

The page features decorative leaf patterns at the top and bottom. The leaves are rendered in a dark blue or purple color with fine lines indicating veins. The background is a textured, mottled purple and blue gradient.

PROGRAMMING FUNDAMENTALS

NAME: MUZAMIL AHMAD KHAN
STUDENT ID: 16941
DEPARTMENT: BS-SE
FINAL TERM ASSIGNMENT



Iqra National University Peshawar Pakistan

Department of Computer Science
Spring Semester, Final Term Exam, June 2020

Paper:	Programming Fundamentals	Date and Starting Time:	26/June/2020, 9:00 am
Program:	BS (CS & SE)	Uploading Date and End Time:	26/June/2020, 3:00 pm
Teacher Name:	Dr. Fazal-e-Malik	Marks	50

Note: Attempt all Questions. Help can be taken from net where ever is required.

Q1).

A). What is the purpose of if statement? Discuss its two different forms with examples.

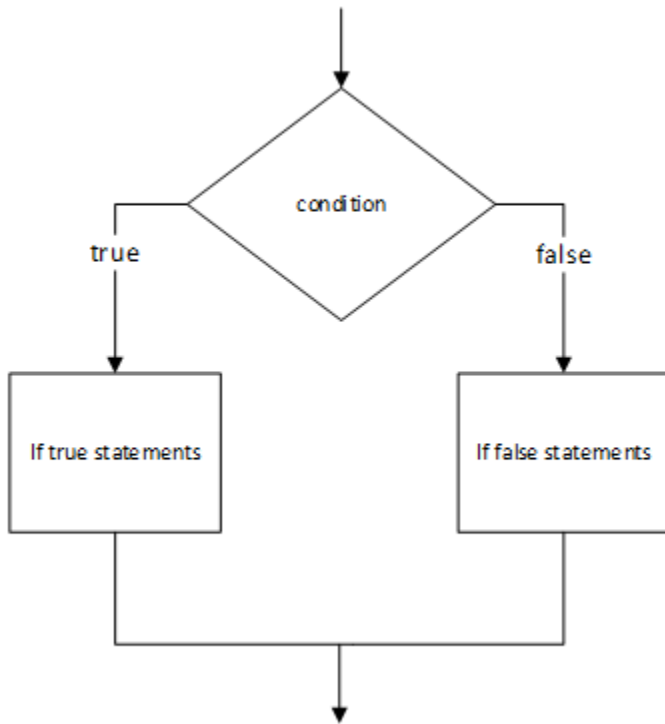
Ans). The **if/else** statement

The if/else statement extends the if statement by specifying an action if the if (*true/false expression*) is false.

```
if (condition){ // do this if condition is true // if true statements}else{ // do this if condition is false // if false statements}
```

With the if statement, a program will execute the true code block or do nothing. With the if/else statement, the program will execute either the true code block or the false code block so something is always executed with an if/else statement.

Flow chart view of if/else



Where to use two statements versus one if/else statement

Use two if statements if both if statement conditions could be true at the same time.

In this example, both conditions can be true. You can pass and do great at the same time.

Use an if/else statement if the two conditions are mutually exclusive meaning if one condition is true the other condition must be false.

```
if (testScore > 60) cout << "You pass" << endl; if (testScore > 90) cout << "You did great" << endl;
```

For example, before noon (AM) and after noon (PM) are mutually exclusive. It is either one or the other. Using a 24-hour time system (12 is

noon and 24 is midnight), if the time of day is ≥ 12 it is PM, else it is AM.

```
if (timeOfDay >=12) cout << "PM" << endl;else // it must
be AM cout << "AM" << endl;
```

Curly brackets with if/else statements

The else part of the if/else statement follows the same rules as the if part. If you want to execute multiple statements for the else condition, enclose the code in curly brackets. If you only need to execute a single statement for the else condition, you do not need to use curly brackets.

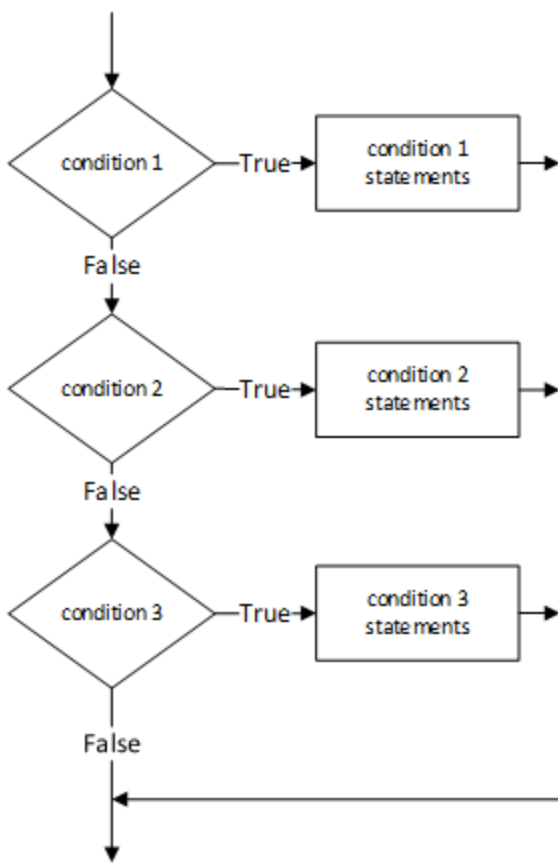
```
if (condition){ multiple statements}else single
statementif (condition) { multiple statements}else
{ multiple statements}
```

The **if/else if** statement

The if/else if statement allows you to create a chain of if statements. The if statements are evaluated in order until one of the if expressions is true or the end of the if/else if chain is reached. If the end of the if/else if chain is reached without a true expression, no code blocks are executed.

```
if (condition1){ // do this if condition1 is true //
condition 1 statements // then exit if/else if}else if
(condition2){ // do this if condition2 is true //
condition 2 statements // then exit if/else if}else if
(condition3) { // do this if condition3 is true //
condition3 statements // then exit if/else if }//
continuation point after if/else if is complete
```

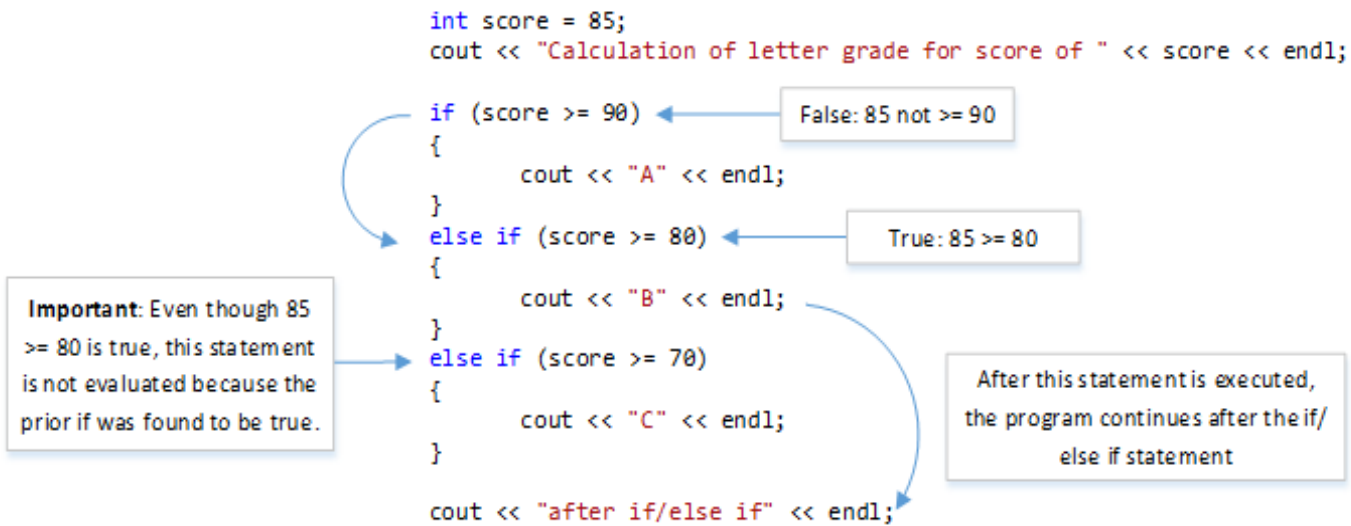
Flow chart view



if/else if flow control

Note: It is very important to understand that once a condition is found to be true, **no other if statements are evaluated** and once the code block for the true statement is completed, the program continues from the end of the if/else if statement.

Let's look at an example of this.



```

Calculation of letter grade for score of 85
B
after if/else if

```

Where to use if/else if

Use the “if/else if” if you need to choose either one or none of a series of options. You could write the above program using separate if statements such as show below but the if/else if approach is cleaner and better indicates that the program should choose either one or none of the options.

```

if (score >= 90)
{
    cout << "A" << endl;
}

if (score < 90 && score >= 80)
{
    cout << "B" << endl;
}

if (score < 80 && score >= 70)
{
    cout << "C" << endl;
}

```

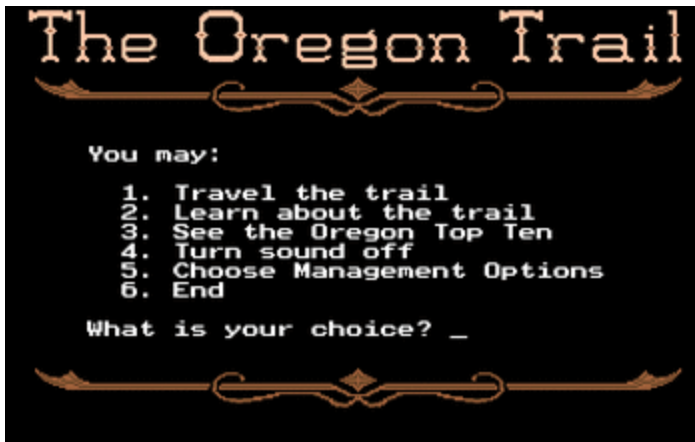
Add a trailing **else** to the **if/else if** statement

You can add a trailing else statement to the if/else if statement if you want to execute code if none of the if statements are true.

```
if (condition1)
{
    // do this if condition is true
    // condition1 statements
    // then exit if/else if
}
else if (condition2)
{
    // do this if condition2 is true
    // condition2 statements
    // then exit if/else if
}
else if (condition3)
{
    // do this if condition3 is true
    // condition3 statements
    // then exit if/else if
}
else // if no if condition was true
{
    // do this if no if conditions were true
    // trailing else statements
}
```

Only executed if condition1,
condition2, and condition3
are all false

Menu driven programs



The C++ *Early Objects* book contains a discussion (p. 181) of menu driven programs. In a menu driven program, the user is presented with a menu or series of options. The user then selects an option and the program proceeds with execution based on the menu selection. if/else if statements are often used to control program flow in menu driven programs.

Q1).

B). Write a C++ program to read two numbers from keyboard and then find the LARGEST number of them.

Ans). `#include <iostream>`
`using namespace std;`

```
int main()
{
    int num1, num2;
    cout<<"Enter first number:";
    cin>>num1;
    cout<<"Enter second number:";
    cin>>num2;
    if(num1>num2)
    {
        cout<<"First number "<<num1<<" is the largest";
    }
    Else
    {
        cout<<"Second number "<<num2<<" is the largest";
    }
    return 0;
}
```

Q2).

A). What are the Logical Operators? Explain them.

ANS). LOGICAL OPERATORS

Kenneth Leroy Busbee and Dave Braunschweig
Overview

A **logical operator** is a symbol or word used to connect two or more expressions such that the value of the compound expression produced depends only on that of the original expressions and on the meaning of the operator.^[1] Common logical operators include AND, OR, and NOT.

Discussion

Within most languages, expressions that yield Boolean data type values are divided into two groups. One group uses the relational operators within their expressions and the other group uses logical operators within their expressions.

The logical operators are often used to help create a test expression that controls program flow. This type of expression is also known as a Boolean expression because they create a Boolean answer or value when evaluated. There are three common logical operators that give a Boolean value by manipulating other Boolean operand(s). Operator symbols and/or names vary with different programming languages:

Language	AND	OR	NOT
C++	&&		!
C#	&&		!
Java	&&		!
JavaScript	&&		!
Python	and	or	not
Swift	&&		!

The vertical dashes or piping symbol is found on the same key as the backslash \. You use the SHIFT key to get it. It is just above the Enter key on most keyboards. It may be a solid vertical line on some keyboards and show as a solid vertical line on some print fonts.

In most languages there are strict rules for forming proper logical expressions. An example is:

6 > 4 && 2 <= 14

6 > 4 and 2 <= 14

This expression has two relational operators and one logical operator. Using the precedence of operator rules the two "relational comparison" operators will be done before the "logical and" operator. Thus:

true && true

True and True

The final evaluation of the expression is: true.

We can say this in English as: It is true that six is greater than four and that two is less than or equal to fourteen.

When forming logical expressions programmers often use parentheses (even when not technically needed) to make the logic of the expression very clear. Consider the above complex Boolean expression rewritten:

(6 > 4) && (2 <= 14)

(6 > 4) and (2 <= 14)

Most programming languages recognize any non-zero value as true. This makes the following a valid expression:

6 > 4 && 8

6 > 4 and 8

But remember the order of operations. In English, this is six is greater than four and eight is not zero. Thus,

true && true

True and True

To compare 6 to both 4 and 8 would instead be written as:

6 > 4 && 6 > 8

6 > 4 and 6 > 8

This would evaluate to false as:

true && false

True and False

Truth Tables

A common way to show logical relationships is in truth tables.

Logical and (&&)

x	y	x and y
false	false	false
false	true	false
true	false	false
true	true	true

Logical or (||)

x	y	x or y
false	false	false
false	true	true
true	false	true
true	true	true

Logical not (!)

x	not x
false	true
true	false

Examples

I call this example of why I hate "and" and love "or".

Every day as I came home from school on Monday through Thursday; I would ask my mother, "May I go outside and play?" She would answer, "If your room is clean and your homework is done then you may go outside and play." I learned to hate the word "and". I could manage to get one of the tasks done and have some time to play before dinner, but both of them... well, I hated "and".

On Friday my mother took a more relaxed viewpoint and when asked if I could go outside and play she responded, "If your room is clean or your homework is done then you may go outside and play." I learned to clean my room quickly on Friday afternoon. Well, needless to say, I loved "or".

For the next example, just imagine a teenager talking to their mother. During the conversation, mom says, "After all, your Dad is reasonable!" The teenager says, "Reasonable. (short pause) Not."

Maybe college professors will think that all their students studied for the exam. Ha ha! Not. Well, I hope you get the point.

Examples:

- $25 < 7 \ || \ 15 > 36$
- $15 > 36 \ || \ 3 < 7$
- $14 > 7 \ \&\& \ 5 \leq 5$
- $4 > 3 \ \&\& \ 17 \leq 7$
- $! \text{ false}$
- $! (13 \neq 7)$
- $9 \neq 7 \ \&\& \ ! 0$
- $5 > 1 \ \&\& \ 7$

More examples:

- $25 < 7 \ \text{or} \ 15 > 36$
- $15 > 36 \ \text{or} \ 3 < 7$
- $14 > 7 \ \text{and} \ 5 \leq 5$
- $4 > 3 \ \text{and} \ 17 \leq 7$
- not False
- $\text{not} (13 \neq 7)$
- $9 \neq 7 \ \text{and} \ \text{not} 0$
- $5 > 1 \ \text{and} \ 7$

Key Terms

logical operator

An operator used to create complex Boolean expressions.

truth tables

A common way to show logical relationships

|| (OR)

The "OR" operator is represented with two vertical line symbols:

```
result = a || b;
```

In classical programming, the logical OR is meant to manipulate boolean values only. If any of its arguments are true, it returns true, otherwise it returns false.

In JavaScript, the operator is a little bit trickier and more powerful. But first, let's see what happens with boolean values.

There are four possible logical combinations:

```
alert( true || true ); // true
alert( false || true ); // true
alert( true || false ); // true
alert( false || false ); // false
```

As we can see, the result is always true except for the case when both operands are false.

If an operand is not a boolean, it's converted to a boolean for the evaluation.

For instance, the number 1 is treated as true, the number 0 as false:

```
if (1 || 0) { // works just like if( true || false )
  alert( 'truthy!' );}
```

Most of the time, OR || is used in an if statement to test if *any* of the given conditions is true.

For example:

```
let hour = 9;
if (hour < 10 || hour > 18) { alert( 'The office is closed.' );}
```

We can pass more conditions:

```
let hour = 12;
let isWeekend = true;
if (hour < 10 || hour > 18 || isWeekend) { alert( 'The office is closed.' ); // it is the weekend }
```

OR "||" finds the first truthy value

The logic described above is somewhat classical. Now, let's bring in the "extra" features of JavaScript.

The extended algorithm works as follows.

Given multiple OR'ed values:

```
result = value1 || value2 || value3;
```

The OR `||` operator does the following:

- Evaluates operands from left to right.
- For each operand, converts it to boolean. If the result is true, stops and returns the original value of that operand.
- If all operands have been evaluated (i.e. all were false), returns the last operand.

A value is returned in its original form, without the conversion.

In other words, a chain of OR `||` returns the first truthy value or the last one if no truthy value is found.

For instance:

```
alert( 1 || 0 ); // 1 (1 is truthy)
alert( null || 1 ); // 1 (1 is the first truthy value)
alert( null || 0 || 1 ); // 1 (the first truthy value)
alert( undefined || null || 0 ); // 0 (all falsy, returns the last value)
```

This leads to some interesting usage compared to a “pure, classical, boolean-only OR”.

1. Getting the first truthy value from a list of variables or expressions.

2. For instance, we have `firstName`, `lastName` and `nickName` variables, all optional.

3. Let's use OR `||` to choose the one that has the data and show it (or anonymous if nothing set):

```
4. let firstName = ""; let lastName = ""; let nickName =
   "SuperCoder"; alert( firstName || lastName || nickName
   || "Anonymous" ); // SuperCoder
```

5. If all variables were falsy, Anonymous would show up.

6. Short-circuit evaluation.

7. Another feature of OR `||` operator is the so-called “short-circuit” evaluation.

8. It means that `||` processes its arguments until the first truthy value is reached, and then the value is returned immediately, without even touching the other argument.

9. That importance of this feature becomes obvious if an operand isn't just a value, but an expression with a side effect, such as a variable assignment or a function call.
10. In the example below, only the second message is printed:
11.

```
true || alert("not printed");false || alert("printed");
```
12. In the first line, the OR `||` operator stops the evaluation immediately upon seeing `true`, so the `alert` isn't run.
13. Sometimes, people use this feature to execute commands only if the condition on the left part is falsy.

&& (AND)

The AND operator is represented with two ampersands `&&`:
`result = a && b;`

In classical programming, AND returns `true` if both operands are truthy and `false` otherwise:

```
alert( true && true ); // true
alert( false && true ); // false
// false
alert( true && false ); // false
alert( false && false ); // false
```

An example with `if`:

```
let hour = 12; let minute = 30;
if (hour == 12 && minute == 30) {
  alert( 'The time is 12:30' );
}
```

Just as with OR, any value is allowed as an operand of AND:

```
if (1 && 0) { // evaluated as true && false
  alert( "won't work, because the result is falsy" );
}
```

AND "&&" finds the first falsy value

Given multiple AND'ed values:

```
result = value1 && value2 && value3;
```

The AND `&&` operator does the following:

- Evaluates operands from left to right.

- For each operand, converts it to a boolean. If the result is false, stops and returns the original value of that operand.
- If all operands have been evaluated (i.e. all were truthy), returns the last operand.

In other words, AND returns the first falsy value or the last value if none were found.

The rules above are similar to OR. The difference is that AND returns the first *falsy* value while OR returns the first *truthy* one.

Examples:

```
// if the first operand is truthy, // AND returns the
second operand: alert( 1 && 0 ); // 0 alert( 1 && 5 ); //
5 // if the first operand is falsy, // AND returns it. The
second operand is ignored alert( null && 5 ); //
null alert( 0 && "no matter what" ); // 0
```

We can also pass several values in a row. See how the first falsy one is returned:

```
alert( 1 && 2 && null && 3 ); // null
```

When all values are truthy, the last value is returned:

```
alert( 1 && 2 && 3 ); // 3, the last one
```

Precedence of AND && is higher than OR ||

The precedence of AND && operator is higher than OR ||.

So the code `a && b || c && d` is essentially the same as if the && expressions were in parentheses: `(a && b) || (c && d)`.

Don't replace if with || or &&

Sometimes, people use the AND && operator as a "shorter to write if".

For instance:

```
let x = 1; (x > 0) && alert( 'Greater than zero!' );
```

The action in the right part of && would execute only if the evaluation reaches it. That is, only if `(x > 0)` is true.

So we basically have an analogue for:

```
let x = 1; if (x > 0) alert( 'Greater than zero!' );
```

Although, the variant with && appears shorter, if is more obvious and tends to be a little bit more readable. So we

recommend using every construct for its purpose: use `if` if we want `if` and use `&&` if we want AND.

! (NOT)

The boolean NOT operator is represented with an exclamation sign `!`.

The syntax is pretty simple:

```
result = !value;
```

The operator accepts a single argument and does the following:

14. Converts the operand to boolean type: `true/false`.
15. Returns the inverse value.

For instance:

```
alert( !true ); // falsealert( !0 ); // true
```

A double NOT `!!` is sometimes used for converting a value to boolean type:

```
alert( !!"non-empty string" ); // truealert( !!null ); // false
```

That is, the first NOT converts the value to boolean and returns the inverse, and the second NOT inverts it again. In the end, we have a plain value-to-boolean conversion.

There's a little more verbose way to do the same thing – a built-in Boolean function:

```
alert( Boolean("non-empty string") ); // truealert( Boolean(null) ); // false
```

Q2).

B). Write a C++ program to get Temperature in Fahrenheit F and then find the Atmosphere according to the below rules:

- **If temperature F is above 40 degree Fahrenheit then display.....Very Hot.**

- If temperature F is between 35 & 40 degree Fahrenheit then display.....Tolerable.
- If temperature F is between 30 & 35 degree Fahrenheit then display.....Warm.

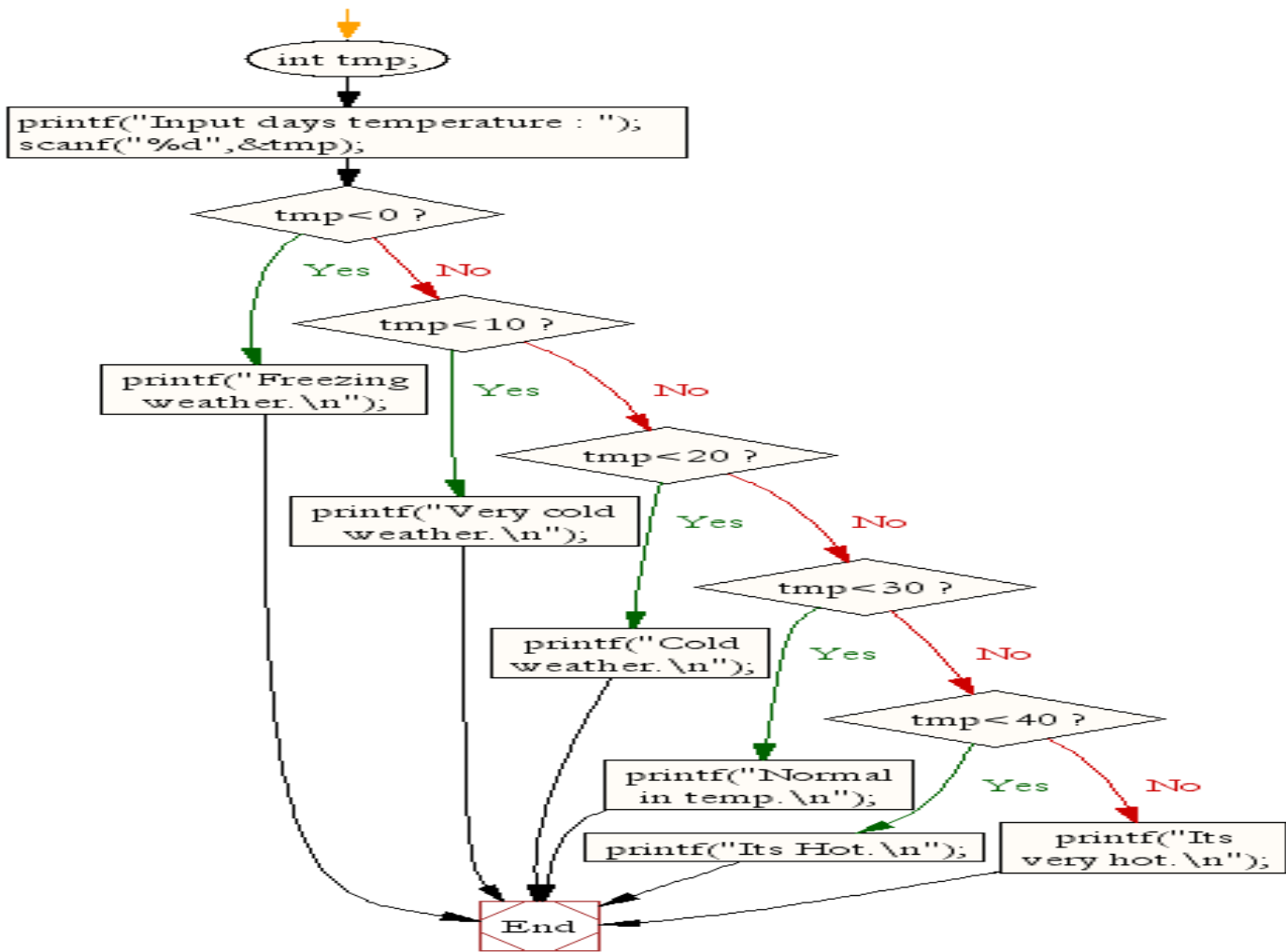
If temperature F is less than 30 degree Fahrenheit then display.....Cool.

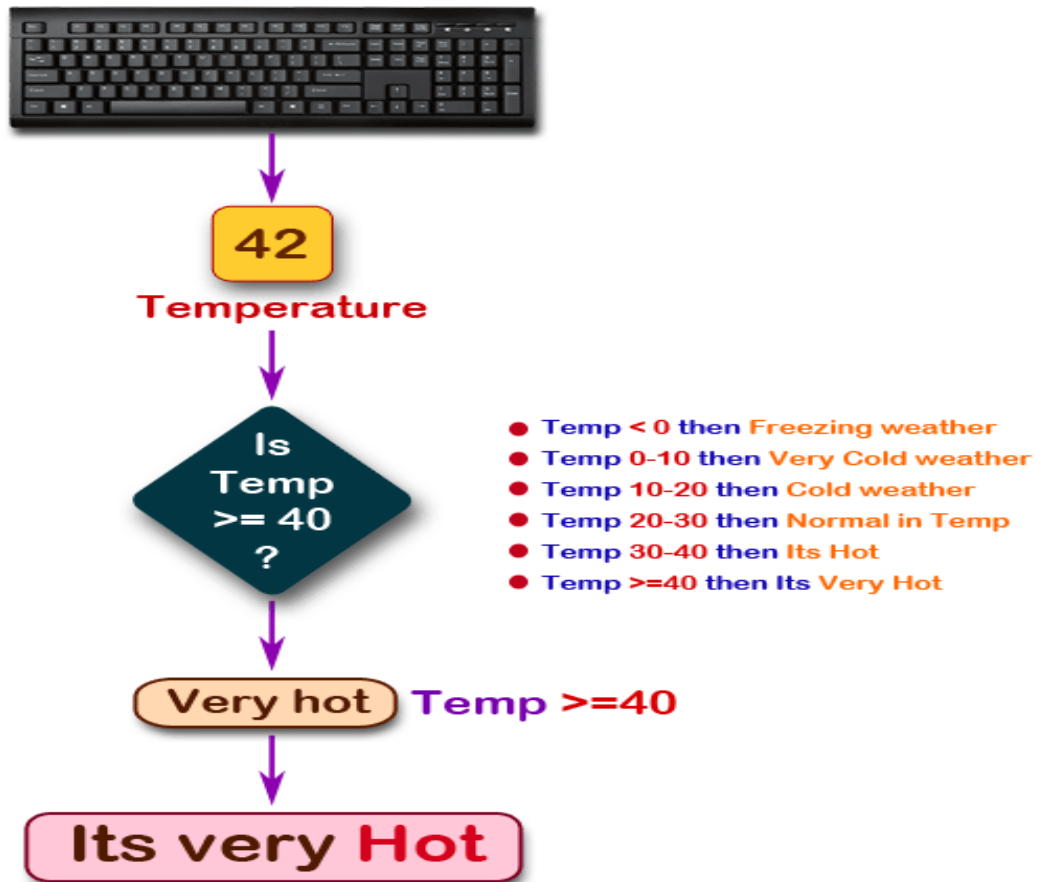
Ans). #include <iostream>
using namespace std;

```
int main()
{
    int f;
    cout<<"Enter temperature in fahrenheit";
    cin>>f;

    if(f > 40)
    {
        cout<<"Very hot. "<<f<<" is the temperature";
    }
    else if(f>=35 && f<=40)
    {
        cout<<"tolerable. "<<f<<" is the temperature";
    }else if(f >=30 && f<35){
        cout<<"warm. "<<f<<" is the temperature";
    }else if(f <30 ){
        cout<<"Cool. "<<f<<" is the temperature";
    }
    return 0;
```

}





© w3resource.com

Q3).

A). What does looping mean? Explain different loops in c++.

Ans). In this tutorial, we will learn about the C++ for loop and its working with the help of some examples.

In computer programming, loops are used to repeat a block of code.

For example, let's say we want to show a message 100 times. Then instead of writing the print statement 100 times, we can use a loop.

That was just a simple example; we can achieve much more efficiency and sophistication in our programs by making effective use of loops.

There are 3 types of loops in C++.

- for loop
- while loop
- do...while loop

This tutorial focuses on C++ for loop. We will learn about the other type of loops in the upcoming tutorials.

C++ for loop

The syntax of for-loop is:

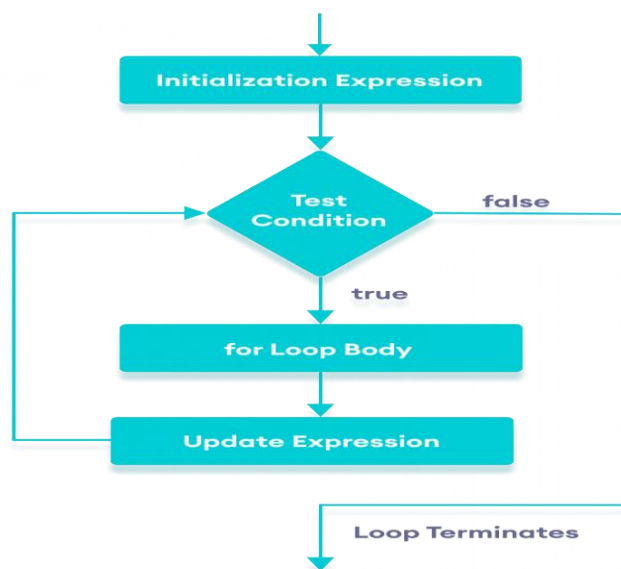
```
for (initialization; condition; update) { // body of-loop }
```

Here,

- **initialization** - initializes variables and is executed only once
- **condition** - if true, the body of for loop is executed if false, the for loop is terminated
- **update** - updates the value of initialized variables and again checks the condition

To learn more about conditions, check out our tutorial on [C++ Relational and Logical Operators](#).

Flowchart of for Loop in C++



Flowchart of for loop in C++

Example 1: Printing Numbers From 1 to 5

```
#include <iostream>using namespace std;int main()
{
    for (int i = 1; i <= 5; ++i) {
        cout << i
        << " ";
    }
    return 0;
}
```

[Run Code](#)

Output

1 2 3 4 5

Here is how this program works

Iteration	Variable	$i \leq 5$	Action
1st	$i = 1$	true	1 is printed. i is increased to 2.
2nd	$i = 2$	true	2 is printed. i is increased to 3.
3rd	$i = 3$	true	3 is printed. i is increased to 4.
4th	$i = 4$	true	4 is printed. i is increased to 5.
5th	$i = 5$	true	5 is printed. i is increased to 6.
6th	$i = 6$	false	The loop is terminated

Example 2: Display a text 5 times

```
// C++ Program to display a text 5 times#include
<iostream>using namespace std;int main() {
    for (int i = 1; i <= 5; ++i) {
        cout << "Hello World! " <<
        endl;
    }
    return 0;
}
```

[Run Code](#)

Output

Hello World!Hello World!Hello World!Hello World!Hello World!

Here is how this program works

Iteration	Variable	$i \leq 5$	Action
1st	$i = 1$	true	Hello World! is printed and i is increased to 2.
2nd	$i = 2$	true	Hello World! is printed and i is increased to 3.

3rd	<code>i = 3</code>	true	Hello World! is printed and i is increased to 4.
4th	<code>i = 4</code>	true	Hello World! is printed and i is increased to 5.
5th	<code>i = 5</code>	true	Hello World! is printed and i is increased to 6.
6th	<code>i = 6</code>	false	The loop is terminated

Example 3: Find the sum of first n Natural Numbers

```
// C++ program to find the sum of first n natural
// positive integers such as 1,2,3,...n are known
// as natural numbers
#include <iostream> using namespace
std; int main() { int num, sum; sum = 0; cout <<
"Enter a positive integer: "; cin >> num; for (int
count = 1; count <= num; ++count) { sum +=
count; } cout << "Sum = " << sum << endl; return
0; }
```

[Run Code](#)

Output

Enter a positive integer: 10 Sum = 55

In the above example, we have two variables num and sum. The sum variable is assigned with 0 and the num variable is assigned with the value provided by the user.

Note that we have used a for loop.

```
for(int count = 1; count <= num; ++count)
```

Here,

- `int count = 1`: initializes the count variable
- `count <= num`: runs the loop as long as count is less than or equal to num
- `++count`: increase the count variable by 1 in each iteration

When count becomes 11, the condition is false and sum will be equal to $0 + 1 + 2 + \dots + 10$.

Ranged Based for Loop

In C++11, a new range-based for loop was introduced to work with collections such as **arrays** and **vectors**. Its syntax is:

```
for (variable : collection) { // body of loop}
```

Here, for every value in the collection, the for loop is executed and the value is assigned to the variable.

Example 4: Range Based for Loop

```
#include <iostream>using namespace std;int main()  
{    int num_array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
for (int n : num_array) {        cout << n << " ";    }  
return 0;}
```

[Run Code](#)

Output

1 2 3 4 5 6 7 8 9 10

In the above program, we have declared and initialized an `int` array named `num_array`. It has 10 items.

Here, we have used a range-based for loop to access all the items in the array.

C++ Infinite for loop

If the condition in a for loop is always true, it runs forever (until memory is full). For example,

```
// infinite for loopfor(int i = 1; i > 0; i++) { //  
block of code}
```

In the above program, the condition is always true which will then run the code for infinite times.

Check out these examples to learn more:

- [C++ Program to Calculate Sum of Natural Numbers](#)
- [C++ Program to Find Factorial](#)

- [C++ Program to Generate Multiplication Table](#)

C++ WHILE AND DO...WHILE LOOP

In this tutorial, we will learn the use of while and do...while loops in C++ programming with the help of some examples.

In computer programming, loops are used to repeat a block of code.

For example, let's say we want to show a message 100 times. Then instead of writing the print statement 100 times, we can use a loop.

That was just a simple example; we can achieve much more efficiency and sophistication in our programs by making effective use of loops.

There are **3** types of loops in C++.

16. for loop
17. while loop
18. do...while loop

In the previous tutorial, we learned about the [C++ for loop](#). Here, we are going to learn about while and do...while loops.

C++ while Loop

The syntax of the while loop is:

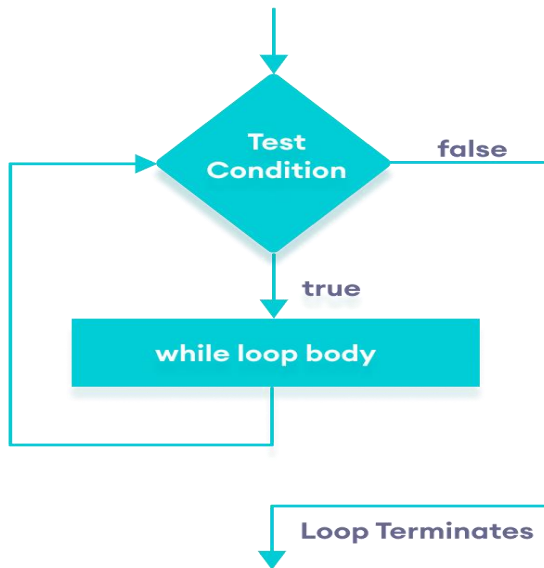
```
while (condition) { // body of the loop }
```

Here,

- A while loop evaluates the condition
- If the condition evaluates to true, the code inside the while loop is executed.
- The condition is evaluated again.
- This process continues until the condition is false.
- When the condition evaluates to false, the loop terminates.

To learn more about the conditions, visit [C++ Relational and Logical Operators](#).

Flowchart of while Loop



Flowchart of C++ while loop

Example 1: Display Numbers from 1 to 5

```
// C++ Program to print numbers from 1 to 5#include <iostream>using namespace std;int main() {    int i = 1;    // while loop from 1 to 5    while (i <= 5) {        cout << i << " ";        ++i;    }    return 0;}
```

[Run Code](#)

Output

1 2 3 4 5

Here is how the program works.

Iteration	Variable	$i \leq 5$	Action
1st	$i = 1$	true	1 is printed and i is increased to 2.
2nd	$i = 2$	true	2 is printed and i is increased to 3.
3rd	$i = 3$	true	3 is printed and i is increased to 4.
4th	$i = 4$	true	4 is printed and i is increased to 5.

5th	i = 5	true	5 is printed and i is increased to 6.
6th	i = 6	false	The loop is terminated

Example 2: Sum of Positive Numbers Only

```
// program to find the sum of positive numbers// if the
user enters a negative number, the loop ends// the
negative number entered is not added to the sum#include
<iostream>using namespace std;int main() {    int number;
int sum = 0;    // take input from the user    cout <<
"Enter a number: ";    cin >> number;    while (number >=
0) {        // add all positive numbers        sum +=
number;        // take input again if the number is
positive        cout << "Enter a number: ";        cin >>
number;    }    // display the sum    cout << "\nThe sum
is " << sum << endl;    return 0;}
```

[Run Code](#)

Output

```
Enter a number: 6Enter a number: 12Enter a number: 7Enter
a number: 0Enter a number: -2The sum is 25
```

In this program, the user is prompted to enter a number, which is stored in the variable number.

In order to store the sum of the numbers, we declare a variable sum and initialize it to the value of 0.

The while loop continues until the user enters a negative number. During each iteration, the number entered by the user is added to the sum variable.

When the user enters a negative number, the loop terminates. Finally, the total sum is displayed.

C++ do...while Loop

The `do...while` loop is a variant of the `while` loop with one important difference: the body of `do...while` loop is executed once before the condition is checked.

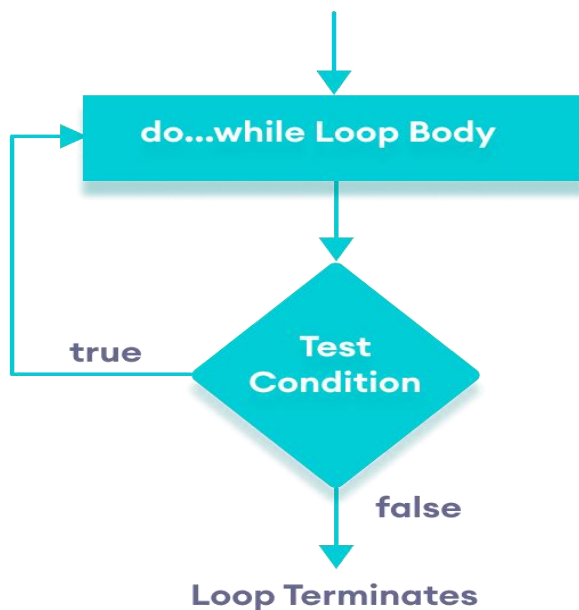
Its syntax is:

```
do { // body of loop;}while (condition);
```

Here,

- The body of the loop is executed at first. Then the condition is evaluated.
- If the condition evaluates to `true`, the body of the loop inside the `do` statement is executed again.
- The condition is evaluated once again.
- If the condition evaluates to `true`, the body of the loop inside the `do` statement is executed again.
- This process continues until the condition evaluates to `false`. Then the loop stops.

Flowchart of do...while Loop



Flowchart of C++ do...while loop

Example 3: Display Numbers from 1 to 5

```
// C++ Program to print numbers from 1 to 5#include<iostream>using namespace std;int main() { int i = 1;
```

```
// do...while loop from 1 to 5 do { cout << i <<
"; ++i; } while (i <= 5); return 0;}
```

[Run Code](#)

Output

1 2 3 4 5

Here is how the program works.

Iteration	Variable	$i \leq 5$	Action
	$i = 1$	not checked	1 is printed and i is increased to 2
1st	$i = 2$	true	2 is printed and i is increased to 3
2nd	$i = 3$	true	3 is printed and i is increased to 4
3rd	$i = 4$	true	4 is printed and i is increased to 5
4th	$i = 5$	true	5 is printed and i is increased to 6
5th	$i = 6$	false	The loop is terminated

Example 4: Sum of Positive Numbers Only

```
// program to find the sum of positive numbers// If the
user enters a negative number, the loop ends// the
negative number entered is not added to the sum#include
<iostream>using namespace std;int main() { int number =
0; int sum = 0; do { sum += number; //
take input from the user cout << "Enter a number:
"; cin >> number; } while (number >= 0);
// display the sum cout << "\nThe sum is " << sum <<
endl; return 0;}
```

[Run Code](#)

Output 1

Enter a number: 6Enter a number: 12Enter a number: 7Enter a number: 0Enter a number: -2The sum is 25

Here, the `do...while` loop continues until the user enters a negative number. When the number is negative, the loop terminates; the negative number is not added to the `sum` variable.

Output 2

Enter a number: -6The sum is 0.

The body of the `do...while` loop runs only once if the user enters a negative number.

Infinite while loop

If the `condition` of a loop is always `true`, the loop runs for infinite times (until the memory is full). For example,

```
// infinite while loopwhile(true) { // body of the loop}
```

Here is an example of an infinite `do...while` loop.

```
// infinite do...while loopint count = 1;do { // body of loop} while(count == 1);
```

In the above programs, the `condition` is always `true`. Hence, the loop body will run for infinite times.

for vs while loops

A `for` loop is usually used when the number of iterations is known. For example,

```
// This loop is iterated 5 timesfor (int i = 1; i <=5; ++i) { // body of the loop}
```

Here, we know that the `for`-loop will be executed 5 times.

However, `while` and `do...while` loops are usually used when the number of iterations is unknown. For example,

```
while (condition) { // body of the loop}
```

Check out these examples to learn more:

- [C++ Program to Display Fibonacci Series](#)
- [C++ Program to Find GCD](#)
- [C++ Program to Find LCM](#)

C++ BREAK STATEMENT

In this tutorial, we will learn about the break statement and its working in loops with the help of examples.

In computer programming, the break statement is used to terminate the loop in which it is used.

The syntax of the break statement is:

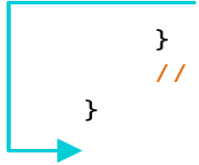
break;

Before you learn about the break statement, make sure you know about:

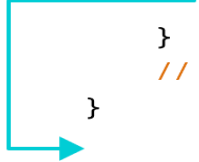
- [C++ for loop](#)
- [C++ if...else](#)
- [C++ while loop](#)

Working of C++ break
Statement

```
for (init; condition; update) {
    // code
    if (condition to break) {
        break;
    }
    // code
}
```



```
while (condition) {
    // code
    if (condition to break) {
        break;
    }
    // code
}
```



Working of break statement in C++

Example 1: break with for loop

```
// program to print the value of i
#include <iostream> using namespace std;
int main() {
    for (int i = 1; i <= 5; i++) {
        // break condition
        if (i == 3)
            break;
        cout << i << endl;
    }
    return 0;
}
```

[Run Code](#)

Output

12

In the above program, the for loop is used to print the value of `i` in each iteration. Here, notice the code:

```
if (i == 3) { break; }
```

This means, when `i` is equal to **3**, the break statement terminates the loop. Hence, the output doesn't include values greater than or equal to 3.

Note: The break statement is usually used with decision-making statements.

Example 2: break with while loop

```
// program to find the sum of positive numbers// if the user enters a negative numbers, break ends the loop// the negative number entered is not added to sum#include <iostream>using namespace std;int main() {    int number;    int sum = 0;    while (true) {        // take input from the user        cout << "Enter a number: ";        cin >> number;        // break condition        if (number < 0)        {            break;        }        // add all positive numbers        sum += number;    }    // display the sum    cout << "The sum is " << sum << endl;    return 0;}
```

[Run Code](#)

Output

```
Enter a number: 1Enter a number: 2Enter a number: 3Enter a number: -5The sum is 6.
```

In the above program, the user enters a number. The while loop is used to print the total sum of numbers entered by the user. Here, notice the code,

```
if(number < 0) {    break;}
```

This means, when the user enters a negative number, the break statement terminates the loop and codes outside the loop are executed.

The while loop continues until the user enters a negative number.

break with Nested loop

When break is used with nested loops, break terminates the inner loop. For example,

```
// using break statement inside// nested for loop#include <iostream>using namespace std;int main() {    int number;    int sum = 0;    // nested for loops    // first loop    for (int i = 1; i <= 3; i++) {        // second loop        for (int j = 1; j <= 3; j++) {            if (i == 2)                cout << "i = " << i << ", j = " << j << endl;            break;        }    }    return 0;}
```

Run Code

Output

$i = 1, j = 1$
 $i = 1, j = 2$
 $i = 1, j = 3$
 $i = 2, j = 1$
 $i = 2, j = 2$
 $i = 2, j = 3$
 $i = 3, j = 1$
 $i = 3, j = 2$
 $i = 3, j = 3$

In the above program, the break statement is executed when $i == 2$. It terminates the inner loop, and the control flow of the program moves to the outer loop.

Hence, the value of $i = 2$ is never displayed in the output.

Q3).

B). Write a C++ Program to read a number from keyboard and then determine whether it is even or odd number?

```
Ans). #include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "Enter an integer:
";    cin >> n;
    if ( n % 2 == 0)
        cout << n << " is even.";
    Else
        cout << n << " is odd.";
    return 0;
}
```

Q4).

A). What is the purpose of using *break* and *continue* statements?

Ans). The major difference between break and continue statements in C language is that a break causes the innermost enclosing loop or switch

to be exited immediately. Whereas, the continue statement causes the next iteration of the enclosing for, while, or do loop to begin. The continue statement in while and do loops takes the control to the loop's *test-condition* immediately, whereas in the for loop it takes the control to the *increment* step of the loop.

The continue statement applies only to loops, not to switch. A continue inside a switch inside a loop causes the next loop iteration.

Practically, break is used in switch, when we want to exit after a particular case is executed; and in loops, when it becomes desirable to leave the loop as soon as a certain condition occurs (for instance, you detect an error condition, or you reach the end of your data prematurely). The continue statement is used when we want to skip one or more statements in loop's body and to transfer the control to the next iteration.

Difference Between break and continue

Difference Between break and continue

break	continue
A break can appear in both switch and loop (for, while, do) statements.	A continue can appear only in loop (for, while, do) statements.
A break causes the switch or loop statements to terminate the moment it is executed. Loop or switch ends abruptly when break is encountered.	A continue doesn't terminate the loop, it causes the loop to go to the next iteration. All iterations of the loop are executed even if continue is encountered. The continue statement is used to skip statements in the loop that appear after the continue.

The break statement can be used in both switch and loop statements.	The continue statement can appear only in loops. You will get an error if this appears in switch statement.
When a break statement is encountered, it terminates the block and gets the control out of the switch or loop.	When a continue statement is encountered, it gets the control to the next iteration of the loop.
A break causes the innermost enclosing loop or switch to be exited immediately.	A continue inside a loop nested within a switch causes the next loop iteration.

Similarities Between break and continue

Both break and continue statements in C programming language have been provided to alter the normal flow of program.

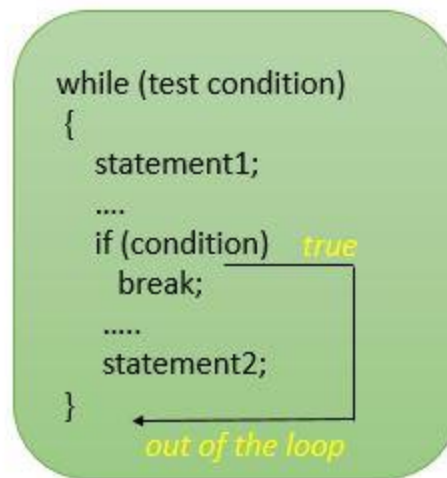
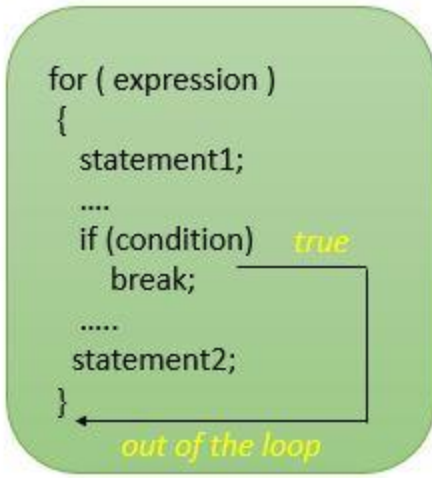
Example using break

The following function, trim, removes trailing blanks, tabs and newlines from the end of a string, using a break to exit from a loop when the rightmost non-blank, non-tab, non-newline is found.

```
/* trim: remove trailing blanks, tabs, newlines */int
trim(char s[]){ int n; for (n = strlen(s)-1; n >= 0; n-
-) if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
break; s[n+1] = '\0'; return n;}
```

strlen returns the length of the string. The for loop starts at the end and scans backwards looking for the first character that is not a blank or tab or newline. The loop is broken when one is found, or when n becomes negative (that is, when the entire string has been scanned).

How does **break** statement works?



Example using continue

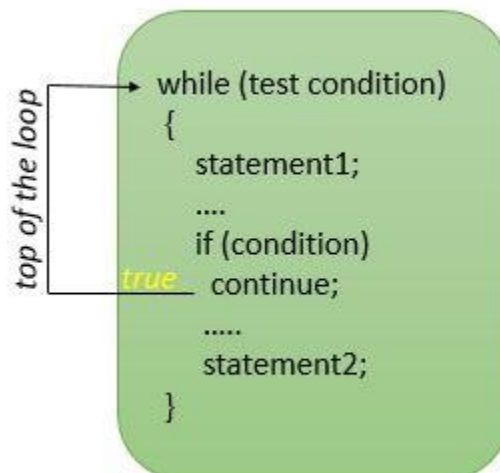
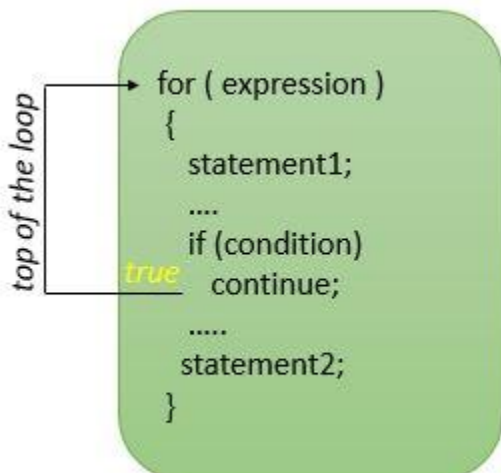
As an example, the following piece of code sums up the non-negative elements in the array a; negative values are skipped.

```

/* sum up non-negative elements of an array */ #include
<stdio.h> int main(){ int a[10] = {-1, 2, -3, 4, -5, 6, -
7, 8, -9, 10}; int i, sum = 0; for (i = 0; i < 10; i++)
{ if (a[i] < 0) /* skip negative elements */ continue;
sum += a[i]; /* sum positive elements */ } printf("Sum
of positive elements: %d\n", sum);} OUTPUT====Sum of
positive elements: 30

```

How continue statement work?



Q4).

B). Write a C++ program to find the sum of the following numbers:

1+2+3+.....+10

Ans). `#include <iostream>`

`using namespace std;`

`int main()`

`{`

`int i, sum=0;`

`cout << "\n\n Find the first 10 natural numbers:\n";`

`cout << "-----\n";`

`cout << " The natural numbers are: \n";`

`for (i = 1; i <= 10; i++)`

`{`

`cout << i << " ";`

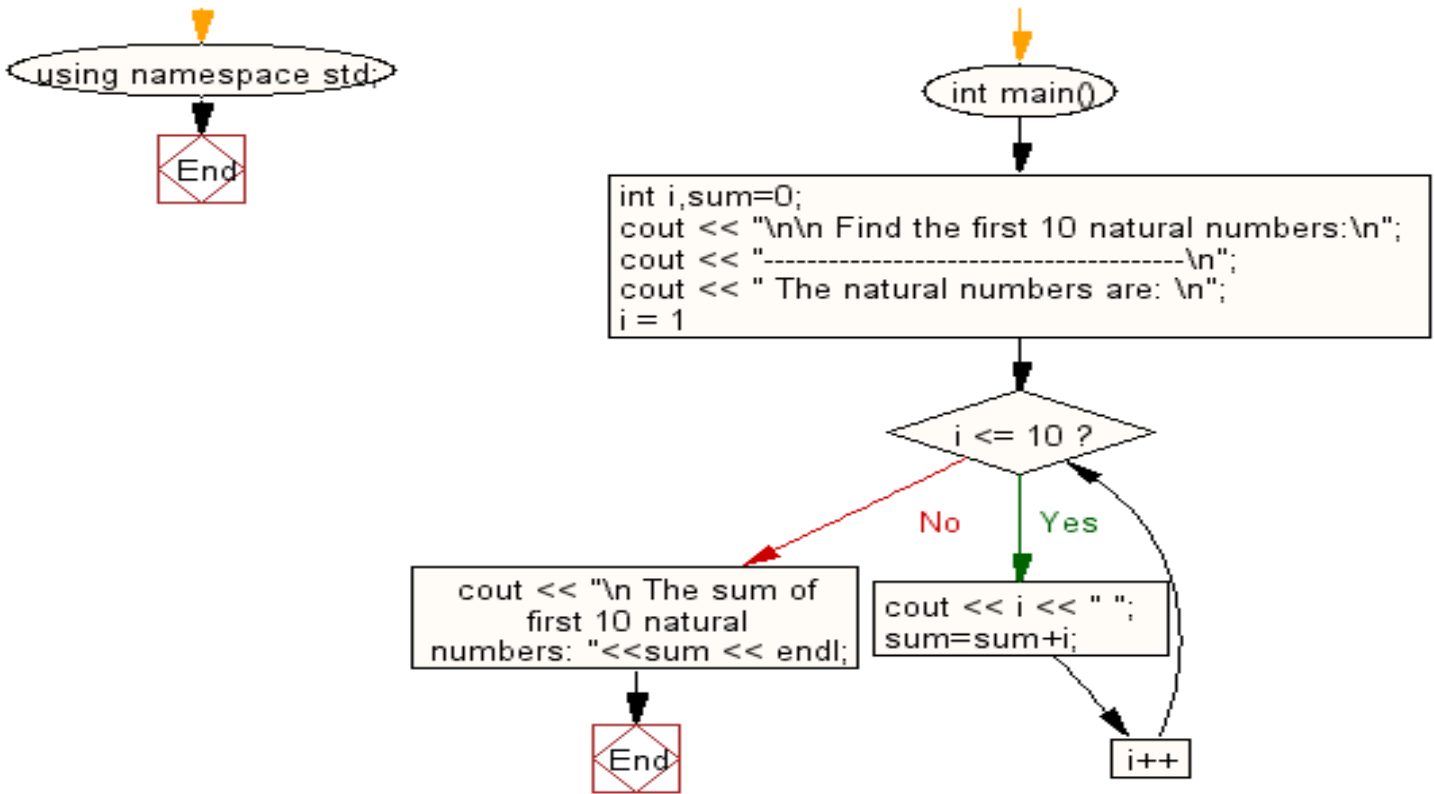
`sum=sum+i;`

`}`

`cout << "\n The sum of first 10 natural numbers: "<<sum`

`<< endl;`

`}`



Q5). What is an array? Explain on-Dimensional and Two-dimensional Arrays with examples.

Ans). What is an Array?

An array is a collection of one or more values of the same type. Each value is called an element of the array. The elements of the array share the same variable name but each element has its own unique index number (also known as a subscript). An array can be of any type, For example: **int**, **float**, **char** etc. If an array is of type **int** then its elements must be of type **int** only.

To store roll no. of **100** students, we have to declare an array of size **100** i.e **roll_no[100]**. Here size of the array is **100** , so it is capable of storing **100** values. In C, index or subscript starts from **0**, so **roll_no[0]** is the first element, **roll_no[1]** is the second element and so on. Note that the last element of the array will be at **roll_no[99]** not at **roll_no[100]** because the index starts at **0**.

`roll_no[0]`



array variable

subscript or index

Arrays can be single or multidimensional. The number of subscript or index determines the dimensions of the array. An array of one dimension is known as a one-dimensional array or 1-D array, while an array of two dimensions is known as a two-dimensional array or 2-D array.

Let's start with a one-dimensional array.

One-dimensional array

Conceptually you can think of a one-dimensional array as a row, where elements are stored one after another.



an array of 6 elements

TheCguru.com

Syntax: `datatype array_name[size];`

datatype: It denotes the type of the elements in the array.

array_name: Name of the array. It must be a valid identifier.

size: Number of elements an array can hold.

here are some example of array declarations:

1	<code>int num[100];</code>
2	<code>float temp[20];</code>
3	<code>char ch[50];</code>

`num` is an array of type `int`, which can only store `100` elements of type `int`.

`temp` is an array of type `float`, which can only store `20` elements of type `float`.

`ch` is an array of type `char`, which can only store `50` elements of type `char`.

Note: When an array is declared it contains garbage values.

The individual elements in the array:

1	<code>num[0], num[1], num[2],</code>
2	<code>num[99]</code>
3	<code>temp[0], temp[1], temp[2],</code>
	<code>temp[19]</code>
	<code>ch[0], ch[1], ch[2],, ch[49]</code>

We can also use variables and symbolic constants to specify the size of the array.

1	<code>#define SIZE 10</code>
2	
3	<code>int main()</code>
4	<code>{</code>
5	<code>int size = 10;</code>
6	
7	<code>int my_arr1 [SIZE]; // ok</code>
8	<code>int my_arr2 [size]; // not</code>
9	<code>allowed until C99</code>
10	<code>// ...</code>
	<code>}</code>

Note: Until C99 standard, we were not allowed to use variables to specify the size of the array. If you are using a compiler which supports C99 standard, the above code would compile successfully. However, If you're using an older version of C compiler like Turbo C++, then you will get an error.

The use of symbolic constants makes the program maintainable, because later if you want to change the size of the array you need to modify it at once place only i.e in the `#define` directive.

Accessing elements of an
array

The elements of an array can be accessed by specifying array name followed by subscript or index inside square brackets (i.e `[]`). Array subscript or index starts at `0`. If the size of an array is `10` then the first element is at index `0`, while the last element is at index `9`. The first valid subscript (i.e `0`) is known as the *lower bound*, while last valid subscript is known as the *upper bound*.

```
int my_arr[5];
```

then elements of this array are;

First element – `my_arr[0]`

Second element – `my_arr[1]`

Third element – `my_arr[2]`

Fourth element – `my_arr[3]`

Fifth element – `my_arr[4]`

Array subscript or index can be any expression that yields an integer value. For example:

```
int i = 0, j = 2;
my_arr[i]; // 1st element
my_arr[i+1]; // 2nd element
my_arr[i+j]; // 3rd element
```

In the array `my_arr`, the last element is at `my_arr[4]`, What if you try to access elements beyond the last valid index of the array?

```
printf("%d", my_arr[5]); // 6th
element
printf("%d", my_arr[10]); // 11th
element
printf("%d", my_arr[-1]); //
element just before 0
```

Sure indexes `5`, `10` and `-1` are not valid but C compiler will not show any error message instead some garbage value will be printed. The C language doesn't check bounds of the array. It is the responsibility of the programmer to check array bounds whenever required.

Processing 1-D arrays

The following program uses for loop to take input and print elements of a 1-D array.

```

1      #include<stdio.h>
2
3      int main()
4      {
5          int arr[5], i;
6
7          for(i = 0; i < 5; i++)
8          {
9              printf("Enter a[%d]: ", i);
10             scanf("%d", &arr[i]);
11         }
12
13         printf("\nPrinting elements of
14         the array: \n\n");
15
16         for(i = 0; i < 5; i++)
17         {
18             printf("%d ", arr[i]);
19         }
20
21         // signal to operating system
22         program ran fine
         return 0;
     }

```

Expected Output:

```

1
2
3
4
5
6
7
8
9

```

How it works:

In Line 5, we have declared an array of 5 integers and variable **i** of type **int**. Then a for loop is used to enter five elements into an array. In **scanf()** we have used **&** operator (also known as the address of operator) on element **arr[i]** of an array, just like we had done with variables of type **int**, **float**, **char** etc. Line 13

prints "Printing elements of the array" to the console. The second for loop prints all the elements of an array one by one. The following program prints the sum of elements of an array.

```
1 #include<stdio.h>
2
3 int main()
4 {
5     int arr[5], i, s = 0;
6
7     for(i = 0; i < 5; i++)
8     {
9         printf("Enter a[%d]: ", i);
10        scanf("%d", &arr[i]);
11    }
12
13    for(i = 0; i < 5; i++)
14    {
15        s += arr[i];
16    }
17
18    printf("\nSum of elements
19 = %d ", s);
20
21    // signal to operating system
22    program ran fine
    return 0;
}
```

Expected Output:

```
1
2
3
4
5
6
7
```

How it works:

The first for loop asks the user to enter five elements into the array. The second for loop reads all the elements of an array one by one and accumulate the sum of all the elements in the variable **s**. Note that it is necessary to initialize the variable **s** to **0**,

otherwise, we will get the wrong answer because of the garbage value of `s`.

Initializing Array

When an array is declared inside a function the elements of the array have garbage value. If an array is global or static, then its elements are automatically initialized to `0`. We can explicitly initialize elements of an array at the time of declaration using the following syntax:

Syntax: `datatype array_name[size] = { val1, val2, val3, valN };`

`datatype` is the type of elements of an array.

`array_name` is the variable name, which must be any valid identifier.

`size` is the size of the array.

`val1, val2 ...` are the constants known as initializers. Each value is separated by a comma(,) and then there is a semi-colon (;) after the closing curly brace (}).

Here is are some examples:

1	<code>float temp[5] = {12.3, 4.1, 3.8,</code>
2	<code>9.5, 4.5}; // an array of 5 floats</code>
3	<code>int arr[9] = {11, 22, 33, 44, 55, 66,</code>
	<code>77, 88, 99}; // an array of 9 ints</code>

While initializing 1-D array it is optional to specify the size of the array, so you can also write the above statements as:

1	<code>float temp[] = {12.3, 4.1, 3.8, 9.5,</code>
2	<code>4.5}; // an array of 5 floats</code>
3	<code>int arr[] = {11, 22, 33, 44, 55, 66,</code>
	<code>77, 88, 99}; // an array of 9 ints</code>

If the number of initializers is less than the specified size then the remaining elements of the array are assigned a value of `0`.

1	<code>float temp[5] = {12.3, 4.1};</code>
---	---

here the size of `temp` array is 5 but there are only two initializers. After this initialization the elements of the array are as follows:

`temp[0]` is 12.3

`temp[1]` is 4.1

`temp[2]` is 0

`temp[3]` is 0

`temp[4]` is 0

If the number of initializers is greater than the size of the array then, the compiler will report an error. For example:

```
1 int num[5] = {1, 2, 3, 4, 5, 6, 7, 8}
// error
```

The following program finds the highest and lowest elements in an array.

```
1 #include<stdio.h>
2 #define SIZE 10
3
4 int main()
5 {
6     int my_arr[SIZE] =
7     {34,56,78,15,43,71,89,34,70,91};
8     int i, max, min;
9
10    max = min = my_arr[0];
11
12    for(i = 0; i < SIZE; i++)
13    {
14        // if value of current
15        element is greater than
16        previous value
17        // then assign new value
18        to max
19        if(my_arr[i] > max)
20        {
21            max = my_arr[i];
22        }
23
24        // if the value of current
25        element is less than previous
26        element
27        // then assign new value
        to min
```

```
28
29
30
31
32
33
```

```
    if(my_arr[i] < min)
    {
        min = my_arr[i];
    }
}

printf("Lowest value = %d\n",
min);
printf("Highest value = %d",
max);

// signal to operating system
everything works fine
return 0;
}
```

Expected Output:

```
1
2
```

How it works:

In line 6, first, we have declared and initialized an array of 10 integers. In the next line, we have declared three more variables of type `int` namely: `i`, `max` and `min`. In line 9, we have assigned the value of the first element of `my_arr` to `max` and `min`. A for loop is used to iterate through all the elements of an array. Inside the for loop, the first if condition (`my_arr[i] > max`) checks whether the current element is greater than `max`, if it is, we assign the value of the current element to `max`.

The second if statement checks whether the value of the current element is smaller than the value of `min`. If it is, we assign the value of the current element to `min`. This process continues until there are elements in the array left to iterate.

When the process is finished, `max` and `min` variables will have maximum and minimum values respectively.

Passing 1-D array elements to
a function

We can pass elements of 1-D array just like any normal variables. The following example demonstrates the same.

```
1 #include<stdio.h>
2 void odd_or_even(int a);
3
4 int main()
5 {
6     int my_arr[] = {13,56,71,38,93},
7     i;
8
9     for(i = 0; i < 5; i++)
10    {
11        // passing one element at
12        a time to odd_or_even()
13        function
14        odd_or_even(my_arr[i]);
15    }
16
17    // signal to operating system
18    program ran fine
19    return 0;
20 }
21
22 void odd_or_even(int a)
23 {
24     if(a % 2 == 0)
25     {
26         printf("%d is even\n", a);
27     }
28
29     else
30     {
31         printf("%d is odd\n", a);
32     }
33 }
```

Expected Output:

```
1
2
3
4
5
```

Passing the whole Array to a
Function

Just like normal variables you can pass an array variable to a function. But before you do so, make sure the formal arguments is declared as an array variable of same data type. For example:

```
1 int main()  
2 {  
3  
4     int a[10];  
5  
6     function_1(a);  
7  
8     return 0;  
9 }  
10  
11 void function_1(int arr[10])  
12 {  
13  
14     statement ;  
15     ...  
16 }
```

Here we are passing an array of **10** integers to **function_1()**, that's why the formal argument of **function_1()** is also declared as an array of **10** integers.

It is optional to specify the size of the array in the formal arguments. This means you can also declare formal argument of **function_1()** as follows:

```
1 void function_1(int arr[])  
2 {  
3  
4     statement ;  
5     ...  
6 }
```

While learning about formal and actual arguments, we have learned that changes made in the formal arguments do not affect the actual arguments. This is not the case with arrays. When an array is passed as an actual argument, the function gets access to the original array, so any changes made inside the function will affect the original array.

```
1 #include<stdio.h>  
2 void new_array(int a[]);
```

3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38

```
int main()
{
    int my_arr[] = {13,56,71,38,93},
    i;

    printf("Original array: \n\n");

    for(i = 0; i < 5; i++)
    {
        printf("%d ", my_arr[i]);
    }

    new_array(my_arr);

    printf("\n\nModified array :
\n\n");

    for(i = 0; i < 5; i++)
    {
        printf("%d ", my_arr[i]);
    }

    // signal to operating system
    program ran fine
    return 0;
}

void new_array(int a[])
{
    int i;

    // multiply original elements
    by 2

    for(i = 0; i < 5; i++)
    {
        a[i] = 2 * a[i];
    }
}
```

Expected Output:

1
2
3

```
4  
5  
6  
7
```

How it works:

The first for loop in `main()` function prints the initial values of the elements of an array. In line 15, `new_array()` function is called with an actual argument of `my_arr`. The Control is transferred to function `new_array()`. The function multiplies each element of the array by 2 and assigns back this new value to the current index. Since `new_array()` is working on the original array, not on a copy of the original array, any changes made by `new_array()` function affect the original array. When the function finishes, control again passes back to `main()` function, where second for loop prints the elements of the array.

Two-dimensional Array

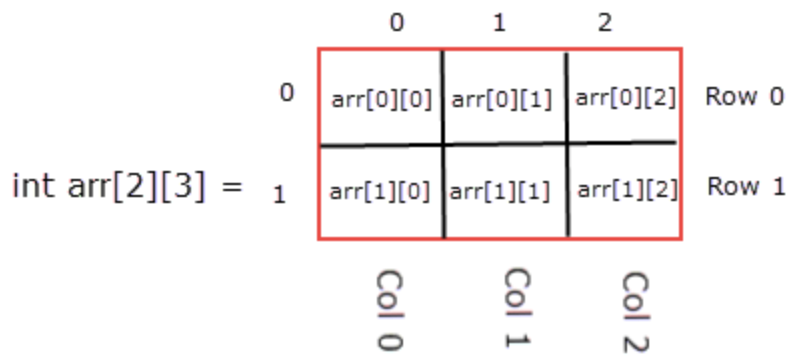
The syntax declaration of 2-D array is not much different from 1-D array. In 2-D array, to declare and access elements of a 2-D array we use 2 subscripts instead of 1.

Syntax: `datatype array_name[ROW][COL];`

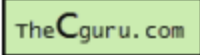
The total number of elements in a 2-D array is `ROW*COL`. Let's take an example.

```
| int arr[2][3];
```

This array can store `2*3=6` elements. You can visualize this 2-D array as a matrix of 2 rows and 3 columns.



A conceptual representation of 2-D array



The individual elements of the above array can be accessed by using two subscript instead of one. The first subscript denotes row number and second denotes column number. As we can see in the above image both rows and columns are indexed from 0. So the first element of this array is at `arr[0][0]` and the last element is at `arr[1][2]`. Here are how you can access all the other elements:

- `arr[0][0]` – refers to the first element
- `arr[0][1]` – refers to the second element
- `arr[0][2]` – refers to the third element
- `arr[1][0]` – refers to the fourth element
- `arr[1][1]` – refers to the fifth element
- `arr[1][2]` – refers to the sixth element

If you try to access an element beyond valid **ROW** and **COL**, C compiler will not display any kind of error message, instead, a garbage value will be printed. It is the responsibility of the programmer to handle the bounds.

`arr[1][3]` – a garbage value will be printed, because the last valid index of **COL** is 2

`arr[2][3]` – a garbage value will be printed, because the last valid index of **ROW** and **COL** is 1 and 2 respectively

Just like 1-D arrays, we can only also use constants and symbolic constants to specify the size of a 2-D array.

```

1 | #define ROW 2
```

2	#define COL 3
3	
4	int i = 4, j = 6;
5	int arr[ROW][COL]; // OK
6	int new_arr[i][j]; // ERROR

Processing elements of a 2-D
array

To process elements of a 2-D array, we use two nested loop. The outer for loop to loop through all the rows and inner for loop to loop through all the columns. The following program will clear everything.

1	#include<stdio.h>
2	#define ROW 3
3	#define COL 4
4	
5	int main()
6	{
7	int arr[ROW][COL], i, j;
8	
9	for(i = 0; i < ROW; i++)
10	{
11	for(j = 0; j < COL; j++)
12	{
13	printf("Enter arr[%d][%d]:", i, j);
14	scanf("%d", &arr[i][j]);
15	}
16	}
17	
18	printf("\nEntered 2-D array is:\n\n");
19	
20	
21	
22	for(i = 0; i < ROW; i++)
23	{
24	for(j = 0; j < COL; j++)
25	{
26	printf("%3d ", arr[i][j]);
27	}
28	printf("\n");
29	}

```
30
```

```
// signal to operating system  
everything works fine  
return 0;  
}
```

Expected Output:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18
```

How it works:

There is nothing new in this previous program that deserves any explanation. We are just using two nested for loops. The first nested for loop takes input from the user. And the second for loop prints the elements of a 2-D array like a matrix.

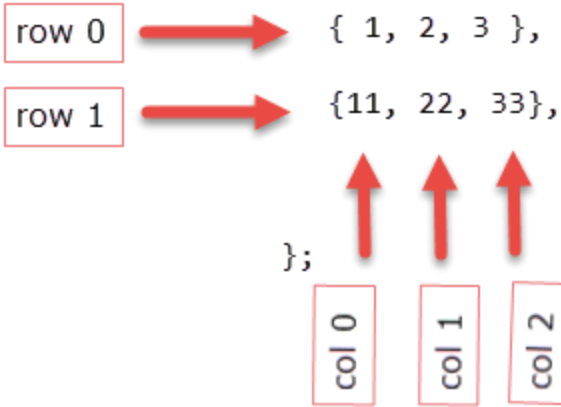
Initializing 2-D array

Initialization of 2-D array is similar to a 1-D array. For e.g:

```
1  
2  
3  
4
```

```
int temp[2][3] = {  
    { 1, 2, 3 }, // row 0  
    {11, 22, 33} // row 1  
};
```

```
int temp[2][3] = {
```



After this initialization, each element is as follows:

1	
2	
3	
4	
5	
6	

Consider another initialization.

1	<pre>int my_arr[4][3] = {</pre>
2	<pre> {10},</pre>
3	<pre> {77, 92},</pre>
4	<pre> {33, 89, 44},</pre>
5	<pre> {12, 11}</pre>
6	<pre>};</pre>

The size of `my_arr` is $4*3=12$, but in the initialization, we have only specified the value of 8 elements. In such cases, the remaining elements will be given the value of 0.

The individual elements are as follows:

1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

```
11
12
13
14
15
```

In 2-D arrays, it is optional to specify the first dimension but the second dimension must always be present. This works only when you are declaring and initializing the array at the same time. For example:

```
1 int two_d[][3] = {
2     {13,23,34},
3     {15,27,35}
4     };
```

is same as

```
1 int two_d[2][3] = {
2     {13, 23, 34},
3     {15, 27, 35}
4     };
```

As discussed earlier you can visualize a 2-D array as a matrix. The following program demonstrates the addition of two matrices.

```
1 #include<stdio.h>
2 #define ROW 2
3 #define COL 3
4
5 int main()
6 {
7     int mat1[ROW][COL],
8     mat2[ROW][COL],
9     mat3[ROW][COL];
10    int i, j;
11
12    printf("Enter first matrix:
13    \n\n");
14
15    for(i = 0; i < ROW; i++)
16    {
17        for(j = 0; j < COL; j++)
18        {
19            printf("Enter a[%d][%d]: ",
20            i, j);
21            scanf("%d", &mat1[i][j]);
22        }
23    }
24 }
```


23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57

```
}  
  
printf("\nEnter Second matrix:  
\n\n");  
  
for(i = 0; i < ROW; i++)  
{  
    for(j = 0; j < COL; j++)  
    {  
        printf("Enter a[%d][%d]: ",  
i, j);  
        scanf("%d", &mat2[i][j]);  
    }  
}  
  
// add mat1 and mat2  
  
for(i = 0; i < ROW; i++)  
{  
    for(j = 0; j < COL; j++)  
    {  
        mat3[i][j] = mat1[i][j] +  
mat2[i][j] ;  
    }  
}  
  
printf("\nResultant array:  
\n\n");  
  
// print resultant array  
  
for(i = 0; i < ROW; i++)  
{  
    for(j = 0; j < COL; j++)  
    {  
        printf("%5d ", mat3[i][j]);  
    }  
    printf("\n");  
}  
  
// signal to operating system  
program ran fine  
return 0;  
}
```

Expected Output:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```

How it works:

Two matrices can be added or subtracted, only if they have the same dimension. In other words, a matrix of size 2×3 can be added to another matrix of 2×3 , but you can't add or subtract it to a matrix of 2×4 or 3×2 . The resultant array will be a matrix of the same dimension as the original two. First two for loops asks the user to enter two matrices. The third for loop adds corresponding elements of `mat1` and `mat2` in a new array `mat3`. Fourth for loop prints the elements of array `mat3`.

Arrays of more than two
dimension

You can even create an array of 3 or more dimensions or more, but generally, you will never need to do so. Therefore, we will restrict ourself to 3-D arrays only.

Here is how you can declare an array of 3 dimensions.

```
int arr[2][3][2];
```

3-D array uses three indexes or subscript. This array can store $2*3*2=12$ elements.

Here is how to initialize a 3-D array.

```
int three_d[2][3][4] = {  
    {12,34,56,12},  
    {57,44,62,14},  
    {64,36,91,16},  
},  
{  
    {87,11,42,82},  
    {93,44,12,99},  
    {96,34,33,26},  
};
```

You can think of this array as 2 2-D arrays and each of these 2-D array has 3 rows and 4 columns;

Here are individual elements of the array:

First Row

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15
```

Second Row

1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

Passing Multidimensional Arrays to Functions

You can pass multi-dimensional arrays to functions just like a 1-D array, but you need to specify the size of the all other dimensions except the first one. For e.g:

If you need to pass `arr[2][3]` to a function called `func_1()`, then you need to declare the `func_1()` like this:

1	<code>void func_1(int my_arr[2][3]) //</code>
2	<code>OK</code>
3	<code>{</code>
4	<code> //...</code>
	<code>}</code>

or like this:

1	<code>void func_1(int my_arr[][3]) //</code>
2	<code>OK</code>
3	<code>{</code>
4	<code> //...</code>
	<code>}</code>

It would be invalid to declare formal argument as follows:

1	<code>void func_1(int my_arr[][]) //</code>
2	<code>error</code>
3	<code>{</code>
4	<code> //...</code>

```
}  
}
```

Similarly to pass a 3-D array you need to declare the function as follows:

```
1 int arr[2][3][4];  
2  
3 void func_1(int my_arr[][3][4])  
4 {  
5     //...  
6 }
```