F.Y.B.Sc.IT-SEM 1

Programming Principles With C (USIT101)

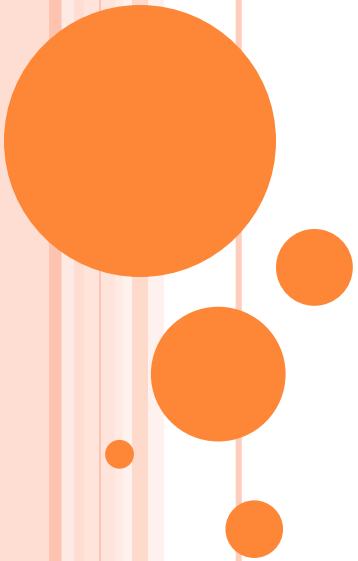
By,
Nikita Madwal

UNIT 2

2.Types of Operators

3.Control Flow

2. Types of Operators



Operators

- The symbols which are used to perform different types of logical and mathematical operations in a C program are called C **operators**
- The data items that **operators** (+,-,*,etc.) act upon are called **operands** (values).
- E.g. a + b
- Here **a** and **b** are ***Operands*** and **+** is ***Operator***

Arithmetic operators

- Arithmetic operators are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus in C programs
- The % operator is referred to as the modulus operator

There are five *arithmetic operators* in C. They are

Operator	*Purpose*
+	addition
-	subtraction
*	multiplication
/	division
%	remainder after integer division

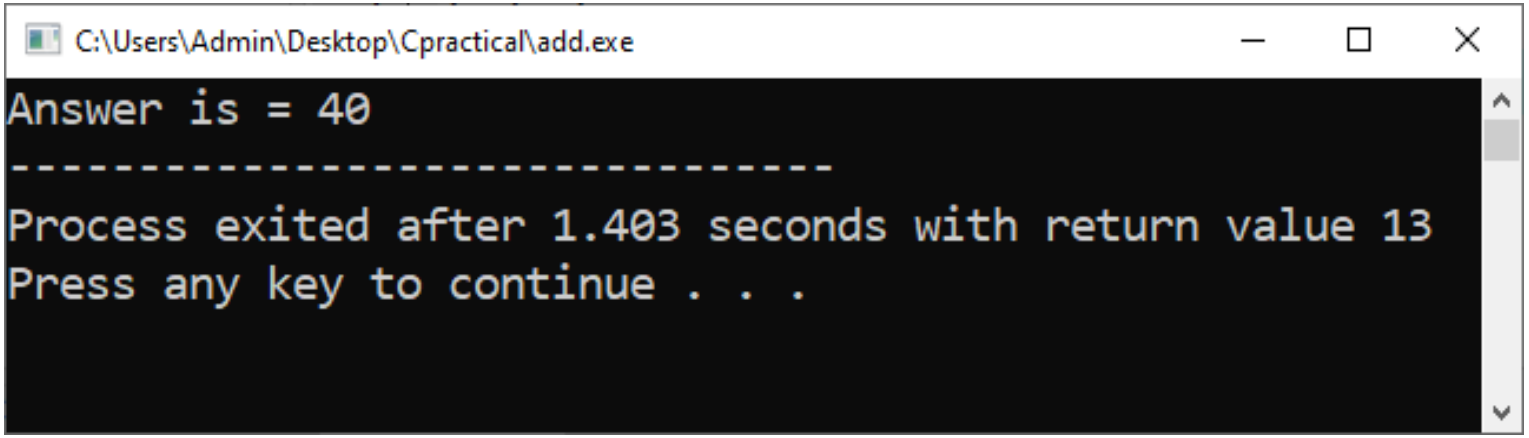
EXAMPLE 3.1 Suppose that a and b are integer variables whose values are 10 and 3, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	*Value*
a + b	13
a - b	7
a * b	30
a / b	3
a % b	1

add.c

```c
#include<stdio.h>
#include<conio.h>
void main()
{
int x=20,y=20,total;

total=x+y;

printf("Answer is = %d",total);
getch();
}
```

Output :



Relational and logical operators

- Relational operators are used to find the relation between two variables. i.e. to compare the values of two variables in a C program
- If the relation is **true**, it ***returns value*** **1**; if the relation is **false**, it ***returns value*** **0**

There are four *relational operators* in C. They are

Operator	*Meaning*
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

- Closely associated with the relational operators are the following two **equality operators**

Operator	*Meaning*
==	equal to
!=	not equal to

- E.g.

Operator	Meaning of Operator	Example
==	Equal to	`5 == 3` is evaluated to 0
>	Greater than	`5 > 3` is evaluated to 1
<	Less than	`5 < 3` is evaluated to 0
!=	Not equal to	`5 != 3` is evaluated to 1
>=	Greater than or equal to	`5 >= 3` is evaluated to 1
<=	Less than or equal to	`5 <= 3` is evaluated to 0

- E.g.

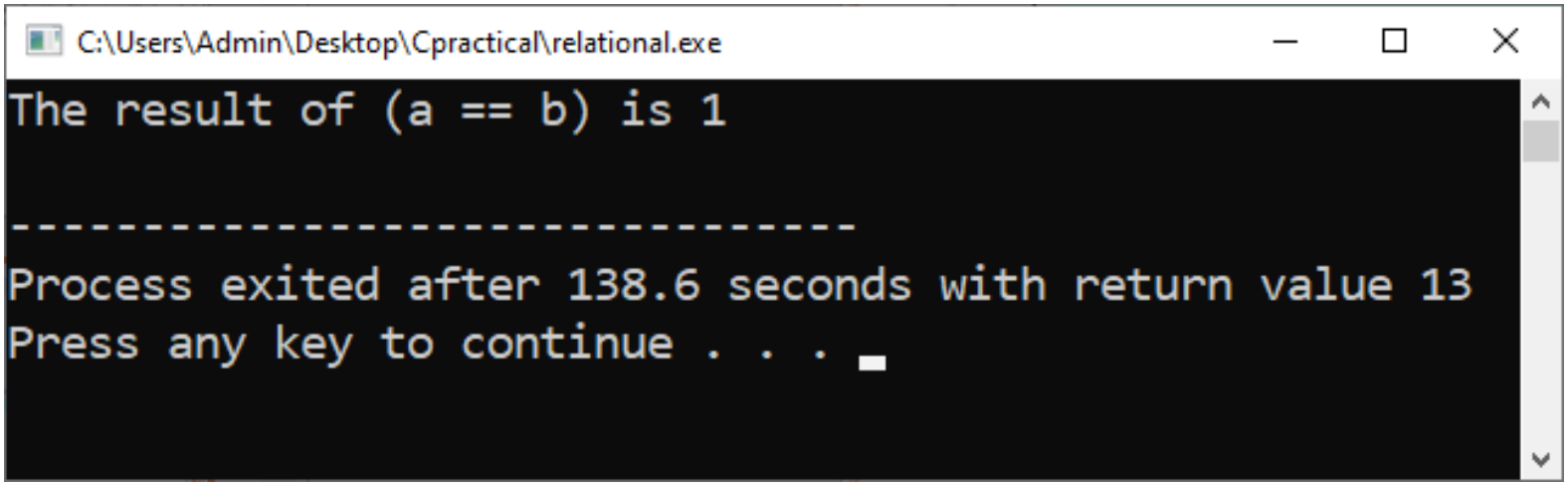
EXAMPLE 3.15 Suppose that i, j and k are integer variables whose values are 1, 2 and 3, respectively. Several logical expressions involving these variables are shown below.

Expression	*Interpretation*	*Value*
`i < j`	true	1
`(i + j) >= k`	true	1
`(j + k) > (i + 5)`	false	0
`k != 3`	false	0
`j == 2`	true	1

relational.c

```c
#include<stdio.h>
#include<conio.h>
void main()
{
    int a = 5, b = 5,result;

    result = (a == b);
    printf("The result of (a == b) is %d \n", result);

    getch();
}
```

Output :



LOGICAL OPERATORS

- C contains two logical operators (also called logical connectives)
- The net effect is to combine the individual logical expressions into more complex conditions that are either true or false.
- The result of a *logical and* operation will be **true** only **if both operands are true**
- the result of a *logical or* operation will be **true** if **either operand is true or if both operands are true**. In other words, the result of a *logical or* operation will be **false** only if **both operands are false**.

Operator	*Meaning*
&&	and
||	or

These operators are referred to as *logical and* and *logical or*, respectively.

Operators	Example/Description
&& (logical AND)	(x>5)&&(y<5) It returns true when both conditions are true
|| (logical OR)	(x>=10)||(y>=10) It returns true when at-least one of the condition is true
! (logical NOT)	!((x>5)&&(y<5)) It reverses the state of the operand "((x>5) && (y<5))" If "((x>5) && (y<5))" is true, logical NOT operator makes it false

EXAMPLE 3.17 Suppose that `i` is an integer variable whose value is 7, `f` is a floating-point variable whose value is 5.5, and `c` is a character variable that represents the character `'w'`. Several complex logical expressions that make use of these variables are shown below.

Expression	*Interpretation*	*Value*
`(i >= 6) && (c == 'w')`	true	1
`(i >= 6) || (c == 119)`	true	1
`(f < 11) && (i > 100)`	false	0
`(c != 'p') || ((i + f) <= 10)`	true	1

The first expression is true because both operands are true. In the second expression, both operands are again true; hence the overall expression is true. The third expression is false because the second operand is false. And finally, the fourth expression is true because the first operand is true.

- C also includes the unary operator (**!**)that negates the value of a logical expression; i.e., it causes an expression that is **originally true to become false**, **and vice versa**.
- This operator is referred to as the logical negation (or logical not) operator.

EXAMPLE 3.18 Suppose that `i` is an integer variable whose value is 7, and `f` is a floating-point variable whose value is 5.5. Several logical expressions which make use of these variables and the logical negation operator are shown below.

Expression	*Interpretation*	*Value*
`f > 5`	true	1
`!(f > 5)`	false	0
`i <= 3`	false	0
`!(i <= 3)`	true	1
`i > (f + 1)`	true	1
`!(i > (f + 1))`	false	0

Assignment operators

- Assignment expressions, which assign the value of an expression to an identifier
- The most commonly used assignment operator is =

identifier = *expression*

where *identifier* generally represents a variable, and *expression* represents a constant, a variable or a more complex expression.

EXAMPLE 3.21 Here are some typical assignment expressions that make use of the = operator.

```
a = 3
x = y
delta = 0.001
sum = a + b
area = length * width
```

Multiple assignments of the form

identifier 1 = *identifier 2* = ··· = *expression*

EXAMPLE 3.23 Suppose that `i` and `j` are integer variables. The multiple assignment expression

```
i = j = 5
```

will cause the integer value 5 to be assigned to both `i` and `j`. (To be more precise, 5 is first assigned to `j`, and the value of `j` is then assigned to `i`.)

- C contains the following five additional assignment operators: +=, -= , *=, /= and %=

EXAMPLE 3.24 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. Several assignment expressions that make use of these variables are shown below. Each expression utilizes the *original* values of `i`, `j`, `f` and `g`.

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

UNARY OPERATOR

- C includes a class of operators that act upon a **single operand** to produce a new value. Such operators are known as **unary operators**
- Perhaps the most common unary operation is **unary minus**, where a numerical constant, variable or expression is **preceded by a minus sign.**

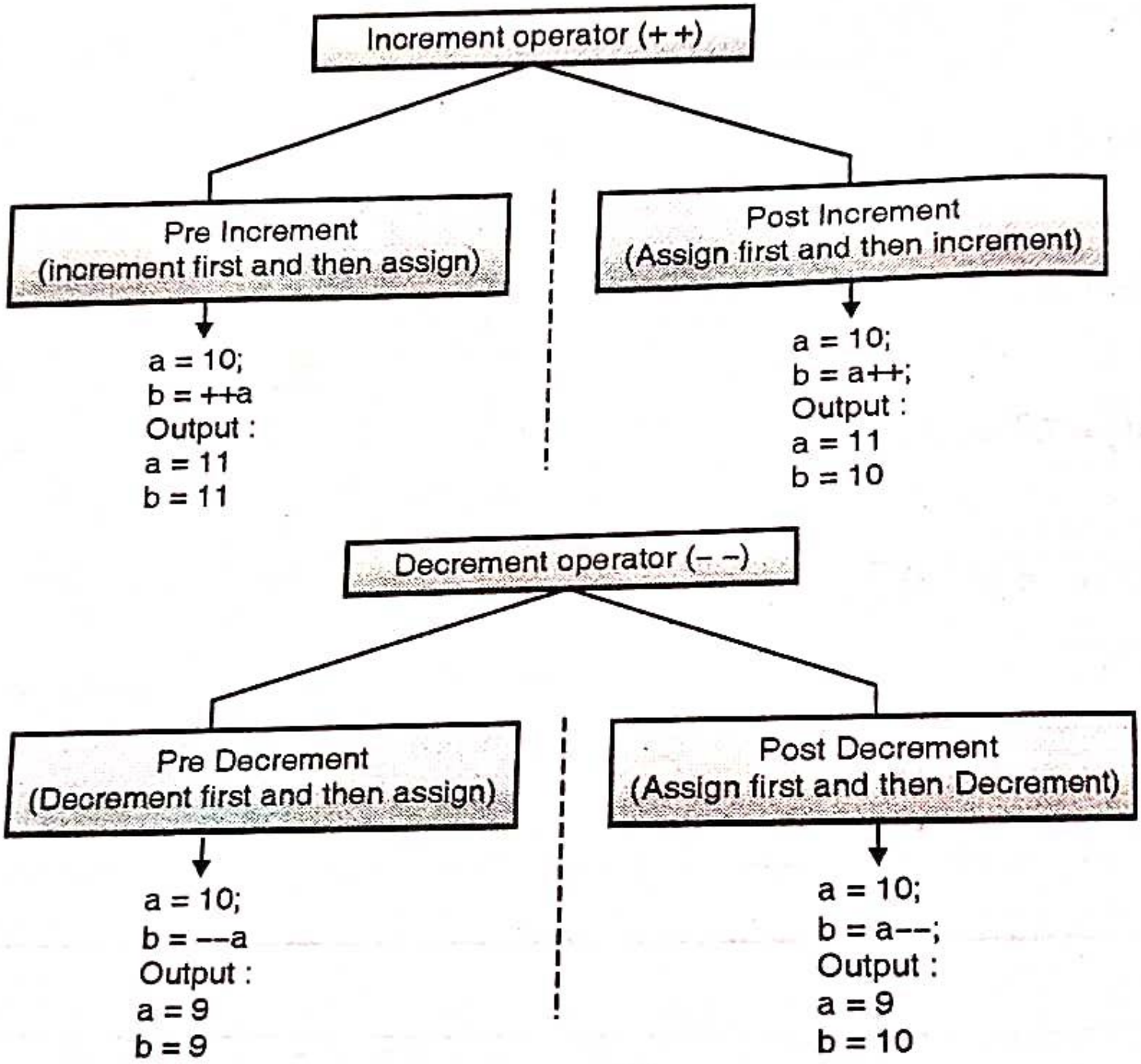
EXAMPLE 3.10 Here are several examples which illustrate the use of the unary minus operation.

```
-743          -0X7FFF       -0.2          -5E-8

-root1        -(x + y)      -3 * (x + y)
```

In each case the minus sign is followed by a numerical operand which may be an integer constant, a floating-point constant, a numeric variable or an arithmetic expression.

Increment and Decrement Operator

- The increment and decrement operators can each be utilized two different ways, depending on whether the operator is **written before or after the operand**(++i or i++)
- There are two other commonly used unary operators:
- The increment operator, ++, and the decrement operator, --.
- The ***increment operator*** causes its operand to be ***increased by 1***, whereas the **decrement operator** causes its operand to be **decreased by 1**


Increment operator (++)
Pre Increment
(increment first and then assign)
a = 10;
b = ++a
Output :
a = 11
b = 11
Post Increment
(Assign first and then increment)
a = 10;
b = a++;
Output :
a = 11
b = 10
Decrement operator (--)
Pre Decrement
(Decrement first and then assign)
a = 10;
b = --a
Output :
a = 9
b = 9
Post Decrement
(Assign first and then Decrement)
a = 10;
b = a--;
Output :
a = 9
b = 10

The conditional operator

- Simple conditional operations can be carried out with the **conditional operator (? :)**
- An expression that makes use of the **conditional operator** is called a **conditional expression**.

A conditional expression is written in the form

```
expression 1  ?  expression 2  :  expression 3
```

- When evaluating a conditional expression, expression 1 is evaluated first. If ***expression 1 is true*** (i.e., if its value is nonzero), then ***expression 2 is evaluated*** and this becomes the value of the conditional expression.
- if **expression 1 is false** (i.e., if its value is zero), then **expression 3 is evaluated** and this becomes the value of the conditional expression.
- **NOTE** that only one of the embedded expressions (either expression 2 or expression 3) is evaluated when determining the value of a conditional expression.

EXAMPLE 3.26 In the conditional expression shown below, assume that `i` is an integer variable.

```
(i < 0) ? 0 : 100
```

The expression `(i < 0)` is evaluated first. If it is true (i.e., if the value of `i` is less than 0), the entire conditional expression takes on the value 0. Otherwise (if the value of `i` is not less than 0), the entire conditional expression takes on the value 100.

Precedence and order of Evaluation

- The operator precedence is a theory that **decides** about the **priority of processing/evaluating** the given expression
- **Operator precedence** will decide which operator is to be evaluated first, and which operator should the order.
- **Associativity** describes whether the expression is to be evaluated from right to left **or** left to right.

Operator	*Description*	*Associativity*	*Rank*
( ) [ ]	Function call Aray element reference	Left to right	1
+ – ++ – – ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical negation Ones complement Pointer reference (indirection) Address Size of an object Type cast (conversion)	Right to left	2
* / %	Multiplication Division Modulus	Left to right	3
+ –	Addition Subtraction	Left to right	4
<< >>	Left shift Right shift	Left to right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right	6
== |=	Equality Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
= * = /= %= += –= &= ^= |= <<= >>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

- E.g.

$$
\begin{aligned}
X &= 15 - 2 * (6 + 18) / 3 + 6 \\
&= 15 - 2 * 24 / 3 + 6 \\
&= 15 - 48/3 + 6 \\
&= 15 - 16 + 6 \\
&= -1 + 6 \\
&= 5
\end{aligned}
$$

Initialization

- Variables are arbitrary(random) names given to a memory location in the system. These memory locations addresses in the memory.
- To facilitate the fetching of these memory addresses, variables are used.
- The memory location referred to by this variable holds a value of our interest.
- Now, these variables once declared, are assigned some value.
- This **assignment** of value to these variables is called **initialization of variables**.

- **Initialization of a variable is of two types:**
- **Static Initialization:** Here, the variable is **assigned a value in advance**. This variable then acts as a constant.
- **Dynamic Initialization:** Here, the variable is **assigned a value at the run time**. The value of this variable can be altered every time the program is being run.

Different ways of initializing a variable in C

- **Method 1 (Static Initialization : Declaring the variable and then initializing it):**

```
int a;
a = 5;
```

- **Method 2 (Static Initialization : Declaring and Initializing the variable together)**

```
int a = 5;
```

- **Method 3 (Static Initialization : Declaring multiple variables simultaneously and then initializing them separately)**

```
int a, b;
a = 5;
b = 10;
```

- **Method 4 (Static Initialization : Declaring multiple variables simultaneously and then initializing them simultaneously)**

```
int a, b;
a = b = 10;
```

```
int a, b = 10, c = 20;
```

- **Method 5 (Dynamic Initialization : Value is being assigned to variable at run time)**

```
int a;
printf("Enter the value of a");
scanf("%d", &a);
```

C Preprocessor

- The C **Preprocessor** is ***not a part*** of the **compiler**, but is a **separate step** in the **compilation process**
- C Preprocessor is just a text substitution tool and it **instructs** the **compiler** to **do required pre-processing *before*** the **actual compilation**
- Preprocessors are programs that **process** our **source code *before* compilation.**
- **Preprocessor programs *provide preprocessors directives*** which **tell** the **compiler** to **preprocess** the **source code *before* compiling**.
- All of these preprocessor directives begin with a **“ # ” (hash)** symbol.

The following section lists down all the important preprocessor directives

- **Directive & Description**
- #define

Substitutes a preprocessor macro.

- #include

Inserts a particular header from another file.

- #if

Tests if a compile time condition is true.

- #else

The alternative for #if

- #elif

#else and #if in one statement

- #endif

Ends preprocessor conditional

#include

- The #include preprocessor is used to include header files to C programs.

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

#define

- The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

C Macros

- A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive.

Object-like Macros

Function-like Macros

Object-like Macros

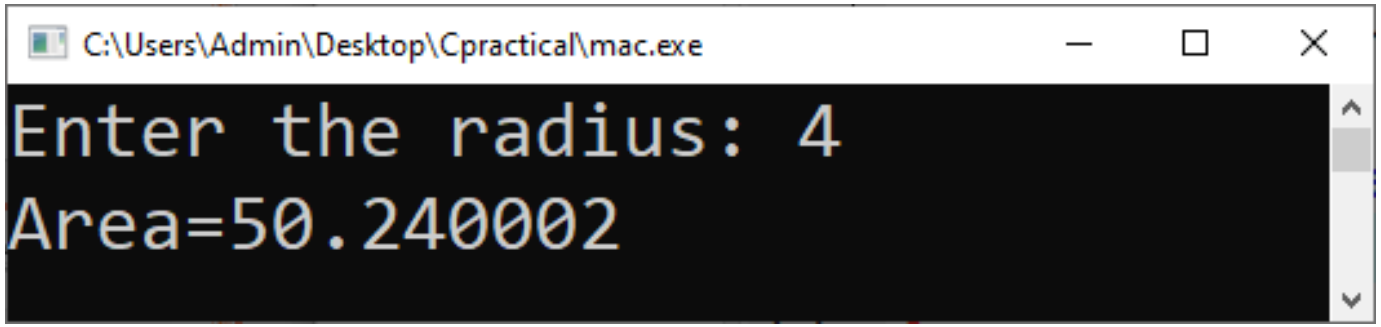
- The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants.

mac.c

```c
#include <stdio.h>
#define PI 3.14
void main()
{
    float radius, area;
    printf("Enter the radius: ");
    scanf("%f", &radius);

    area = PI*radius*radius;  // Notice, the use of PI

    printf("Area=%f",area);
    getch();
}
```

Here, PI is the macro name which will be replaced by the value 3.14.

Output :


C:\Users\Admin\Desktop\Cpractical\mac.exe
Enter the radius: 4
Area=50.240002

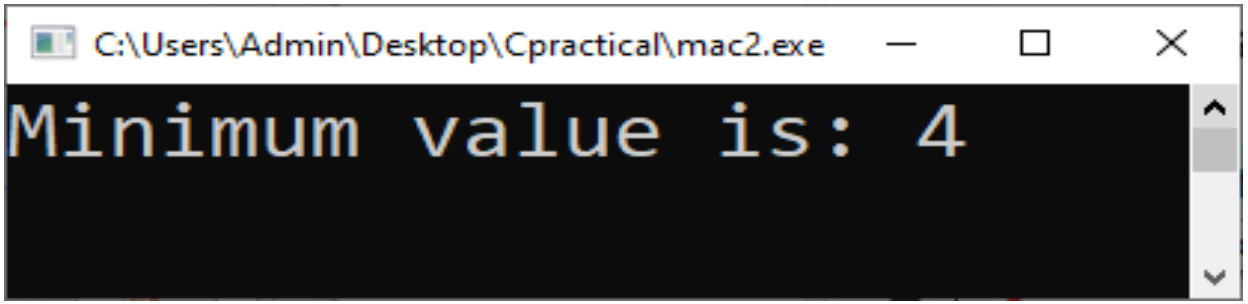
- **Function-like Macros**
  - The function-like macro looks like function call.

mac2.c

```c
#include<stdio.h>
#define MIN(a,b) ((a<b)?(a):(b))
void main()
{
    printf("Minimum value is: %d\n", MIN(4,20));
    getch();
}
```

Here, MIN is the macro name.

Output :

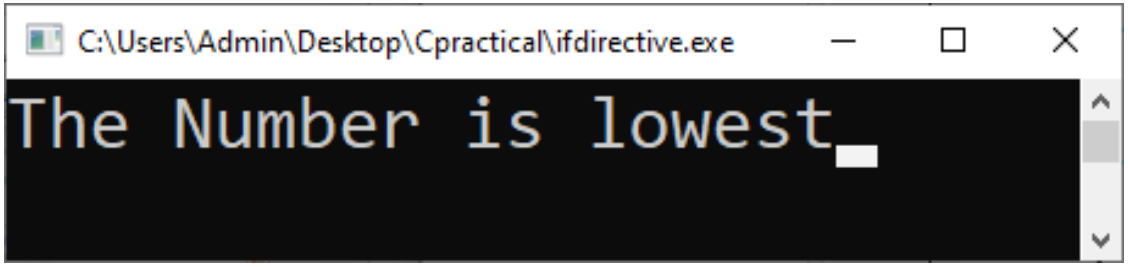


- **#if**
- The #if preprocessor directive evaluates the expression or condition.
- If condition is true, it executes the code otherwise #else or #elif or #endif code is executed.

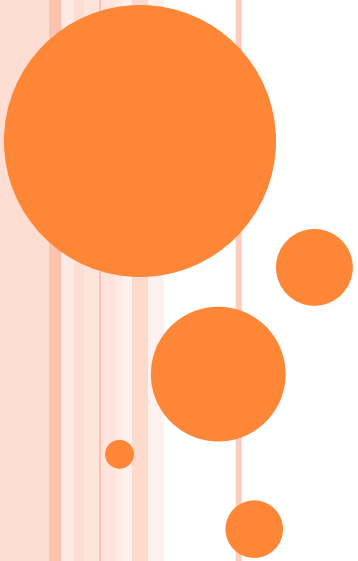
ifdirective.c

```c
#include<stdio.h>

#define NUMBER 45
void main()
{
    #if NUMBER > 55
    printf("The Number is greatest");
    #elif  Number < 55
    printf("The Number is lowest");
    #else
    printf("The Number is same");
    #endif
    getch();
}
```

Output :



3. Control flow



Statements and blocks

- An **expression** such as x = 0 or i++ or printf(...) *becomes* a **statement** when it is **followed by** a ***semicolon***, as in x = 0; i++; printf(...);
- Braces { and } are used to group declarations and statements together into a **compound statement**, or **block**, so that they are syntactically equivalent to a single statement

Decision making within a program

- Decision-making or Control statements are the statements that are used to verify a given condition and decide whether a block of statements gets executed or not based on the condition result.
- In decision control statements (C if else and nested if), group of **statements are executed** when **condition is true.**
- If ***condition is false***, then ***else part statements are executed.***
- There are **3 types** of **decision making control statements** in C language. They are,

I. if statements
II. if else statements
III. nested if statements

If statement

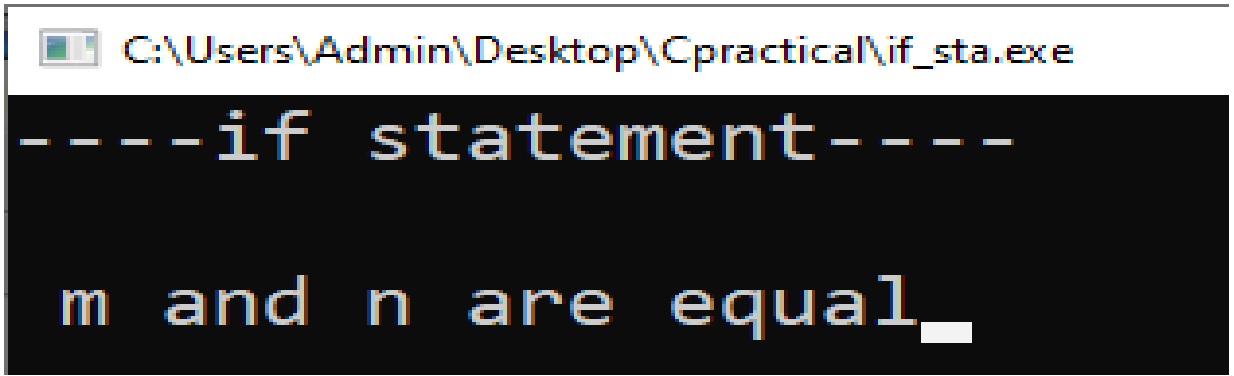
- The if statement is a powerful decision-making statement and is used to control the flow of execution of statements
- In these type of statements, if condition is true, then respective block of code is executed.
- **Syntax:**

```
if (condition)
{
Statements;
 }
```

if_sta.c

```c
#include<stdio.h>
#include<conio.h>
void main()
{
    int m=20,n=20;
    printf("----if statement----");
    if(m==n)
    {
        printf("\n\n m and n are equal");
    }
    getch();

}
```

Output :



IF-ELSE STATEMENT

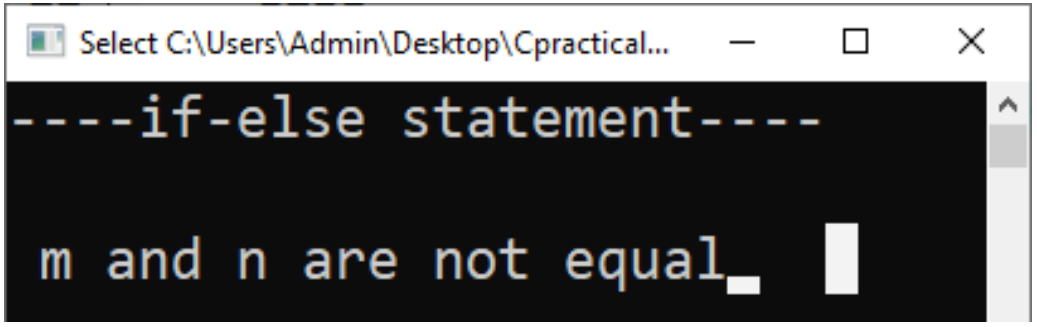
- The if-else statement is used to express decisions
- In these type of statements, group of statements are executed when condition is true. If condition is false, then else part statements are executed.
- **Syntax :**

```
if (condition)
{
Statements;
}
else
{
Statements;
}
```

ifelse.c

```c
#include<stdio.h>
#include<conio.h>
void main()
{
    int m=20,n=30;
    printf("----if-else statement----");

    if(m==n)
    {
        printf("\n\n m and n are equal");
    }
    else
    {
        printf("\n\n m and n are not equal");
    }
    getch();
}
```

Output :



ELSE-IF (NESTED IF) STATEMENT

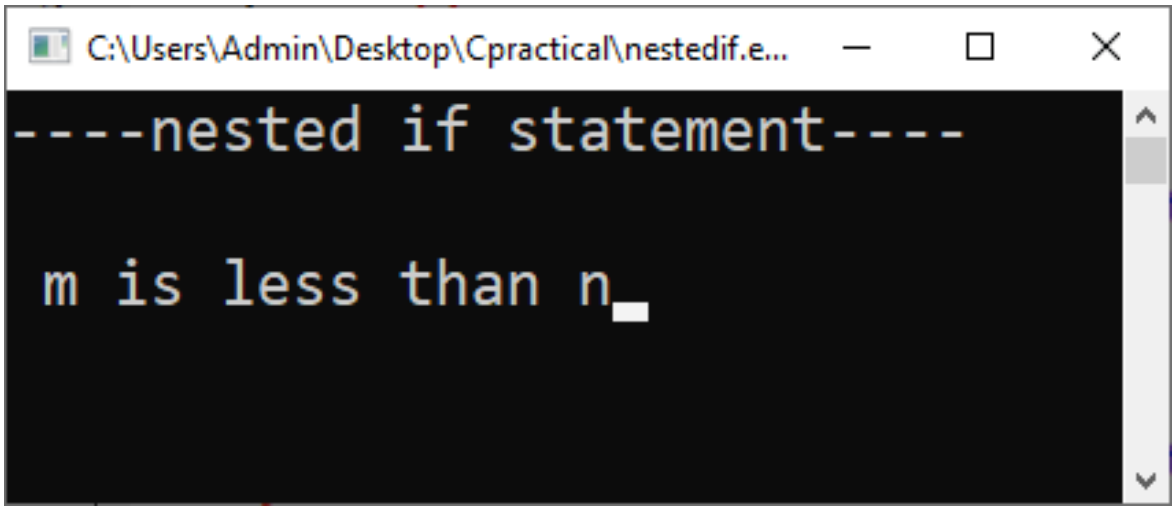
- In this type of statements, if condition 1 is false, then condition 2 is checked and statements are executed if it is true.
- When all the n conditions become false ,then the final else containing the default-statement will be executed.
- This construct is known as the **else if ladder**
- **Syntax :**

```
if (condition1)
{
Statements;
}
else  if(condition2)
{
Statements;
}
else
{
Statements;
}
```

nestedif.c

```c
#include<stdio.h>
#include<conio.h>
void main()
{
    int m=20,n=30;
    printf("----nested if statement----");

    if(m>n)
    {
        printf("\n\n m is greater than n");
    }
    else if(m<n)
    {
        printf("\n\n m is less than n");
    }
    else
    {
        printf("\n\n m is equal to n");
    }
    getch();
}
```

Output :


C:\Users\Admin\Desktop\Cpractical\nestedif.e...
----nested if statement----
m is less than n

"If", "else" and "nested if" decision control statements in C:

- Syntax for each C decision control statements are given in below table with description.

Decision control statements	Syntax	Description

If	if (condition)
	{
	Statements;
	}
		In these type of statements, if condition is true, then respective block of code is executed.

if...else	if (condition)
	{
	Statements;
	}
	else
	{
	Statements;
	}
		In these type of statements, group of statements are executed when condition is true. If condition is false, then else part statements are executed.

nested if	if (condition1)
	{
	Statements;
	}
	else if(condition2)
	{
	Statements;
	}
	else
	{
	Statements;
	}
		If condition 1 is false, then condition 2 is checked and statements are executed if it is true. If condition 2 also gets failure, then else part is executed.

Switch

- We have seen that when one of the many alternatives is to be selected, we can use an if statement to control the selection. However, the complexity of such a program increases dramatically when the confuse even the person who designed it. Fortunately, C has a built-in multiway decision statement known as a **switch**.
- The **switch** statement is a **multi-way decision**.
- The switch statement causes a particular group of statements to be chosen from several available groups

- The control statement that allows us to make a **decision** *from* the **number of choices** is **called** a **switch**, or more correctly a **switch case-default**
- The switch statement tests the value of a given variable (or expression) against a list of values and when a match is found, a block of statements associated with that is executed.
- The switch statement in C is an **alternate *to if-else-if ladder statement*** which allows us to execute multiple operations
- The ***case*** *value* **must** be an ***integer or character constant.***
- The case labeled ***default*** is ***executed*** if ***none of the other cases are satisfied***

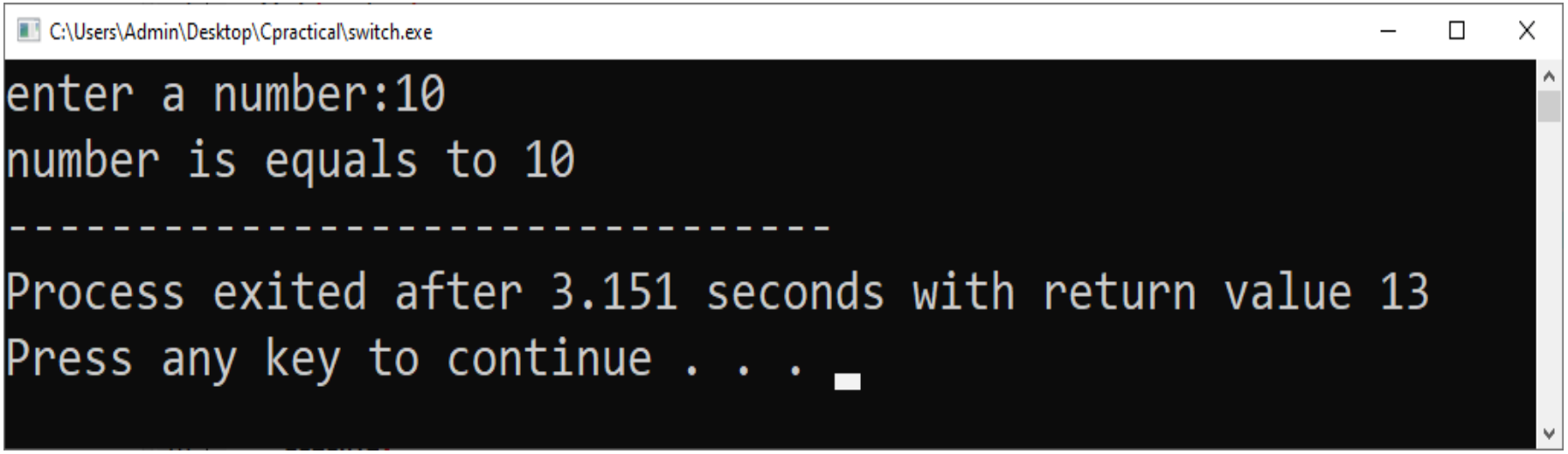
- **Syntax:**

```
switch (expression)
{
case label1:
statements;
break;
case label2:
statements;
break;
default:
statements;
break;
}
```

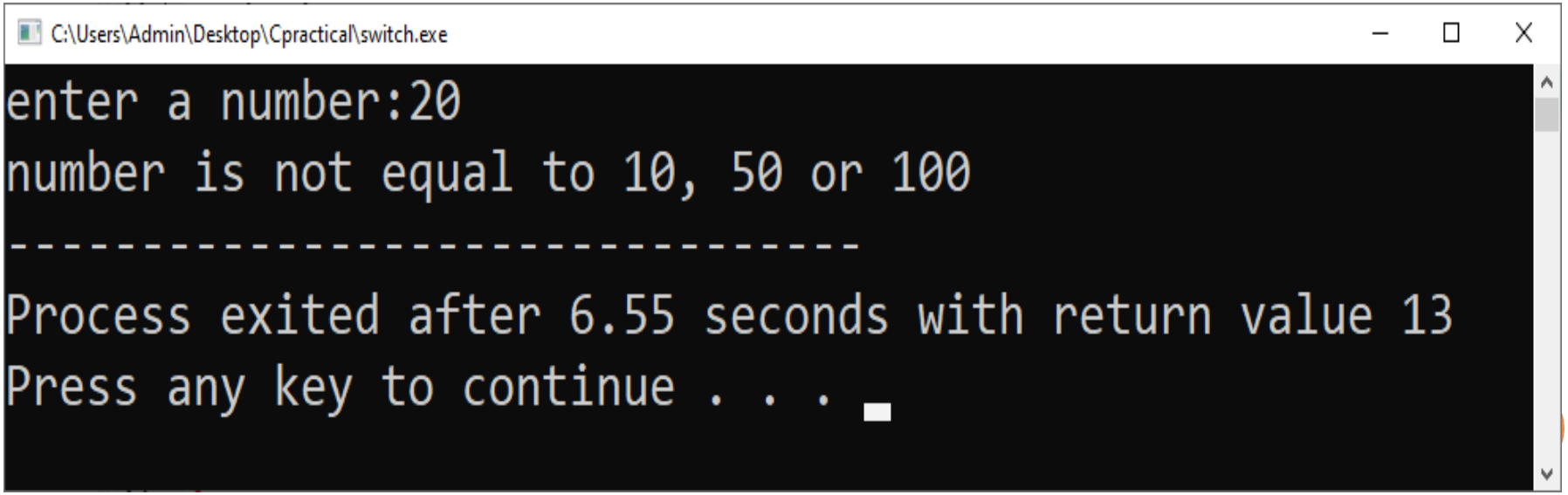
switch.c

```c
#include<stdio.h>
void main()
{
    int number;
    printf("enter a number:");
    scanf("%d",&number);

    switch(number)
    {
        case 10:
        printf("number is equals to 10");
        break;
        case 50:
        printf("number is equal to 50");
        break;
        case 100:
        printf("number is equal to 100");
        break;

        default:
        printf("number is not equal to 10, 50 or 100");
    }
    getch();
}
```

Output :


C:\Users\Admin\Desktop\Cpractical\switch.exe
enter a number:10
number is equals to 10

Process exited after 3.151 seconds with return value 13
Press any key to continue . . .


C:\Users\Admin\Desktop\Cpractical\switch.exe
enter a number:20
number is not equal to 10, 50 or 100

Process exited after 6.55 seconds with return value 13
Press any key to continue . . .

Loops

- In computer programming, a loop **is used** to **achieve the task** of **executing** a **block of instructions multiple times**.
- Process of repeatedly executing a block of statements is known as **“looping”**
- loops are used to execute a single statement or a set of statements, repeatedly, **until a particular condition is satisfied**.
- Loop control statements in C are used to perform looping operations until the given condition is true.
- Control comes out of the loop statements once condition becomes false.

- Loops make a program more readable and enhance its efficiency by simplifying complex tasks.
- Loop consists of two parts:
- **Body of Loop:** consists of a set of statements that need to be continuously executed
- **Conditional Statement :** It is a condition. If it is true, then the next iteration is executed else the execution flow exits the loop.
- ***Types of loop control statements in C:***
- There are 3 types of loop control statements in C language. They are,

1. While
2. do-while
3. for

While loop

- The while statement is used to carry out looping operations, in which a group of statements is executed repeatedly, until some condition has been satisfied.
- The **while *tests*** the ***condition*** **before *executing any of the statements*** within the while loop(*First checked the condition and then statement is executed*)
- It is known as **entry-controlled loop** (**entry controlled loop** is *where the test condition is tested before executing the body of a loop*)

- **Syntax:**

```
while (condition)
{
statements;
 }
```

while.c

```c
#include <stdio.h>
void main()
{
    int i =3;
    printf("---Use of While Loop---\n\n");
    while(i<10)
    {
        printf("%d\t",i);
        i++;
    }
    getch();
}
```

Output :

For loop

- In for loop control statement, loop is executed until a particular condition is satisfied.
- The **for loop** is another **entry-controlled loop**
- **initialization, condition and increment/decrement** is part of for loop.

1. **initialization** : This step allows to declare and initialize any loop control variables
2. **condition** : If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute
3. **increment/decrement** : This statement allows to update any loop control variables

- **Syntax:**

```
for(exp1;exp2;exp3)
{
Statements;
}
```

Where,
exp1 – variable initialization
(e.g. i=0, j=2)
exp2 – condition checking
(e.g. i>5, j<3)
exp3 – increment/decrement
(e.g. ++i, j--)

forloop.c

```c
#include <stdio.h>
void main()
{
  int i;
  printf("**** For Loop****\n\n");
  for (i = 1; i<=10;i++)
  {
    printf("%d ", i);
  }
  getch();
}
```

Output :

DO-WHILE LOOP

- do…while loop is similar to while loop in that the loop continues as long as the specified loop condition remains true.
- The **main difference** is that do..while loop **checks for the condition *after* executing the statements** (i.e. *it tests the condition after executed the statement*)
- In **do-while**, the block of statements is executed **at least once**, even if the **condition fails** for the first time.
- It is also called as **exit-controlled loop** (**exit controlled loop** is *where the condition is checked after the loop's body is executed*)

- **Syntax:**

```
do
{
statements;
 }while (condition);
```

dowhile.c

```c
#include <stdio.h>
void main()
{
    int i =3;
    printf("###### Use of do..While Loop ######\n\n");
    do
    {
        printf("%d\t",i);
        i++;
    }while(i<10);

    getch();
}
```

Output:

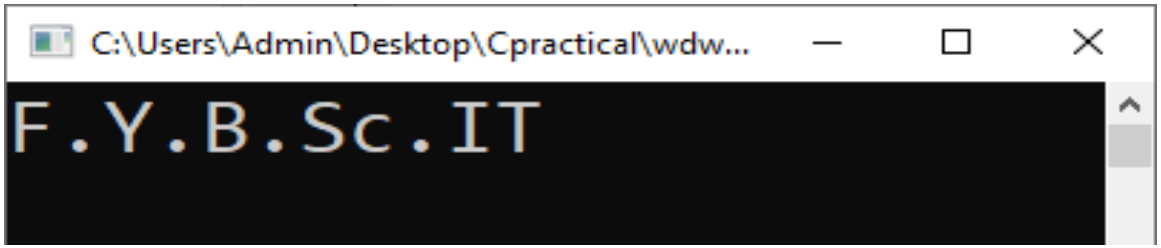
```
C:\Users\Admin\Desktop\Cpractical\dowhile.exe
###### Use of do..While Loop ######

3       4       5       6       7       8       9
```

Difference Between Do-While and While Loop

wdw.c

```c
#include <stdio.h>
void main()
{
    int a=4,b=1;

    do
    {
        printf("F.Y.B.Sc.IT");

    }while(a<b);
    getch();
}
```

Output :



wdw.c

```c
#include <stdio.h>
void main()
{
    int a=4,b=1;

    while(a<b)
    {
        printf("F.Y.B.Sc.IT");

    }
    getch();
}
```

Output :

Break statement

- The **break statement** is **used** to **terminate loops** or **to exit from a switch**
- It can be used within a for, while, do -while, or switch statement
- In **Switch statement**, noticed that each group of statements ends with a break statement, in order to transfer control out of the switch statement. The break statement is **required** within **each of the groups (*cases*)**, in order to prevent the succeeding groups of statements from executing. The **last group (*default*) does not require** a break statement, since control will automatically be transferred out of the switch statement after the last group has been executed. *This last break statement is included, however, as a matter of good programming practice, so that it will be present if another group of statements is added later*.

- If a break statement is included in a **while, do - while or for loop**, then control will immediately be transferred out of the loop when the break statement is encountered. This provides a convenient way to terminate the loop.
- **Syntax:**

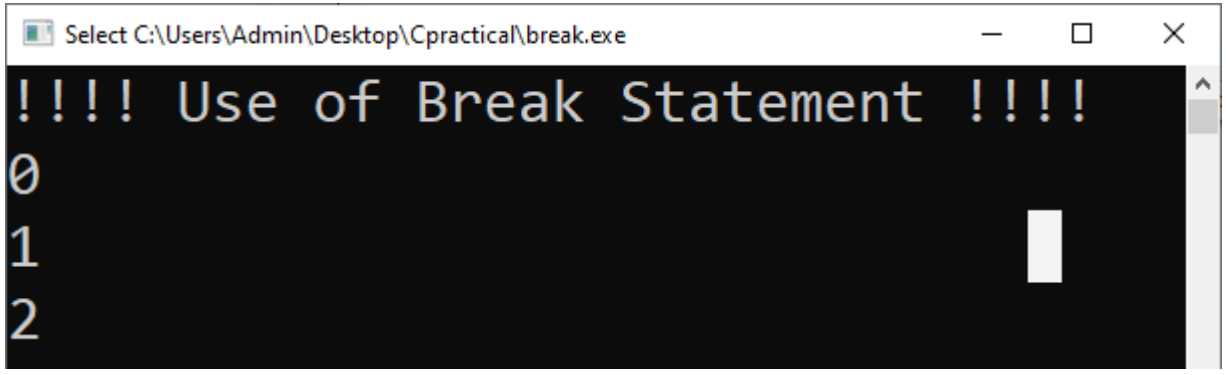
The break statement is written simply as

```
break;
```

break.c

```c
#include<stdio.h>
#include<conio.h>
void main()
{
    int i=0;
    printf("!!!! Use of Break Statement !!!!\n");

    for (i=0; i<5; i++)
    {
        printf("%d\n", i);

            if (i==2)
                break;
    }
    getch();
}
```

Output :



CONTINUE STATEMENT

- The **continue statement passes control** to the **next iteration**(i.e. *it is useful to continue with the next iteration(repetition) of a loop*)
- The loop **does not *terminate*** when a continue statement is encountered
- The continue statement **skips** some **lines of code inside the loop and continues** with the **next iteration**
- **Syntax:**

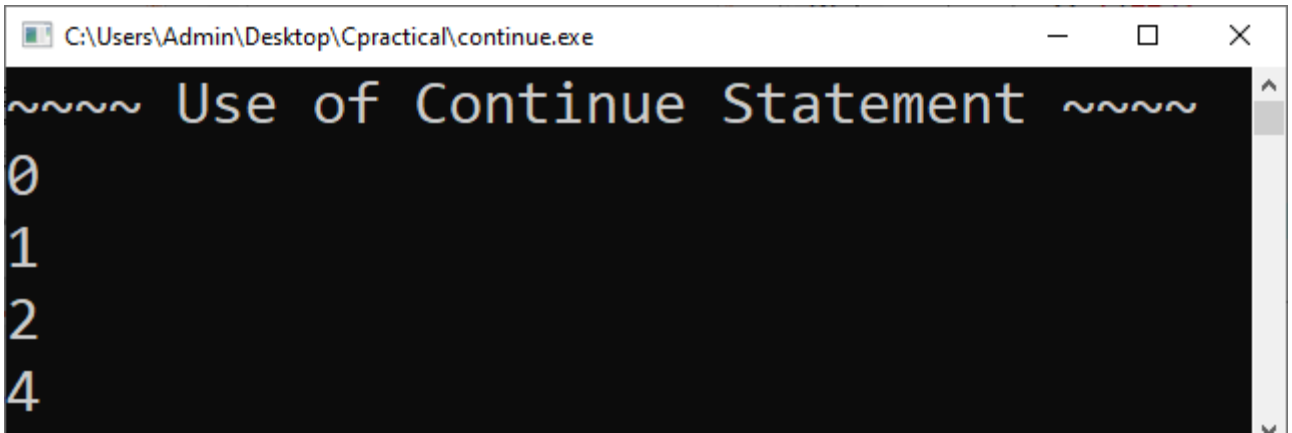
The continue statement can be included within a while, a do - while or a for statement. It is written simply as

```
continue;
```

continue.c

```c
#include<stdio.h>
#include<conio.h>
void main()
{
    int i=0;
    printf("~~~~ Use of Continue Statement ~~~~\n");

    for (i=0; i<5; i++)
    {
        if (i==3)
        {
            continue;
        }
        printf("%d\n", i);
    }
    getch();
}
```

Output :



GOTO AND LABELS STATEMENTS

- The **goto statement** is **used** to **alter** the normal **sequence of program** execution by **transferring control to some other part** of the program.
- The **goto statements** is used to **transfer** the normal flow of a program **to the specified label** in the program
- The label is an identifier that is used to **label** the target statement to **which control will be transferred**.
- The target statement must be labeled, and the **label** must be **followed** by a **colon (:)**
- Each labeled statement within the program (more precisely, within the current function) must have a **unique label**

- **Syntax:**

In its general form, the goto statement is written as

```
goto label;
```

Thus, the target statement will appear as

```
label: statement
```

goto.c

```c
#include<stdio.h>
#include<conio.h>
void main()
{
    int i=0;
    printf("!!!! Use of goto Statement !!!!\n");

    for (i=0; i<6; i++)
    {
        if (i==3)
        goto abc;
        printf("%d\n", i);
    }
    abc: printf("we are in the label");

    getch();
}
```

Output :