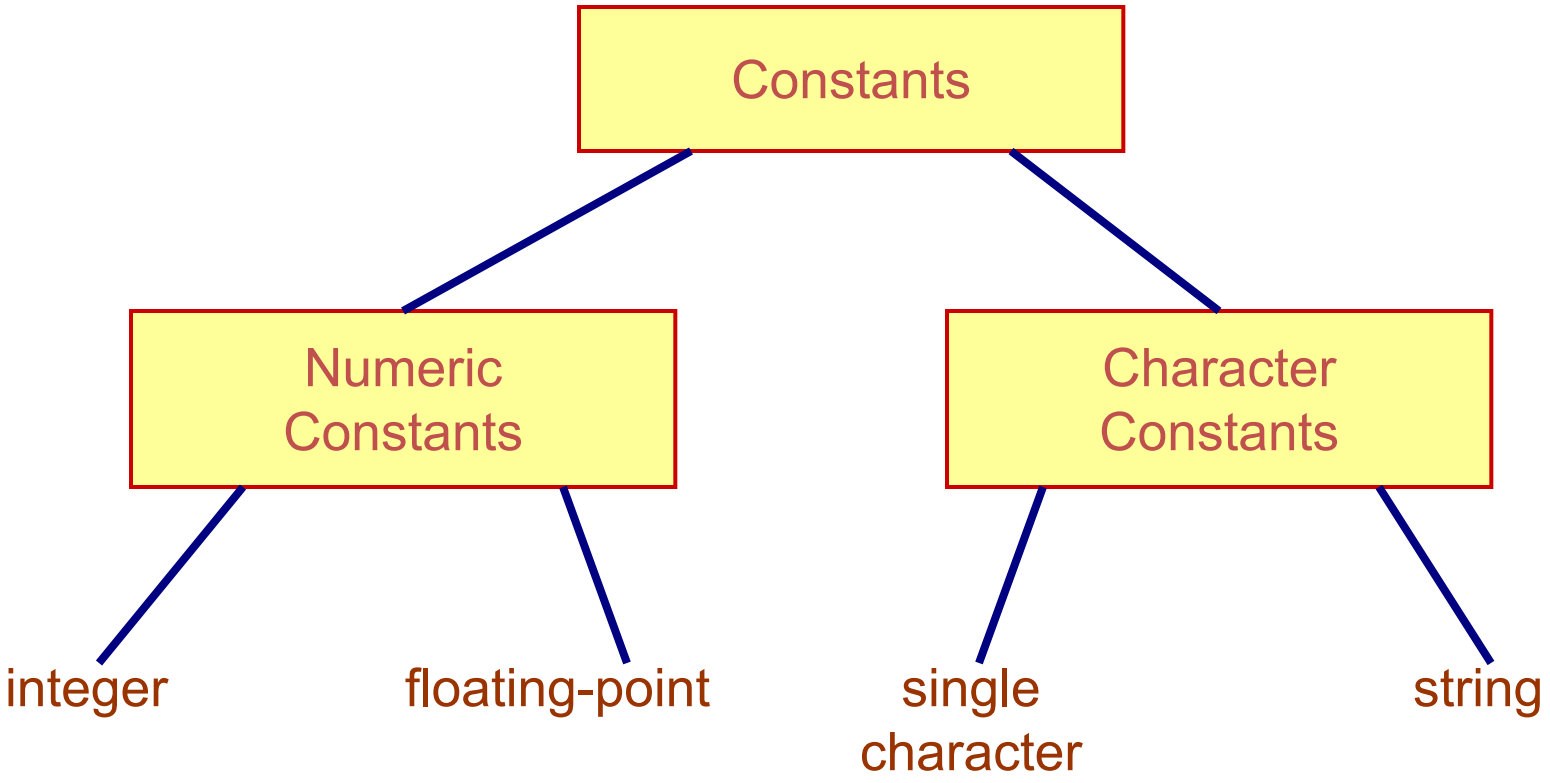
Structured Programming
CSE 103

Professor Dr. Mohammad Abu Yousuf

Constants


Constants
Numeric Constants
Character Constants
integer
floating-point
single character
string

Integer Constants

- Consists of a sequence of digits, with possibly a plus or a minus sign before it.
  - Embedded spaces, commas and non-digit characters are not permitted between digits.

- Maximum and minimum values (for 32-bit representations)

  Maximum :: 2147483647

  Minimum :: – 2147483648

Floating-point Constants

- Can contain fractional parts.

- Very large or very small numbers can be represented.

  23000000 can be represented as 2.3e7

- Two different notations:

  1. Decimal notation

     25.0, 0.0034, .84, -2.234

  2. Exponential (scientific) notation

     3.45e23, 0.123e-12, 123E2

e means "10 to the power of"

Single Character Constants

- Contains a single character enclosed within a pair of single quote marks.
  - Examples :: ‘2’, ‘+’, ‘Z’
- Some special backslash characters

  ‘\n’ new line

  ‘\t’ horizontal tab

  ‘\’’ single quote

  ‘\”’ double quote

  ‘\\’ backslash

  ‘\0’ null

String Constants

- Sequence of characters enclosed in double quotes.
  - The characters may be letters, numbers, special characters and blank spaces.

- Examples:

  “nice”, “Good Morning”, “3+6”, “3”, “C”

Escape Sequences

- There are certain characters in C when they are preceded by a backslash they will have special meaning.

Sequence	Meaning
\a	Bell (alert)
\b	Backspace
\n	Newline
\t	Horizontal tab
\\	Backslash
\'	Single quote
\"	Double quotation
\x*hh*	ASCII char specified by hex digits *hh*
ooo	ASCII char specified by octal digits *ooo*

Example: Constant (The const keyword)

Syntax: const *type variable* = value;

```
1.#include<stdio.h>
2.int main(){
3.    const float PI=3.14;
4.    printf("The value of PI is: %f",PI);
5.    return 0;
6.}
```

Output:

The value of PI is: 3.140000

Example: Constant (The const keyword)

- If you try to change the value of PI, it will render compile time error.

```
1.#include<stdio.h>
2.int main(){
3.    const float PI=3.14;
4.    PI=4.5;
5.    printf("The value of PI is: %f",PI);
6.    return 0;
7.}
```

Output:
Compile Time Error: Cannot modify a const object

Declaration of Variables

- There are two purposes:
  1. It tells the compiler what the variable name is.
  2. It specifies what type of data the variable will hold.

- Syntax:

  data-type variable_1,
  variable_2,……..,variable_n;

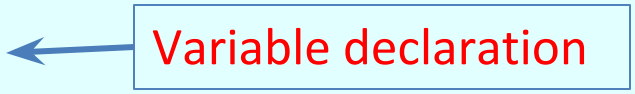
- Examples:

```
int   velocity, distance;
int   a, b, c, d;
float  temp;
char  flag, option;
```

An Example : Variable Declaration

```
#include <stdio.h>
main()
   {
      float  speed, time, distance;

      scanf ("%f %f", &speed, &time);
      distance = speed * time;
      printf ("\n The distance traversed is: %f", distance);
   }
```



Expression

- An *expression represents a single data item, such as a number or a character. The expression may consist of a* single entity, such as a constant, a variable, an array element or a reference to a function. It may also consist of some combination of such entities, interconnected by one or more *operators.*
- Several simple expressions are shown below.

a + b

x = y

c = a + b

x <= y

x == Y

++i

Statement

- A *statement causes the computer to carry out some action. There are three different classes of statements in C.*
- They are *expression statements, compound statements and control statements*.

Expression Statement:

- An expression statement consists of an expression followed by a semicolon.

```
a = 3;          // Assignment statement
c = a + b ;     // Assignment statement
++i;            // Incremental statement
p r i n t f ("Area = %f”, area);
```

Statement (Cont..)

Compound Statement:

- A compound statement consists of several individual statements enclosed within a pair of braces { }.
- A typical compound statement is shown below.

```
{
 p i = 3.141593;
 circumference = 2 * pi * radius;
 area = pi * radius * radius;
}
```

Statement (Cont..)

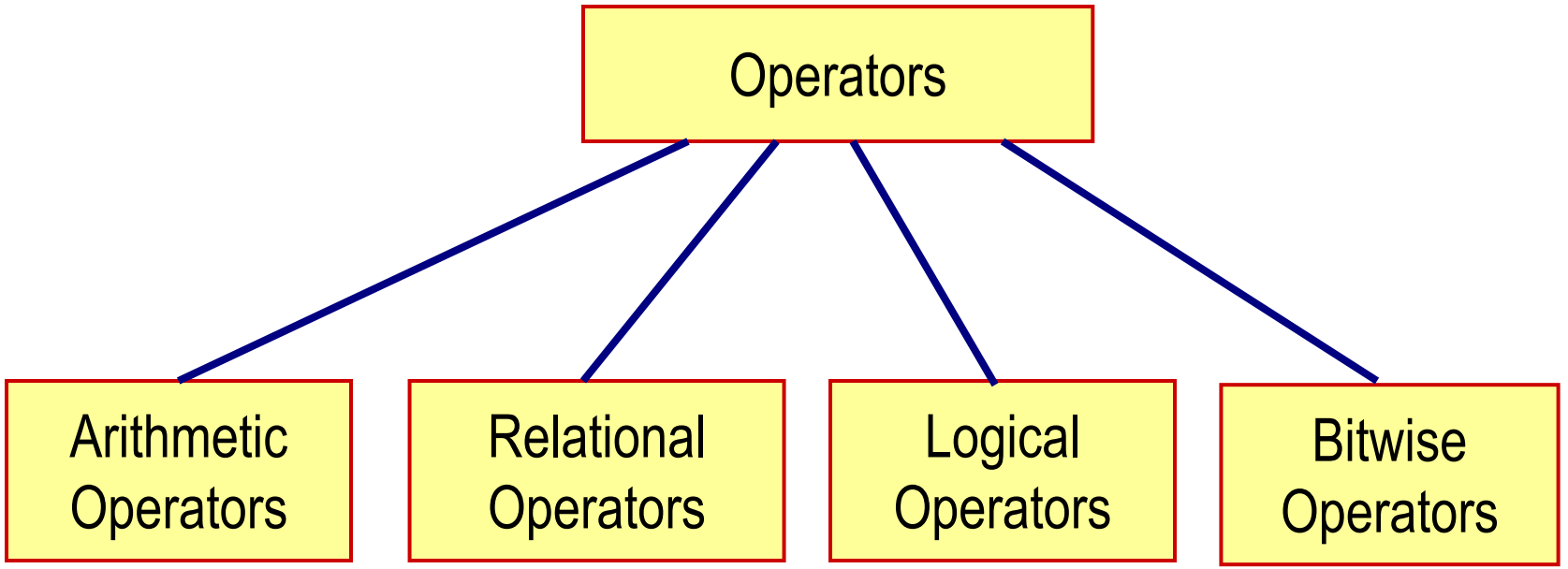
Control Statement:

- Control statements are used to create special program features, such as logical tests, loops and branches.

  Example:

```
while (count <= n) {
        printf ( ' x = ' ) ;
        scanf ( “%f" , &x) ;
        sum += x;
        ++count;
          }
```

Operators


Operators
Arithmetic Operators
Relational Operators
Logical Operators
Bitwise Operators

Arithmetic Operators

- Addition :: +
- Subtraction :: –
- Division :: /
- Multiplication :: *
- Modulus :: %

Examples:

```
distance = rate * time ;
netIncome = income - tax ;
speed = distance / time ;
area = PI * radius * radius;
y = a * x * x + b*x + c;
quotient = dividend / divisor;
remain =dividend % divisor;
```

Arithmetic Operators(Cont..)

- Suppose x and y are two integer variables, whose values are 13 and 5 respectively.

x + y	18
x – y	8
x * y	65
x / y	2
x % y	3

Operator Precedence

- In decreasing order of priority
  1. Parentheses :: ( )
  2. Unary minus :: –5
  3. Multiplication, Division, and Modulus
  4. Addition and Subtraction

- For operators of the *same priority*, evaluation is from *left to right* as they appear.

- Parenthesis may be used to change the precedence of operator evaluation.

Examples: Arithmetic expressions

a + b * c – d / e ⇒ a + (b * c) – (d / e)

a * – b + d % e – f ⇒ a * (– b) + (d % e) – f

a – b + c + d ⇒ (((a – b) + c) + d)

x * y * z ⇒ ((x * y) * z)

a + b + c * d * e ⇒ (a + b) + ((c * d) * e)

Type Cast

- The value of an expression can be converted to a different data type if desired.
- To do so, the expression must be preceded by the name of the desired data type, enclosed in parentheses, i.e.,

  (data type) expression.

Example:

```
int x=10;
float y,z=3.14;
y=(float) x;      /* y=10.0 */
x=(int) z;        /* x=3    */
x=(int) (-z);     /* x=-3   -- rounded approaching zero */
```

Relational Operators

- Used to compare two quantities.

<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Relational Operators: Examples

10 > 20 is false

25 < 35.5 is true

12 > (7 + 5) is false

- When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared.

a + b > c – d is the same as (a+b) > (c+d)

Relational Operators: Examples

- Sample code segment in C

```
if  (x > y)
   printf (“%d is larger\n”, x);
else
  printf (“%d is larger\n”, y);
```

Logical Operators

- There are two logical operators in C (also called logical connectives).

  && □ Logical AND

  || □ Logical OR

- What they do?
  - They act upon operands that are themselves logical expressions.
  - The individual logical expressions get combined into more complex conditions that are true or false.

Logical Operators (Cont..)

- Logical AND
  - Result is true if both the operands are true.
- Logical OR
  - Result is true if at least one of the operands are true.

X	Y	X && Y	X || Y
FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE

Bitwise Operators

- The bitwise operators are the operators used to perform the operations on the data at the bit-level.

Operator	Meaning of operator
&	Bitwise AND operator
|	Bitwise OR operator
^	Bitwise exclusive OR operator
~	One's complement operator (unary operator)
<<	Left shift operator
>>	Right shift operator

Bitwise Operators

- Truth table of bitwise operators

X	Y	X&Y	X|Y	X^Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise AND

For example:

We have two variables a and b.

a =6;

b=4;

The binary representation of the above two variables are given below :

a = 0110

b = 0100

---

a&b = 0100

```
1.#include <stdio.h>
2.int main()
3.{
4.   int a=6, b=4;  // variable
5.   printf("The output is %d",a&b);
6.   return 0;
7.}
```

Output: The output is 4

Bitwise AND

Bitwise AND is a binary operator (operates on two operands). It's denoted by &.
The & operator compares corresponding bits of two operands. If both bits are 1, it gives 1. If either of the bits is not 1, it gives 0.

For example,

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bit Operation of 12 and 25

  00001100
& 00011001

  ________
  00001000 = 8 (In decimal)
```

Bitwise OR

For example:
We consider two variables,
a = 23;
b = 10;

The binary representation of the above two variables would be:
a = 00010111
b = 00001010

_____
a|b = 00011111

```
1.#include <stdio.h>
2.int main()
3.{
4.   int a=23,b=10;  // variable
5.   printf("The output is %d",a|b);
6.   return 0;
7.}
```

Output: The output is 31

Bitwise OR

Bitwise OR is a binary operator (operates on two operands). It's denoted by |.

The | operator compares corresponding bits of two operands. If either of the bits is 1, it gives 1. If not, it gives 0. For example,

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
  00001100
| 00011001
  ___________
  00011101 = 29 (In decimal)
```

Bitwise XOR

For example:
We consider two variables a and b,
a = 12;
b = 10;
The binary representation of the above two variables would be:
a = 00001100
b = 00001010

(a^b) = 0000 1110

```
1.#include <stdio.h>
2.int main()
3.{
4.   int a=12,b=10;  // variable
5.   printf("The output is %d",a^b);
6.   return 0;
7.}
```

Output: The output is 6

Bitwise XOR

Bitwise XOR is a binary operator (operates on two operands). It's denoted by ^.

The ^ operator compares corresponding bits of two operands. If corresponding bits are different, it gives 1. If corresponding bits are same, it gives 0.

For example,

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bitwise XOR Operation of 12 and 25
  00001100
^ 00011001
  ----------
  00010101 = 21 (In decimal)
```

Bitwise Complement

Bitwise complement is an unary operator (works on only one operand). It is denoted by ~.

The ~ operator inverts the bit pattern. It makes every 0 to 1, and every 1 to 0.

```
35 = 00100011 (In Binary)
Bitwise complement Operation of 35

~ 00100011
  ___________
  11011100 = 220 (In decimal)
```

Bitwise Complement

```
1.#include <stdio.h>
2.int main()
3.{
4.   int a= 35;  // variable
5.   printf("The output is %d",~a);
6.   return 0;
7.}
```

Output: The output is -36

Why are we getting output -36 instead of 220?
It's because the compiler is showing 2's complement of that number; negative notation of the binary number.

Bitwise Complement

For any integer n, 2's complement of n will be -(n+1).

Decimal	Binary	2's complement
0	00000000	-(11111111+1) = -00000000 = -0(decimal)
1	00000001	-(11111110+1) = -11111111 = -256(decimal)
12	00001100	-(11110011+1) = -11110100 = -244(decimal)
220	11011100	-(00100011+1) = -00100100 = -36(decimal)

Note: Overflow is ignored while computing 2's complement.

The bitwise complement of 35 is 220 (in decimal). The 2's complement of 220 is -36. Hence, the output is -36 instead of 220.

left shift, right shift

A= 00111100

<< (left shift)

- Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand

  Example: A << 2 will give 240 which is 1111 0000.

>> (right shift)

- Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

  Example: A >> 2 will give 15 which is 1111

left shift, right shift

```
1.#include <stdio.h>
2.int main()
3.{
4.    int a=5; // variable declaration
5.    printf("The value of a<<2 is : %d ", a<<2);
6.    return 0;
7.}
```

```
1.#include <stdio.h>
2.int main()
3.{
4.    char b = ‘A’; // variable declaration
5.    printf("The value of A<<2 is : %d ", A<<2);
6.    return 0;
7.}
```

Other Operators

- Some unary operators: -, ++, --, !

  Example : ++a, --b
- C contains the following five additional assignment operators: +=, -= , *=, /= **and %=.**
- **To see how** they are used, consider the first operator, +=. The assignment expression

  expression 1 += expression 2

  is equivalent to

  expression 1 = expression 1 + expression 2
- Similarly, the assignment expression

  expression I -= expression 2

  is equivalent to

  expression 1 = expression I - expression 2

  and so on for all five operators.

Other Operators

- Suppose that i and j are integer variables whose values are *5 and 7, and f and g are floating-point* variables whose values are *5.5 and -3.25.*

Expression	*Equivalent Expression*	*Final value*
`i += 5`	`i = i + 5`	10
`f -= g`	`f = f - g`	8.75
`j *= (i - 3)`	`j = j * (i - 3)`	14
`f /= 3`	`f = f / 3`	1.833333
`i %= (j - 2)`	`i = i % (j - 2)`	0

Increment and Decrement Operators

```
// Working of increment and decrement operators
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;
    printf("++a = %d \n", ++a);
    printf("--b = %d \n", --b);
    printf("++c = %f \n", ++c);
    printf("--d = %f \n", --d);
    return 0;
}
```

```
Output:
++a = 11
--b = 99
++c = 11.500000
--d = 99.500000
```

Operator Precedence Groups

Operator category	*Operators*	*Associativity*
unary operators	- ++ -- ! sizeof (*type*)	R → L
arithmetic multiply, divide and remainder	* / %	L → R
arithmetic add and subtract	+ -	L → R
relational operators	< <= > >=	L → R
equality operators	== !=	L → R
logical *and*	&&	L→ R
logical *or*	||	L → R
conditional operator	? :	R → L
assignment operators	= += -= *= /= %=	R → L

Operator Precedence in C

```c
#include <stdio.h>
main()
{ int a = 20;
  int b = 10;
  int c = 15;
  int d = 5;
  int e;
  e = (a + b) * c / d; // ( 30 * 15 ) / 5
  printf("Value of (a + b) * c / d is : %d\n", e );

  e = ((a + b) * c) / d; // (30 * 15 ) / 5
  printf("Value of ((a + b) * c) / d is : %d\n" , e );

  e = (a + b) * (c / d); // (30) * (15/5)
  printf("Value of (a + b) * (c / d) is : %d\n", e );

  e = a + (b * c) / d; // 20 + (150/5)
  printf("Value of a + (b * c) / d is : %d\n" , e );
  return 0;
}
```

Operator Precedence in C

Output:

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is : 90
Value of (a + b) * (c / d) is : 90
Value of a + (b * c) / d is : 50
```

Library function

- The C language is accompanied by a number of library functions that carry out various commonly used operations or calculations.
- A library function is accessed simply by writing the function name, followed by a list of arguments that represent information being passed to the function.

Library function (Cont..)

Some commonly used library function

Function	*Type*	*Purpose*
`abs(i)`	int	Return the absolute value of `i`.
`ceil(d)`	double	Round up to the next integer value (the smallest integer that is greater than or equal to `d`).
`cos(d)`	double	Return the cosine of `d`.
`cosh(d)`	double	Return the hyperbolic cosine of `d`.
`exp(d)`	double	Raise e to the power `d` ($e = 2.7182818 \cdots$ is the base of the natural (Naperian) system of logarithms).
`fabs(d)`	double	Return the absolute value of `d`.
`floor(d)`	double	Round down to the next integer value (the largest integer that does not exceed `d`).
`fmod(d1,d2)`	double	Return the remainder (i.e., the noninteger part of the quotient) of `d1/d2`, with same sign as `d1`.
`getchar()`	int	Enter a character from the standard input device.
`log(d)`	double	Return the natural logarithm of `d`.
`pow(d1,d2)`	double	Return `d1` raised to the `d2` power.
`printf(...)`	int	Send data items to the standard output device (arguments are complicated — see Chap. 4).
`putchar(c)`	int	Send a character to the standard output device.
`rand()`	int	Return a random positive integer.
`sin(d)`	double	Return the sine of `d`.

Library function (Cont..)

`sqrt(d)`	double	Return the square root of `d`.
`srand(u)`	void	Initialize the random number generator.
`scanf(...)`	int	Enter data items from the standard input device (arguments are complicated – see Chap. 4).
`tan(d)`	double	Return the tangent of `d`.
`toascii(c)`	int	Convert value of argument to ASCII.
`tolower(c)`	int	Convert letter to lowercase.
`toupper(c)`	int	Convert letter to uppercase.

- suppose that c1 and c2 are character-type variables that represent the characters **P** and **T,** respectively. Several arithmetic expressions that make use of these variables are shown below, together with their resulting values (based upon the ASCII character set).

Expression	*Value*
c1	80
c1 + c2	164
c1 + c2 + 5	169
c1 + c2 + '5'	217

- **Note that P is encoded as (decimal) 80, T is encoded as 84, and 5 is encoded as 53 in the ASCII character set**

Comments in C

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

1. Single Line Comments
2. Multi-Line Comments

- Single Line Comments

Single line comments are represented by double slash //.

```
1.#include<stdio.h>
2.int main(){
3.    //printing information
4.    printf("Hello C");
5.    return 0;
6.}
```

Comments in C

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

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

```
1.#include<stdio.h>
2.int main(){
3.    /*printing information
4.      Multi-Line Comment*/
5.    printf("Hello C");
6.    return 0;
7.}
```

Thank You