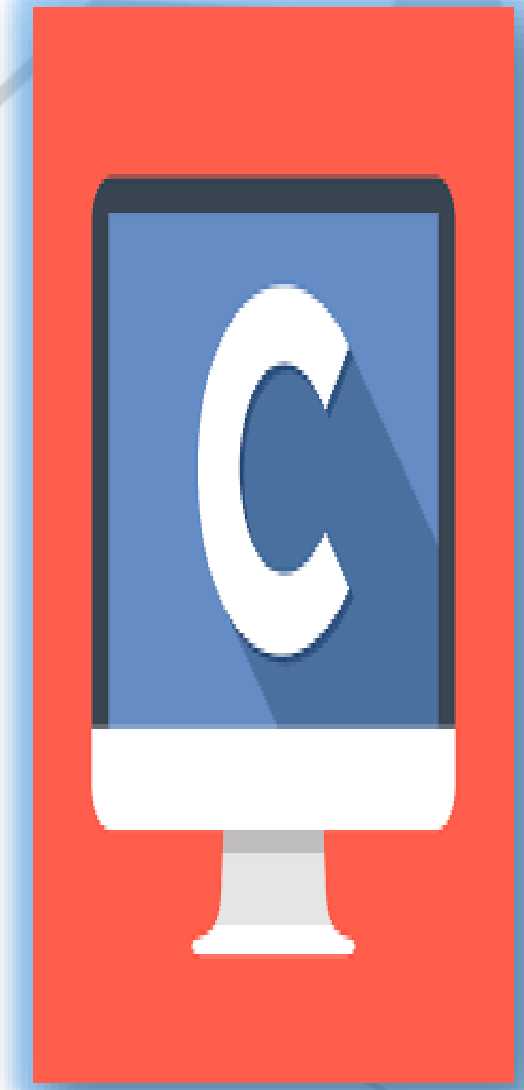
```
scanf("format specifier",&argument_list);
```

OUTPUT FUNCTION : printf( )

- **printf() function:**
  - used for output
  - used to print the character, string, float, integer, and other values onto the output screen
  - an inbuilt library function defined in stdio.h (header file)
  - **standard output:** screen
  - **Syntax:**

```
printf("format specifier",argument_list);
```

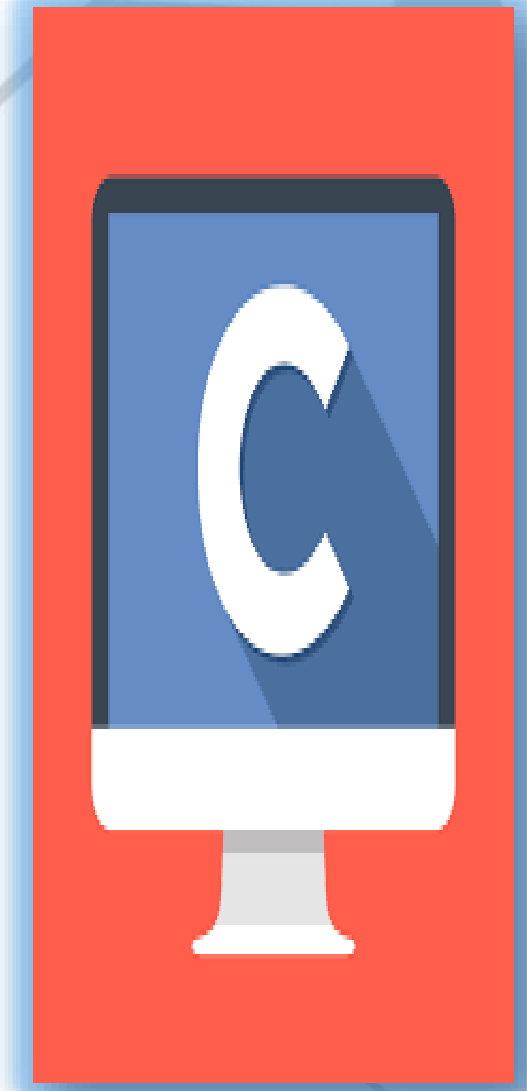
FORMAT SPECIFIER



- Used during input and output functions.

- A **way** to **tell the compiler** what **type of data** is in a **variable** during **taking input** using **scanf()** or **printing** using **printf**()

FORMAT SPECIFIERS FOR scanf( ) and printf( )



FORMAT SPECIFIER	DESCRIPTION
%d	Integer Format Specifier
%f	Float Format Specifier
%c	Character Format Specifier
%s	String Format Specifier
%u	Unsigned Integer Format Specifier
%ld	Long Int Format Specifier
%lf	Double Format Specifier

Comments

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

1.Single Line Comments

Syntax:

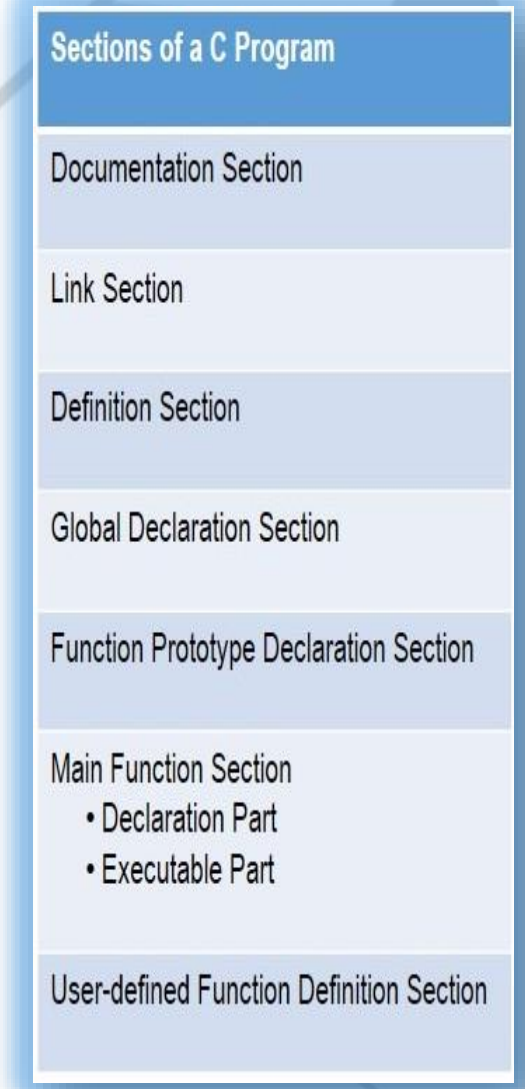
```
//comments in here
```

2.Multi-Line Comments

Syntax:

```
/* Name
   Program description
*/
```

BASIC STRUCTURE OF C PROGRAM



- Documentation Section
- Link Section (Preprocessor Statements)
- Definition Section
- Global Declaration Section
- Function Prototype Declaration Section
- The main( ) Function Section
  - Declaration Part (Local Declarations)
  - Executable Part (Program Statements & Expressions)
- User-defined Function Definition Section

BASIC STRUCTURE OF C PROGRAM

Sections of a C Program
Documentation Section
Link Section
Definition Section
Global Declaration Section
Function Prototype Declaration Section
Main Function Section • Declaration Part • Executable Part
User-defined Function Definition Section

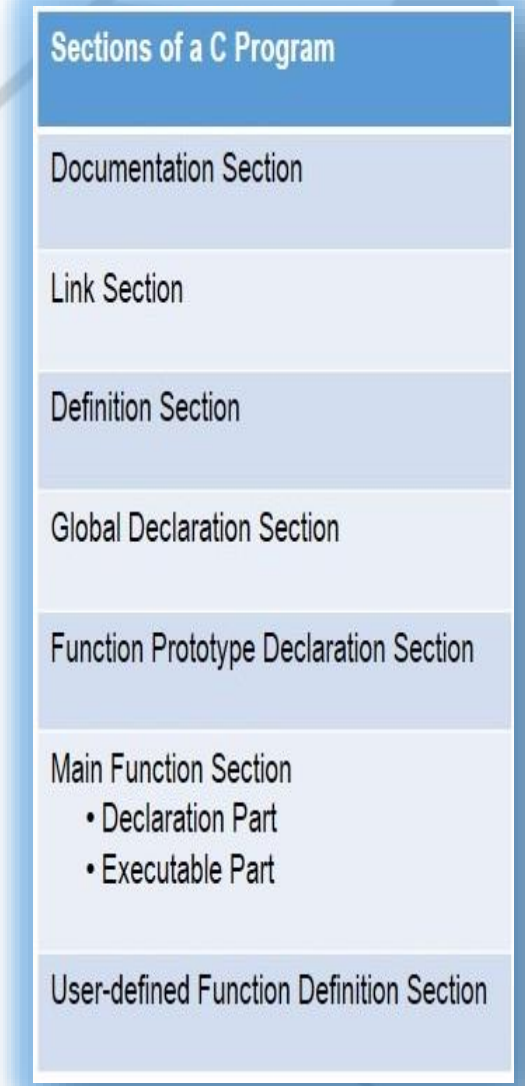
- **Documentation Section:**
  - Consists of **comment lines** which include the **program description**, the **name of the programmer**, **details** like **time & date of writing the program, and other needed information.**
  - Helps anyone to get an **overview** of the program

BASIC STRUCTURE OF C PROGRAM

Sections of a C Program
Documentation Section
Link Section
Definition Section
Global Declaration Section
Function Prototype Declaration Section
Main Function Section • Declaration Part • Executable Part
User-defined Function Definition Section

- **Link Section:**
  - Consists of the **header files of the functions** that are **used in the program**
  - Provides **instructions to the compiler** to **link functions** from the **system library**

SOME OF C HEADER FILES



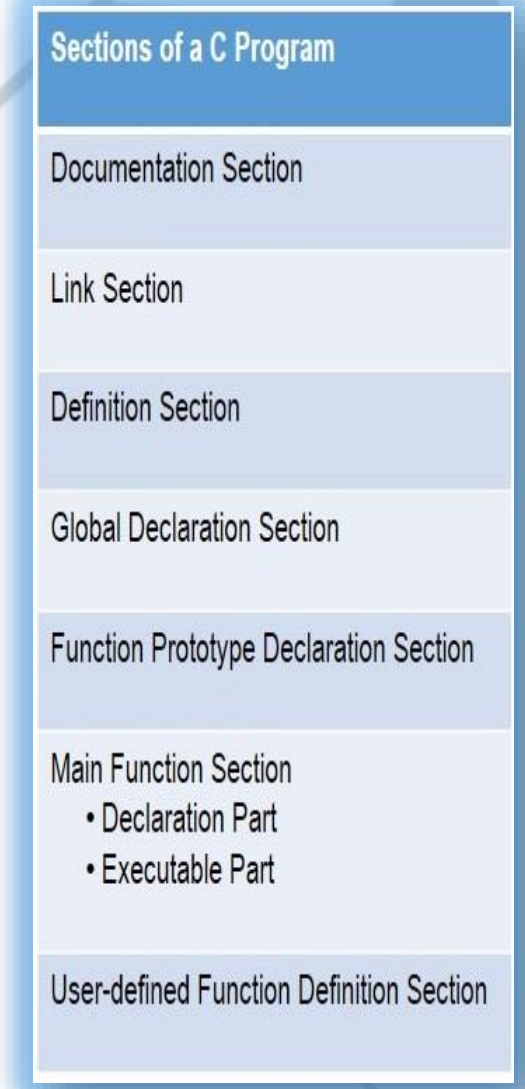
- **stdio.h**
  - Defines core input and output functions
- **stdlib.h**
  - Defines numeric conversion functions, pseudo-random network generator, memory allocation
- **stdint.h**
  - Defines exact width integer types
- **string.h**
  - Defines string handling functions
- **math.h**
  - Defines common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

THE PREPROCESSOR IN C

Sections of a C Program
Documentation Section
Link Section
Definition Section
Global Declaration Section
Function Prototype Declaration Section
Main Function Section • Declaration Part • Executable Part
User-defined Function Definition Section

- Used to process the C program before compiling
- Provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control.
- A separate **program** invoked by the compiler as the first part of translation.

THE PREPROCESSOR DIRECTIVE IN C

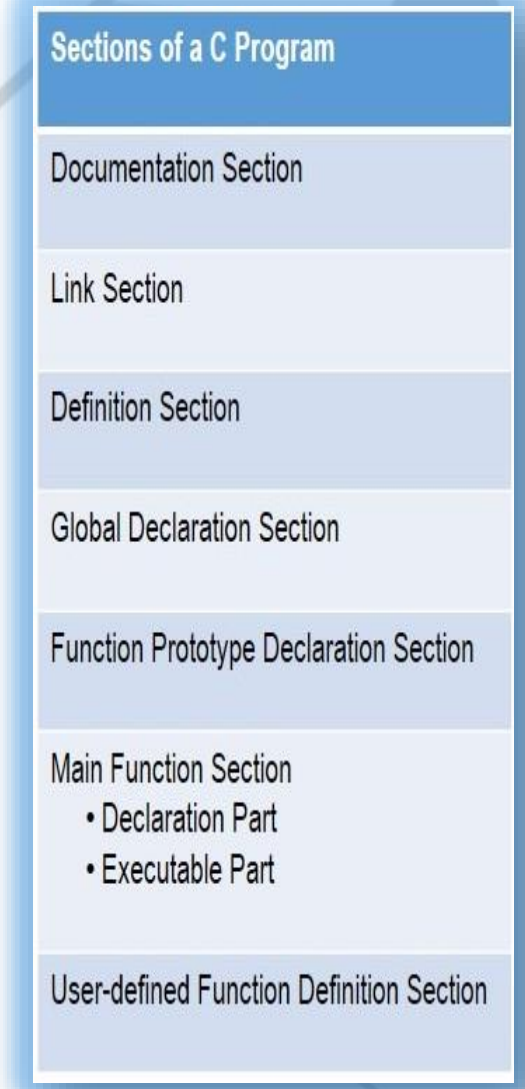


❑ Preprocessor directives:

- **Lines** included in a program that begin with the character #, which make them different from a typical source code text.

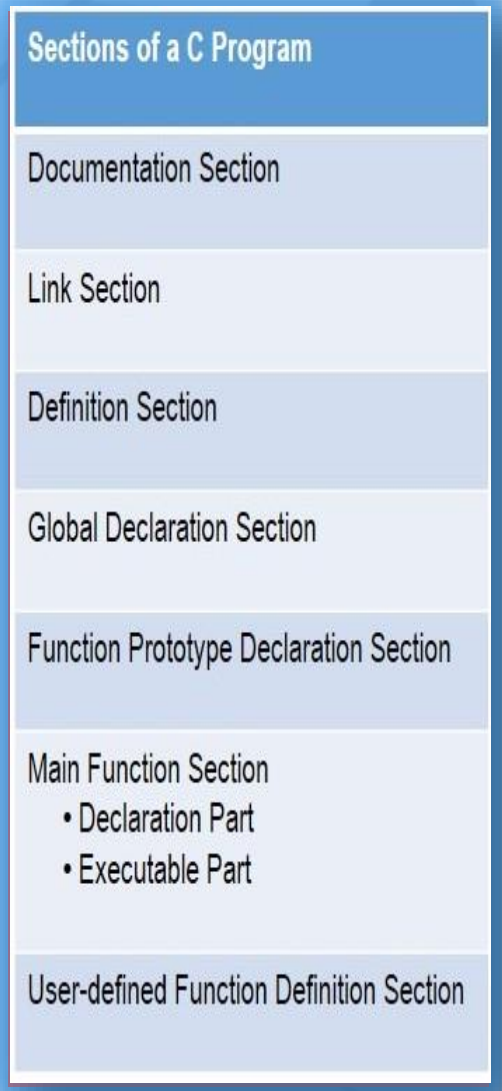
- Commands used in preprocessor

THE PREPROCESSOR DIRECTIVE IN C



PREPROCESSOR	SYNTAX DESCRIPTION
Macro	**Syntax:** #define This macro defines constant value and can be any of the basic data types.
Header file inclusion	**Syntax:** #include <file_name> The source code of the file "file_name" is included in the main program at the specified place.
Conditional compilation	**Syntax:** #ifdef, #endif, #if, #else, #ifndef Set of commands are included or excluded in source program before compilation with respect to the condition.
Other directives	**Syntax:** #undef, #pragma #undef is used to undefine a defined macro variable. #pragma is used to call a function before and after main function in a C program.

THE #include DIRECTIVE IN C

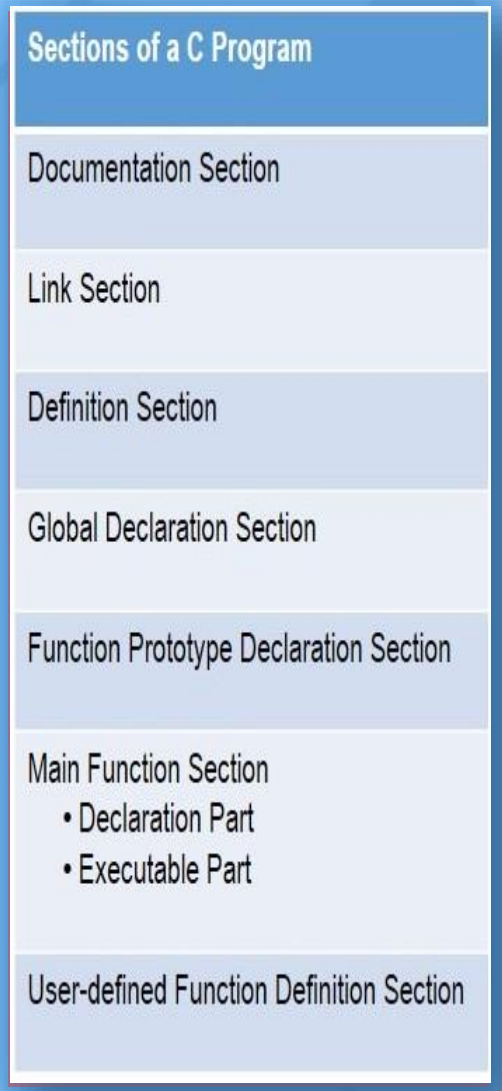


- A **#include** directive tells the preprocessor to insert the contents of another file into the source code at the point where the **#include** directive is found.

- **Syntax:**

  #include <*header_file*>

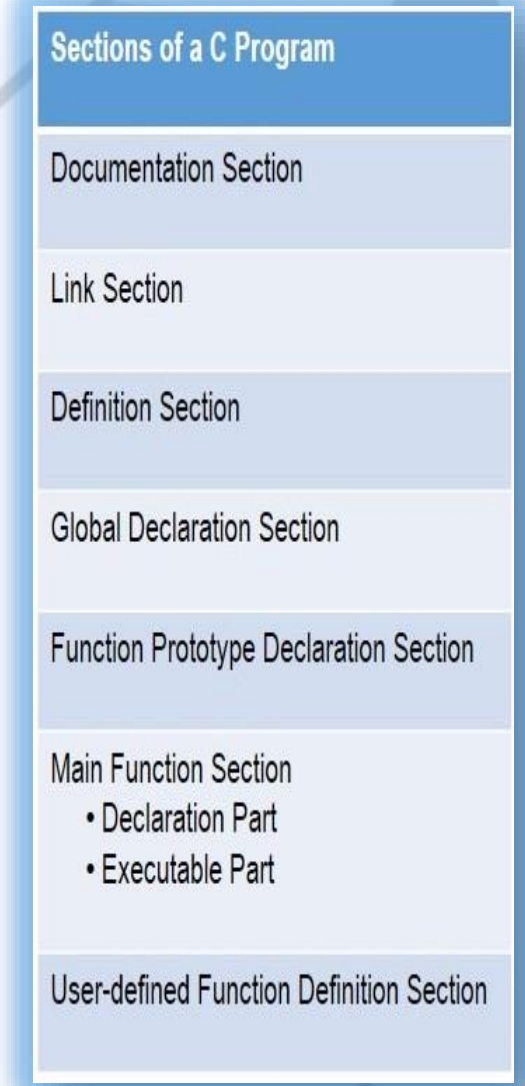
THE stdio.h HEADER FILE IN C



- The **stdio.h** header:
  - "<stdio.h>" contains declaration of ***printf()*** and ***scanf()***
  - **stdio** stands for **Standard Input Output**.
  - **.h** is an extension which denotes that the file is a Header file.

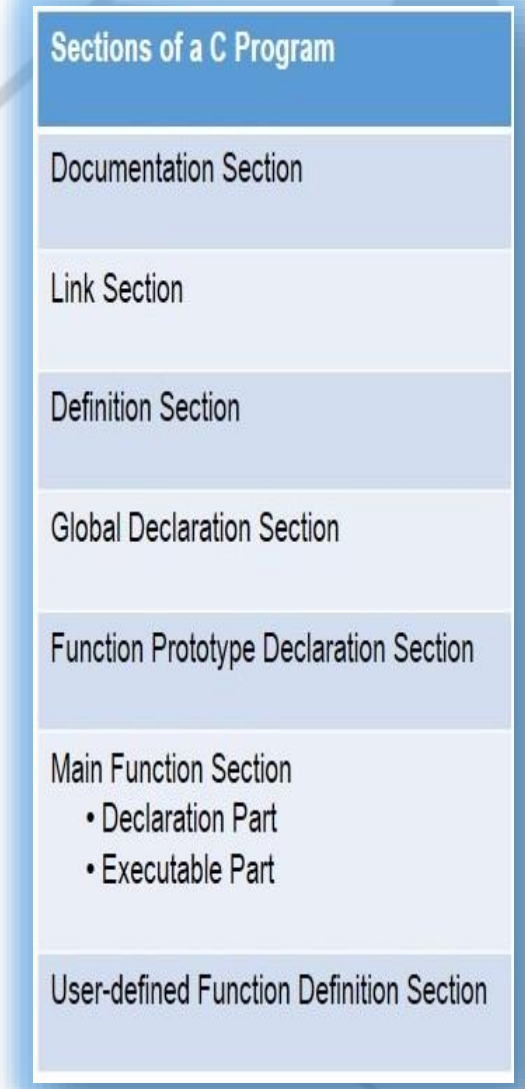
- **Header file:**
  - A library which contains
  - a set of predefined methods or functions.

BASIC STRUCTURE OF C PROGRAM



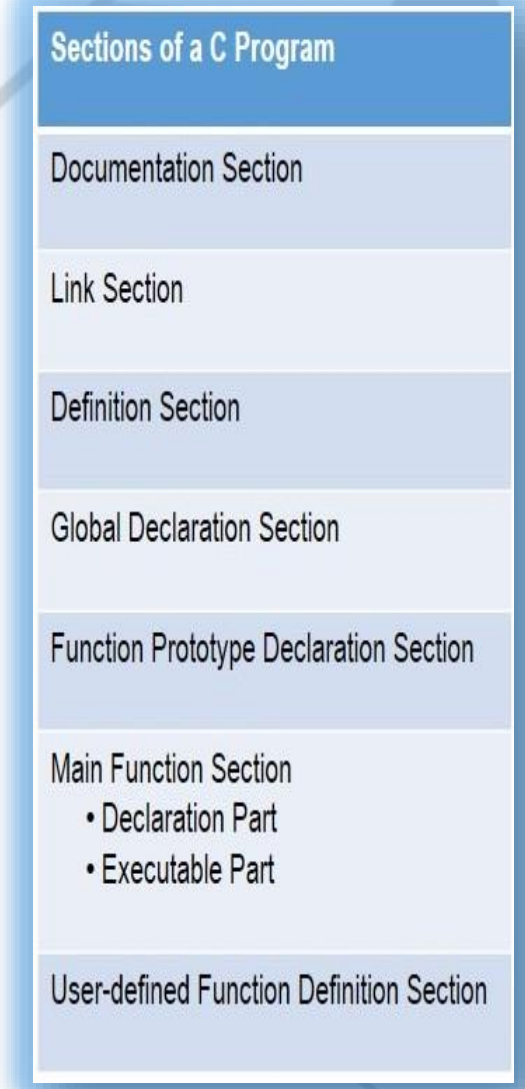
- **Definition Section:**
  - used to describe **all the symbolic constants** (**macros**).
- **Global Declaration Section:**
  - used to define those **variables** that can be **used anywhere** in the **program** and is **used in more than one function**
- **Function Prototype Declaration Section:**
  - used to **declare** all the **user-defined functions**

BASIC STRUCTURE OF C PROGRAM



- **Main Function Section:**
  - **All C programs** must have **a main() function** which contains two parts:
    1. **Declaration part**
    2. **Execution part**
  - These two parts are declared within the opening and closing curly braces of the main().

BASIC STRUCTURE OF C PROGRAM



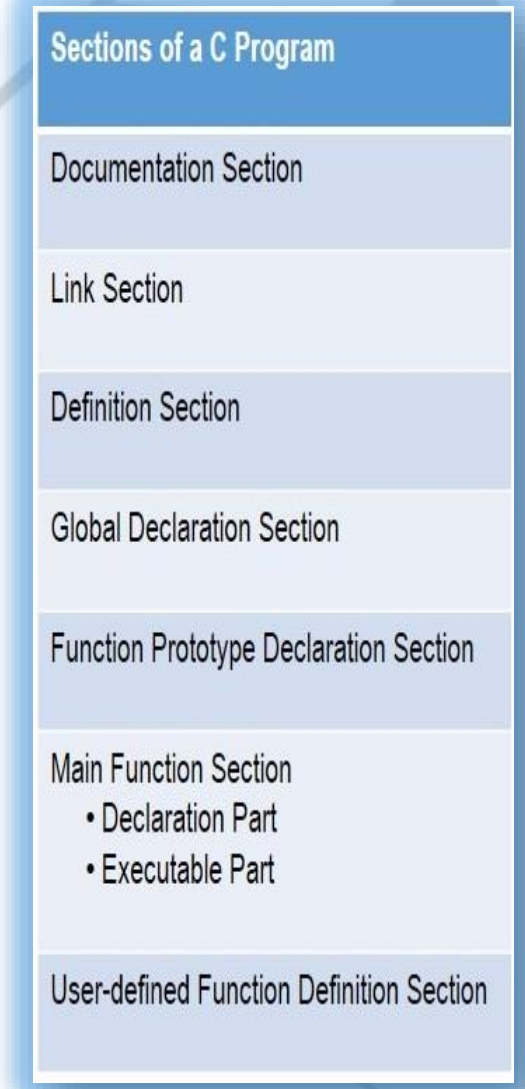
- **Main Function Section:**
  - **Declaration part:**
    - ✓used to declare all the variables that will be used in the executable part (within the program)
  - **Execution part:**
    - ✓must contain at least one statement in it

BASIC STRUCTURE OF C PROGRAM

Sections of a C Program
Documentation Section
Link Section
Definition Section
Global Declaration Section
Function Prototype Declaration Section
Main Function Section • Declaration Part • Executable Part
User-defined Function Definition Section

- **Main Function Section:**
  - **Important Notes:**
    - ✓The execution of the program
    - ✓begins at the opening brace (**{**) and
    - ✓ends with the closing brace (**}**).
    - ✓Each statement in
    - ✓the declaration and execution parts
    - ✓must end with a semi-colon (**;**).

BASIC STRUCTURE OF C PROGRAM



- **User-defined Function Definition Section:**
  - Contains **all** the **user-defined functions** that are **used** to **perform** a **specific task**
  - **User-defined functions** are **called** in the **main()** function

BASIC STRUCTURE OF C PROGRAM: EXAMPLE

Sections of a C Program
Documentation Section
Link Section
Definition Section
Global Declaration Section
Function Prototype Declaration Section
Main Function Section • Declaration Part • Executable Part
User-defined Function Definition Section

```
/*************************************************************************/
/* Description: This is a test program that illustrates the Basic Structure of a C Program*/
/* Author      : Blasminda Catubig Mayol                                  */
/* Date        : September 1, 2019                                        */
/* Place       : University of San Carlos, Cebu City, Philippines         */
/*************************************************************************/

#include<stdio.h>

#define c 10
#define d 20

int m = 22, n = 44;
int a = 50, b = 80;
void test();
```

BASIC STRUCTURE OF C PROGRAM: EXAMPLE

Sections of a C Program
Documentation Section
Link Section
Definition Section
Global Declaration Section
Function Prototype Declaration Section
Main Function Section • Declaration Part • Executable Part
User-defined Function Definition Section

```
int main()
  {
        int x, y, prod;
        x = 5;
        y = 4;
        prod = x * y;

        printf("\nAll variables are accessed from the main function");
        printf("\nValues: m=%d:n=%d:a=%d:b=%d\n", m,n,a,b);

        test();

        printf("\n\nThe values of c = %d; d = %d.", c, d);
        printf("\n\nThe product of x and y = %d.", prod);
        return 0;
  }
```

BASIC STRUCTURE OF C PROGRAM: EXAMPLE

Sections of a C Program
Documentation Section
Link Section
Definition Section
Global Declaration Section
Function Prototype Declaration Section
Main Function Section • Declaration Part • Executable Part
User-defined Function Definition Section

```
void test()
  {
        printf("\n\nAll variables are accessed from the test function");
        printf("\nValues: m=%d:n=%d:a=%d:b=%d", m, n, a, b);
        printf("\nc = %d; d = %d.\n", c, d);
  }
```

STRUCTURE OF C PROGRAM

Sections of a C Program
Documentation Section
Link Section
Definition Section
Global Declaration Section
Function Prototype Declaration Section
Main Function Section • Declaration Part • Executable Part
User-defined Function Definition Section

Return Statement:

- This is the last part of any C program that refers to the returning of the values from a function.
- This return statement and return value depend upon the return-type of the function.

STRUCTURE OF C PROGRAM

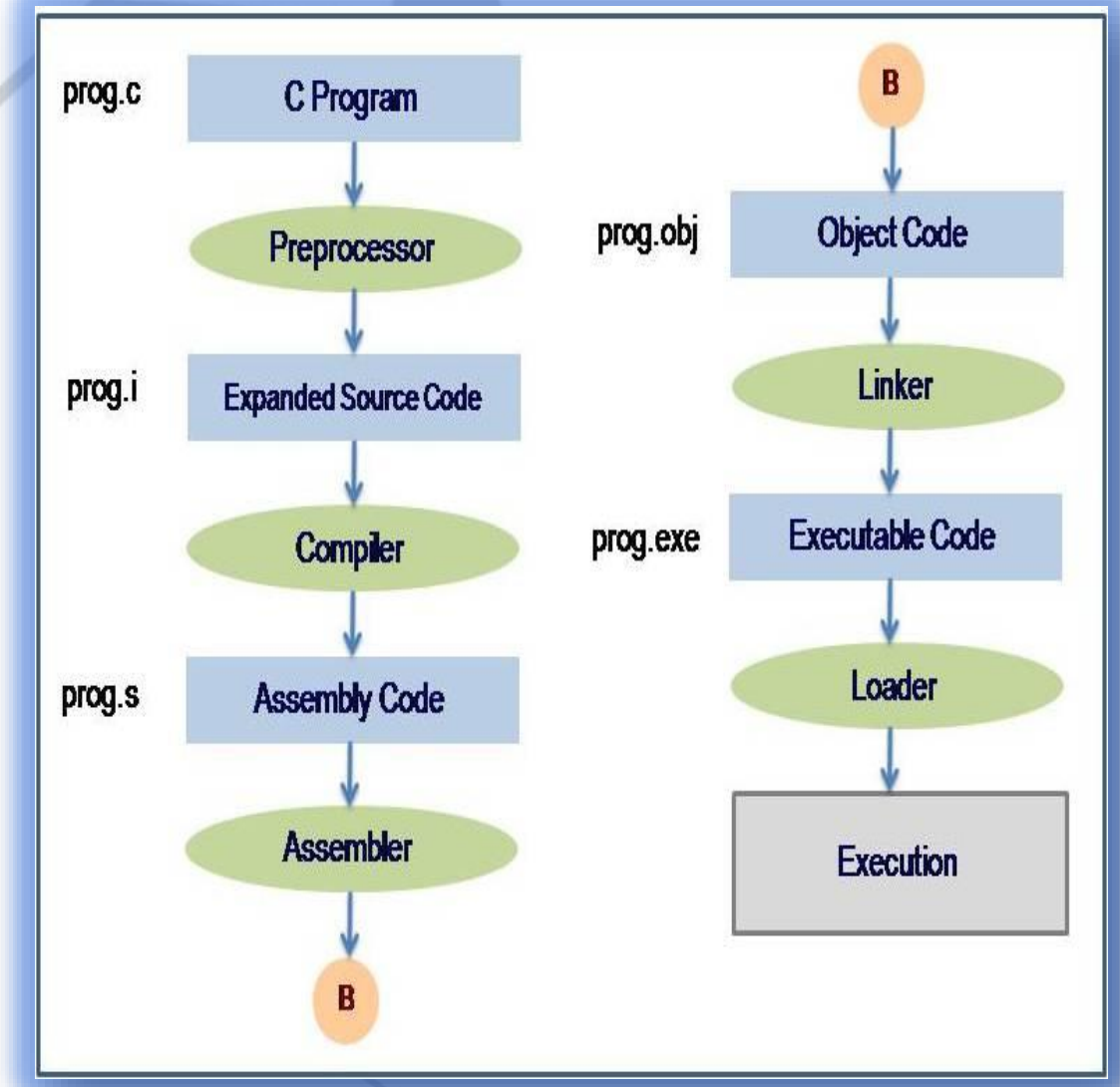
Sections of a C Program
Documentation Section
Link Section
Definition Section
Global Declaration Section
Function Prototype Declaration Section
Main Function Section • Declaration Part • Executable Part
User-defined Function Definition Section

❑ Return Statement:

▪ Important Notes:

- 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.

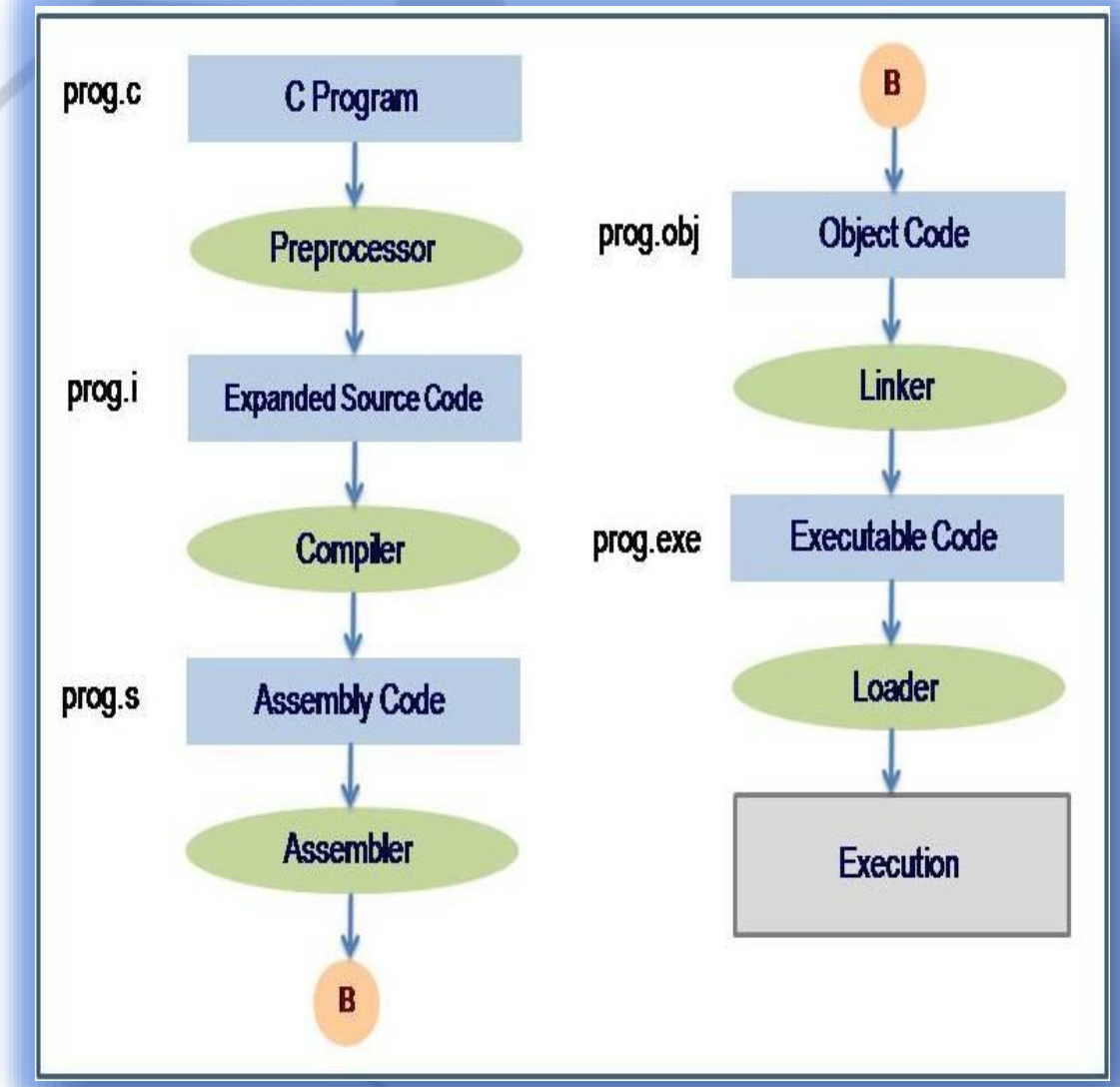
EXECUTION FLOW OF C



#1 FIRST:

- **C program** (source code: **prog.c**) is **sent to preprocessor first**.
- The **preprocessor** is responsible to convert **preprocessor directives** into their respective values.
- The **preprocessor** generates an **expanded source code**.

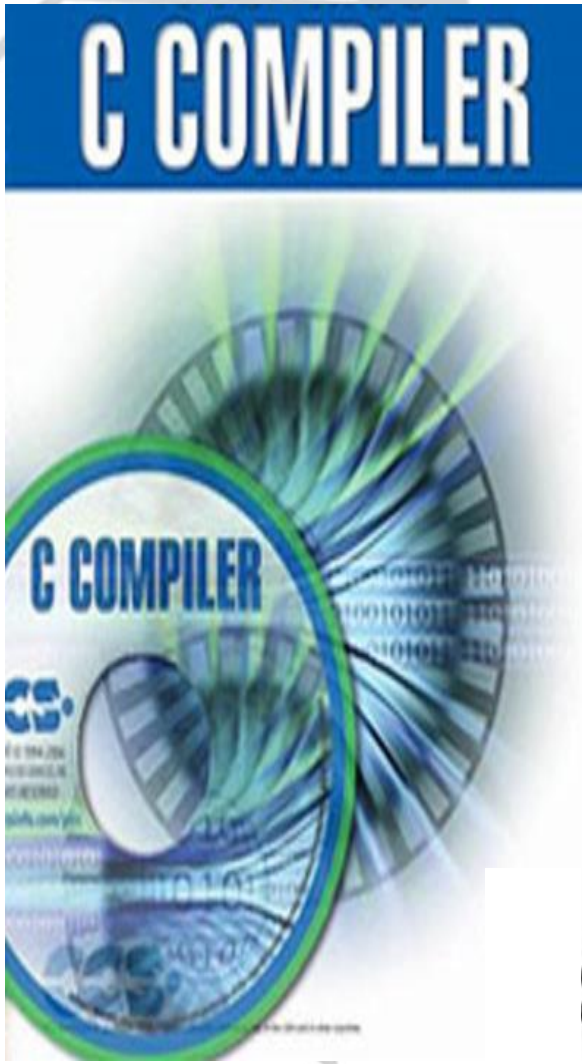
EXECUTION FLOW OF C



#2 SECOND:

- **Expanded source code** (**prog.i**) is
- sent to **compiler**
- which **compiles the code** and
- converts it **into assembly code**.

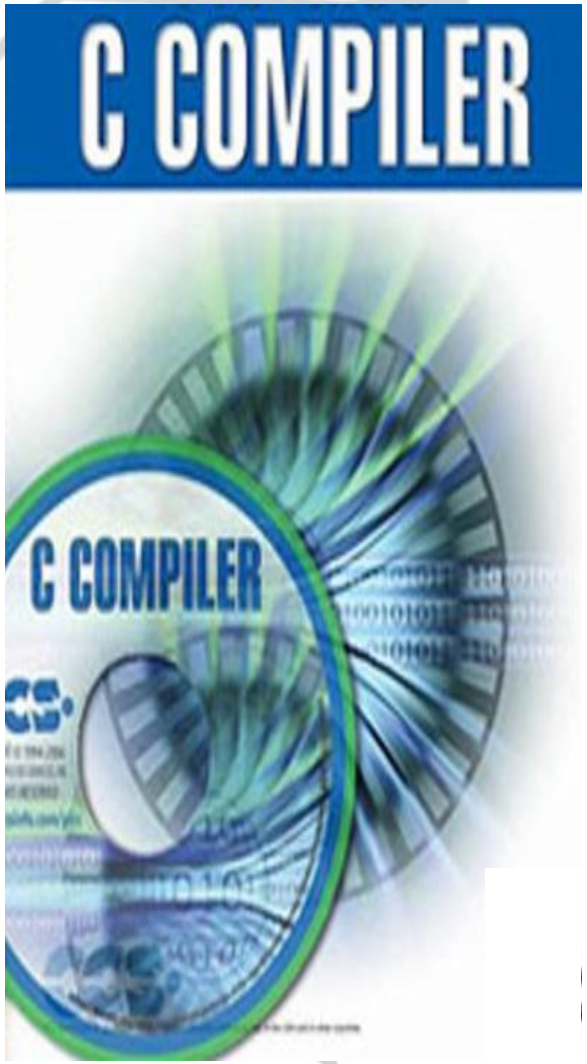
WHAT IS COMPILER?



- A **compiler** is a **computer program**
- that **transforms** human-readable **source code**
- into a **machine code**.

- **In simple terms:**
  - **Compiler takes the code** that a **programmer wrote**
  - and **turned into the binary code**
  - that the **computer can understand**.

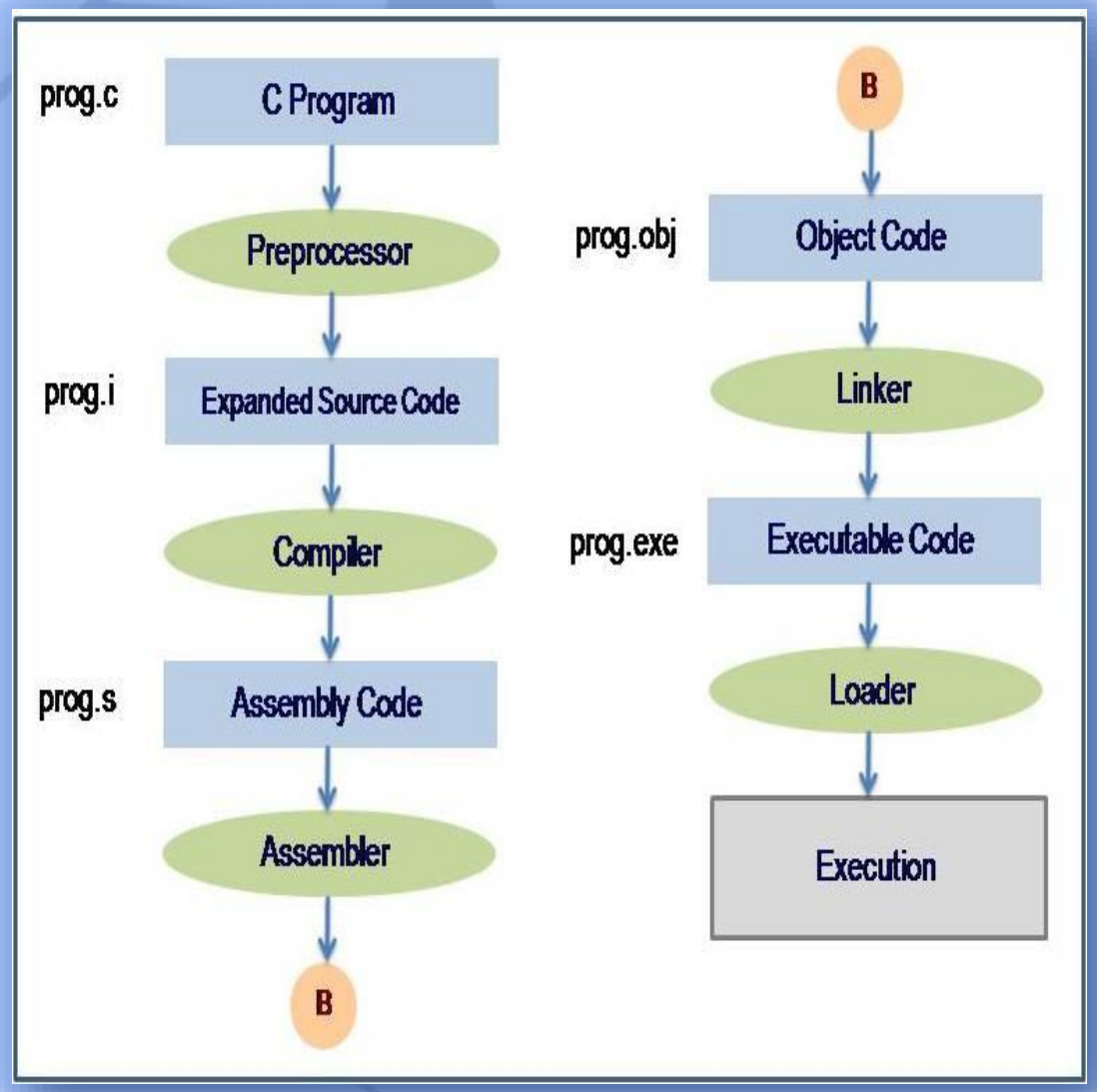
WHAT IS C COMPILER?



- **C compiler** is a software
- that transforms
- the human-readable **C program code**
- to machine-readable code.

- **Compilation:**
  - the process of transforming the code
  - from High-Level Language
  - to Machine Level Language

EXECUTION FLOW OF C

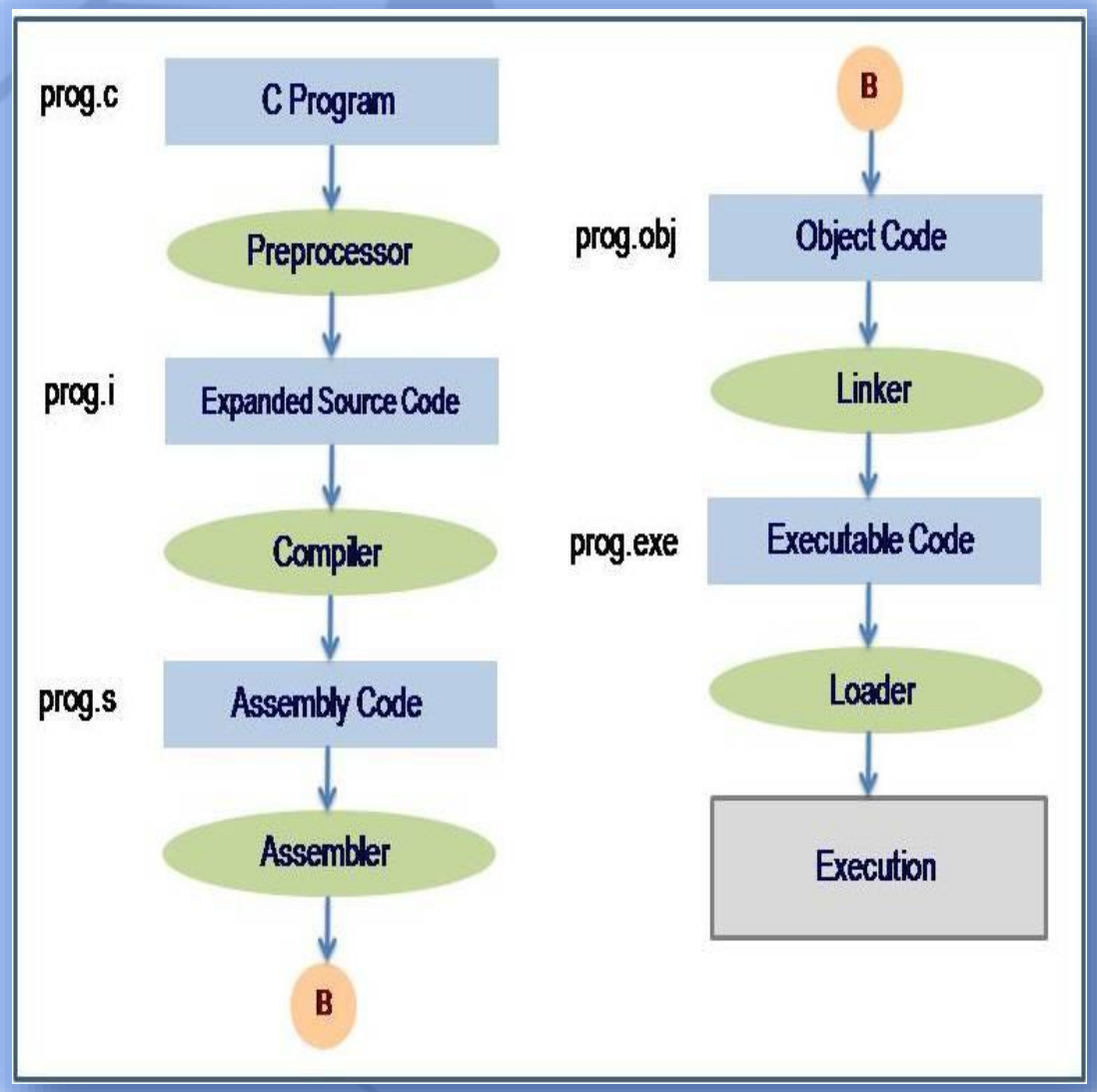


#3 THIRD:

- The **assembly code** (**prog.s**) is
- sent to **assembler**
- which **assembles the code** and
- **converts** it into **object code**.

- Now a **prog.obj** file is generated.

EXECUTION FLOW OF C



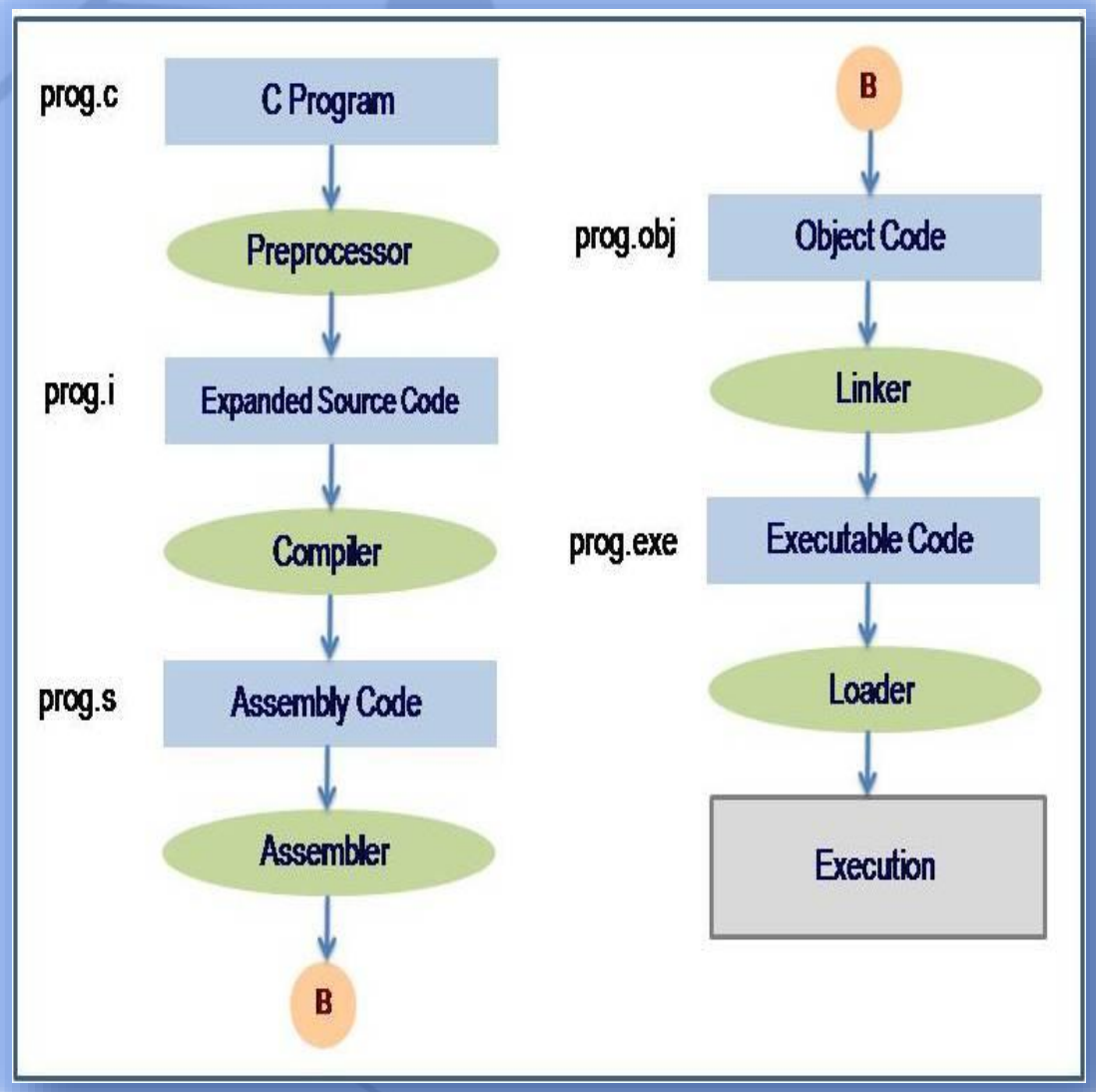
#4 FOURTH:

- The **object code** (**prog.obj**) is
- sent to **linker**
- which **links it to the library**
- such as **header files**.

- Then the **object code** is
- **converted** into **executable code**.

- A **prog.exe** file is generated.

EXECUTION FLOW OF C



#5 FIFTH:

- The **executable code** (**prog.exe**) is
- sent to **loader**
- which **loads it into memory** and
- then **it is executed**.

- **After execution**,
- **output** is
- **sent to console (monitor/screen)**.