Decision Making in C++

Methods of decision making in C++
C++ decision making can be done via the following –

- if statement
- if-else statements
- if-else-if ladder
- nested if statements
- switch statements
- Jump Statements:
    - break
    - continue
    - goto
    - return

1. If statement
It’s one of the most basic statements. We test if a given condition is true. If it is then we implement some lines of code. Following is the syntax for the same.

```
if(condition)
{
   // Following statement(s) will be executed
   // if condition is true
}
```

Example

```
#include<iostream>
using namespace std;

int main()
{
    int a = 20, b = 10;

    if (a > b)
    {
        cout << "a is greater than b" << endl;
    }

    cout << "Outside if loop now";
     return 0;
}
```

2. if-else Statement

- If a given condition is true we implement some lines of code.
- If a given condition was not true we implement some other lines of code

Following is the syntax for the same.

```
if(condition) {
    // Following statement(s) will be executed
    // if condition is true
}
else{
    // Following statement(s) are executed
    // if condition was false
}
```

Example

```
#include<iostream>
using namespace std;

int main()
{
    int a = 20;

    if (a > 100)
    {
        cout << "a is greater than 100" << endl;
    }
    else{
        cout << "a is smaller than 100" << endl;
    }
   return 0;
}
```

3. If-else-if Ladder

We can have multiple conditions checking to do this we use if-else-if ladder. Following is the syntax for the same –

```
if(condition){
    // execute these statements if true
}
else if(condition){
    // execute these statements if true
}
else if(condition){
    // execute these statements if true
}
else{
    // execute this if none of above are true
}
```

Example

```
#include<iostream>
using namespace std;

int main()
{
    int num = -20;

    if(num == 0){
        cout << "Number is Zero"; } else if(num > 0){
        cout << "Number is positive";
    }
    else{
        cout << "Number is negative";
    }
   return 0;
}
```

4. Nested if-else in C++

If else conditions can be nested within one another as well.

```
if (condition1)
{
   // Executes when condition1 is true
   if (condition2)
   {
      // Executes when condition2 is true
   }
}
```

Example

```
#include<iostream>
using namespace std;

int main()
{
    int num = 20;

    if(num >= 0){
        if(num == 0){
            cout << "Number is Zero";
        }
        else{
            cout << num << " is positive";
        }
    }
    else{
        cout << num << " is negative";
    }
   return 0;
}
```

5. Switch Case

It is ideal for whenever we can to merge a lot of if – else loops into multiple cases checking and execute unique statements if one of them is true.

```
switch(expression)
{
   case constant-expression1:
      // statement(s);
      break;

   case constant-expression2:
      // statement(s);
      break;

   // you can have any number of case statements
   default: // Optional but ideal
      // statement(s);
}
```

Example

```
#include<iostream>
using namespace std;

int main()
{
  char op;
  int num1 = 20, num2 = 5;

  cout << "Enter operator either + or - or * or /: "; cin >> op;

  switch(op)
  {
    case '+':
        cout << num1+num2;
        break;
    case '-':
        cout << num1-num2;
        break;
    case '*':
        cout << num1*num2;
        break;
    case '/':
        cout << num1/num2;
        break;

    //if case label not available example op = %
    default:
        cout << "Select valid choice";
        break;
    }
  return 0;
}
```

Jump Statements

Now we will discuss jump statements which are

- Jump Statements:
    - break
    - continue
    - goto
    - return

Knowledge for looping like – For, while, do while is important. Make sure that you visit C++ Loop Types before moving the topics below.

Break Statement

Whenever a **break** statement is encountered inside a loop (for or While or Do-while), the loop is immediately terminated and program control resumes at the next statement following the loop

```
for(init;conditions;increments){

    if(some condition(s)){
        // For loop immediately terminates
        // no matter if there are pending iterations
        break;
    }
}
// program flow will come here immediately
```

Example

```
#include<iostream>
using namespace std;

int main()
{
    // ideally should run 10 times b/w [1,10]
    for(int i = 1; i <= 10; i++){
        // whole loop terminates when i becomes divisible by 5
        if(i % 5 == 0)
            break;

        cout << "i: " << i << endl;
    }
  return 0;
}
```

Continue Statement

Whenever a continue statement is encountered inside a loop, the current iteration of the loop is skipped and the loop executes from the next iteration

```
for(init;conditions;increments){

    if(some condition(s)){
        // Current iteration of loop is skipped
        // Starts from next iteration
        break;
    }
}
// program flow will come here after loop completion
```

Example

```
#include<iostream>
using namespace std;

int main()
{
    // ideally should run 10 times b/w [1,10]
    for(int i = 1; i <= 10; i++){

        // Even iterations are skipped
        // it true then continue statement forces
        // loop to be started from next iteration
        if(i % 2 == 0)
            continue;

        // statements skipped if continue statement executes
        cout << "i: " << i << endl;
    }
    return 0;
}
```

Goto Statement

It is a unique type of jump statement which is sometimes also called an unconditional jump statement. Which is used to jump from anywhere to anywhere within a function.

Let us look at a sample goto program to understand more about –

Example

```
#include<iostream>
using namespace std;

// function to check even or not
void checkNum(int num)
{
    if (num > 0)
    // jump to positive
        goto positive;
    else if(num < 0)
    // jump to negative
        goto negative;
    else
    // jump to zero
        goto zero;

positive:
    cout << num << " is positive";
    // return if positive
    return;
negative:
    cout << num << " is negative";
    // return if negative
    return;
zero:
    cout << num << " is zero";
}

int main()
{
    int num = -12;
    checkNum(num);
    return 0;
}
```

Do not use Goto -1. Goto statement is rarely used by programmers and its used is discouraged due to complex implimentation of goto statement.

2. Use makes code analysis and verification difficult

3. It's better to use break, continue or functions instead of goto statement

Return Statement

Return statement helps us to return from the execution flow of a program from wherever the return statement is encountered. It may or may not need any conditional statement.

As soon as we see return statement the program flow is returned to wherever the function was called from. A void function may not have a return type but for all others the return type must be a specified datatype.

Example

```
#include<iostream>
using namespace std;

// Here we have a non void return type
// function to calculate multiplication of two numbers
int multiply(int a, int b)
{
    int mul = a * b;
    return mul;
}

// returns void
// function to display result
void display(int val)
{
    cout << "The product is : "<< val;
    return;
}

int main()
{
    int num1 = 15;
    int num2 = 4;

    int result = multiply(num1, num2);
    display(result);

    return 0;
}
```

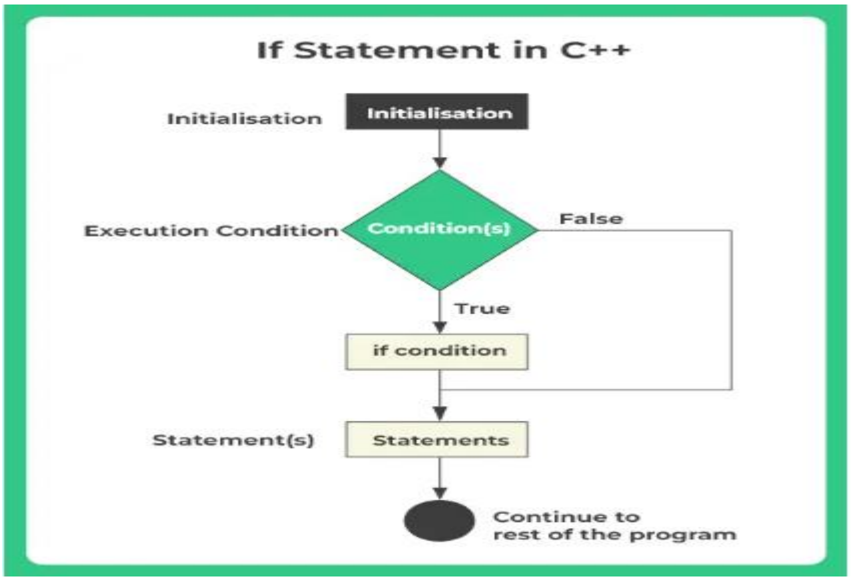
If Statement in C++

Syntax:

```
if(condition){
// statements to be executed if condition is true;
}
```

Working of if statement in C++:

- **Step 1 :** In the first step, condition/expression inside parenthesis is checked whether it is true or false.
- **Step 2 :** If statement is **True**, statements inside if block will be executed for performing the operations.
- **Step 3** : If the condition is **False** , statements after the end of if block will be executed



Example 1:

```
#include<iostream>
using namespace std;
int main()
{
   int j = 7;
   if (j > 1) {  // checking the condition and take decision
      cout<<"PrepInsta is one of the best for placement preparation"< < endl;
   }
   cout<<"Prepinsta"< < endl;
   return 0;
}
```

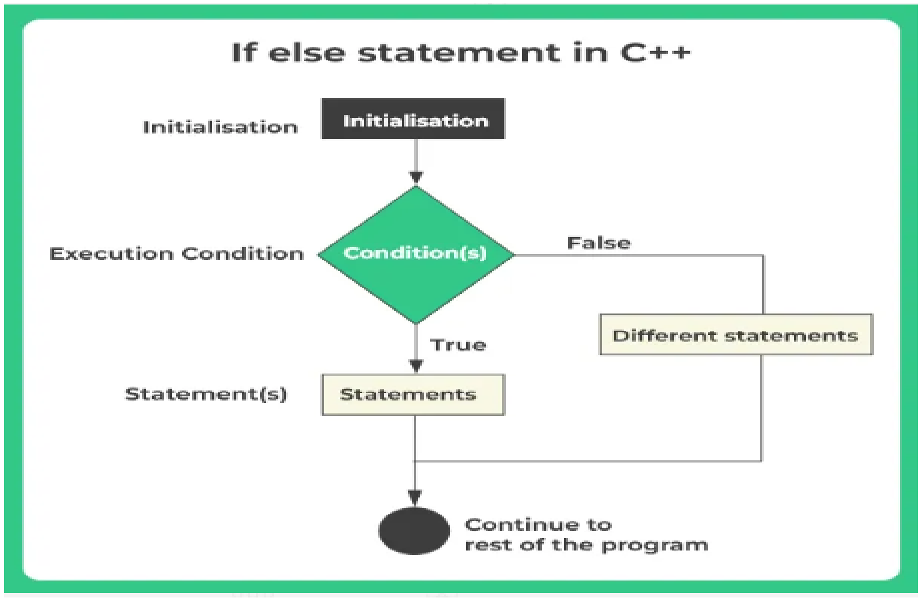
If else statement in C++

Syntax:

```
if(condition){

// statements to be executed if condition is true;
}

else{

// statements to be executed if condition is false;

}
```

Working of if else statement :

- **Step 1 :** In the first step, condition/expression is checked whether it is true or false.
- **Step 2 :** If statement is **True**, if block will be executed for performing the operations.
- **Step 3** : If the condition is **false** , else block will be executed for performing the executions



Implementation of If else statement in C++

```
#include <iostream>
using namespace std;

int main() {
    int i=5;

    if(i>1){     // checking the condition and take decision
        cout<<"PrepInsta is one of the best for placement preparation";
    }
    else{
       cout<<"PrepInsta";
    }
    return 0;
}
```

While Loop in C++

While Loop Syntax

```
while(condition(s))
{
    // execute statement(s)

}
```

- Certain condition(s) are checked before the loop
- If the condition(s) is(are) true, code inside is executed
- The condition then is evaluated again. If true then the code inside is executed again.
- This will keep happening until the condition doesn’t become false.

Program 1: (Print first 10 Natural Numbers)

```
#include <iostream>
using namespace std;

int main()
{
    int i = 1; // initialization

    // condition
    while(i <= 10)
    {
        cout << i << " ";
        i++; // increment
    }

    return 0;
}
```

Infinite while loop

A while loop can run infinitely if the conditions are not checked properly. Following are examples of while loops running infinitely –

- Explicitly writing true in condition

- Setting conditions that they will always be true
- Forgetting to increment the iterator of the loop

Syntax for Infinite while loop

```
// condition will always be true infinitely
while(true)
{
   // do some work here
}
```

Example 1

```
#include <iostream>
int i = 100;

// condition will always be true infinitely
while(i > 10)
{
   cout << i << " ";
   i++;
}
// will keep on printing numbers from 100 to infinity
```

Inappropriate use of the semicolon

- A semicolon after the condition part will stop the next statements not to execute
- Hence ***all the statements after the semicolon are not executed and just dead statements***

Do-while Loop in C++

Do While Loop in C++ Language

- Do while loop is very similar to the while loop. To do repetitive work control structure has been created.
- In do while we execute some statements first and then check a condition.
- That is why Do-while is known as exit controlled loop. Which means that it will be implemented atleast once
- If the condition is met we execute it again. We keep doing this until the condition keeps meeting.

Syntax:

```
do{
    // execute statement(s)
}while(condition(s));
```

Along with increment and initialization logic it may look like :

```
initialization;

do{
    // execute statement(s)
    incrementation;

}while(condition(s));
```

Example 1:

```
#include<iostream>
using namespace std;
int main()
{
    int i = 1000;
    // i is set as 1000 which is definitely greater than 5
    // the condition set below is (i <= 5) which will surely be never met
    // as the loop is incrementing (i++)
    // but still the loop is executed once anyways

    do{
        cout << i << ". Hello World\n";
        i++;
    }
    while(i <= 5);
    return 0;
}
```

For loop in C++

Looping in C++

Whenever, we need to do some repetitive task over and over again. We use looping, by using logical statements we tell our program how many times we want repetitive statements to be executed over and over.

Example –

- Printing hello world 100 times
- Printing first 100 natural numbers

Basic Syntax of a for Loop

```
for(statement1 ; statement2; statement3)
{
    // execute this code
}
```

- **Statement 1:** Called **initialization part.** Executed once, before execution of code block.
- **Statement 2:** Called **condition part** of the loop. Executed every time before execution of code block to check if the condition is satisfied
- **Statement 3:** Called **incrementation part**. Executing every time after code block inside is executed

Example 1:

```
#include<iostream>
using namespace std;
int main()
{
    int n = 10;
    for(int i = 1; i <= n; i++){
        cout << i << " ";
    }
    return 0;
}
```

For-each loop in C++

For-Each Loop in C++ Explained

The For-Each loop, also known as the Range-Based for loop, was introduced in C++11. It provides an elegant and concise way to iterate through elements within a range-based container, such as arrays, vectors, or other containers that support iteration. The For-Each loop greatly simplifies the traditional For loop syntax, making the code more expressive and less prone to errors.

Syntax

```
for (variable : collection)
{
// statement(s)
}
```

Benefits of Using For-Each Loop

Prevents Off-By-One Errors: The loop automatically handles iteration boundaries, preventing common off-by-one errors often encountered in traditional For loops.

Simplicity and Readability: The For-Each loop simplifies the syntax and makes the code more readable, especially when dealing with complex data structures.

Efficient and Optimized: The For-Each loop is optimized by the compiler, ensuring efficient iteration through the container elements.

No Need for Index Management: Unlike traditional For loops, you don't need to manage the loop index manually, reducing code complexity

Common Use Cases of For-Each Loop

1. Iterating Through Arrays

For-each loop in C++ can be effectively used to iterate through arrays:

```
int numbers[] = {1, 2, 3, 4, 5};

for (int num : numbers) {
    // Code to process each element in the array
}
```

2. Traversing Vectors

For-each loop can also be used to traverse vectors:

```
#include <vector>

std::vector fruits = {"apple", "banana", "orange"};

for (const std::string& fruit : fruits) {
    // Code to work with each fruit
}
```

3. Iterating Through Containers

You can use the For-Each loop with other containers, such as lists, maps, and sets:

```
#include <list>
#include <map>
#include <set>

std::list temperatures = {23.5, 26.1, 20.8, 19.9};

for (double temp : temperatures) {
    // Code to process each temperature reading
}
```

Example 1

```
#include<iostream>
using namespace std;
int main()
{
    int arr[] = {10, 20, 30, 40, 50};
    // introduced in C++11
    // allows to print each item of array/vector
    for (int x : arr)
        cout << x << " ";
    return 0;
}
```

Limitation of for-each loop

- It is applicable only for arrays and container classes, it cannot be used for normal variables
- It cannot be customized with conditions, I,e you can only print data from start to end

Why to use for_each

- for_each loops improve the overall performance of code
- It increases code readability

Nested Loops in C++

Nested Loops in C++ Language

C++ allows nesting of looping structures like – For, while, Do-while, for-each etc. To allow for complex code logic and control structure.We will first look at an example syntax for nesting and then a few programs to understand how it can be done.
Even though nesting is allowed for all types of loops. For loop nesting is most common. Both homogeneous and heterogeneous nesting is allowed. Example –
1. For loop within For loop / while loop within while loop
2. For loop within while loop / Do-while within for loop

Homogeneous nesting

For Loop

```
for (initialization; condition(s); increment) {
   // statement(s) of outer loop
    for (initialization; condition(s); increment) {
        // statement(s) of inner loop
   }
   // statement(s) of outer loop
}
```

While Loop

```
while(condition(s)) {
   // statement(s) of outer loop
   while(condition(s)) {
       // statement(s) of inner loop
   }
   // statement(s) of outer loop
}
```

Do-while Loop

```
do{
    // statement(s) of outer loop
    do{
        // statement(s) of inner loop
    }while(condition);
    // statement(s) of outer loop
}while(condition);
```

Heterogeneous nesting

While all kinds of combinations of nesting can be done, we have taken an example of for loop inside a while loop and that while loop inside do-while

Example

```
do{
    // statement(s) of do-while loop
    while(condition) {
        // statement(s) of while loop
        for ( initialization; condition; increment ) {
            // statement(s) of for loop
        }
        // statement(s) of while loop
    }
    // statement(s) of do-while loop
}while(condition);
```

Execution flow

- When we are working with ***nested loops*** always execution starts from the outer loop test condition, if it is true control will pass to the outer loop body
- Now the inner loop test condition is checked, if the inner loop test condition is true, control will pass within the inner loop until the inner loop test condition becomes false
- In the inner loop if the condition becomes false control goes to the Outer loop
- Until the outer loop test condition becomes false outer loop executes n number of times.

Example 1

```
#include<iostream>
using namespace std;
int main(){
    for(int i = 1; i <= 5; i++)
    {
        cout << "Table of " << i << " : ";
        for(int j = 1; j <= 10; j++){
            cout << i*j << " ";
        }
        cout << "\n";
    }
    return 0;
}
```

Switch Case in C++

Switch Case in C++ Language

- First, the compiler searches the switch expression, top to bottom from available case labels
- If a case label is matched with the switch expression then body under case label gets executed, then all other case labels get skipped
- If no case label is matched with switch expression body under default get executed
- Remember, only one case label gets executed from available case labels, i.e case label that is matched with the switch expression

Syntax:

```
switch (expression)
{
    case constant1:
         //code
        break;
    case constant2:
        // code
        break;
        .
        .
        .
    default:
}
```

C++ program to demonstrate switch case

```
#include<iostream>
using namespace std;
int main ()
{
  char oper = '+';
  float num1 = 8, num2 = 3;
  cout << "Number 1 = " << num1 << endl;
  cout << "Number 2 = " << num2 << endl;
```

```
 cout << "operator = " << oper << endl;
 cout << "result: ";
 switch (oper)
   {
   case '+':
     cout << num1 << " + " << num2 << " = " << num1 + num2;
     break;
   case '-':
     cout << num1 << " - " << num2 << " = " << num1 - num2;
     break;
   case '*':
     cout << num1 << " * " << num2 << " = " << num1 * num2;
     break;
   case '/':
     cout << num1 << " / " << num2 << " = " << num1 / num2;
     break;
   default:

     cout << "Error! The operator is not correct";
     break;
   }

 return 0;
}
```

goto Statement in C++

What is C++ goto statement ?

In C language, the goto statement is used as unconditional jump statement which is used to jump from one statement to another statement at another location. By using the goto statement you can jump to above and below the current code flow.

Note: We can use goto statement when we want to exit the nested loops or multiple loops instead of break statement for each loop.

Syntax

```
goto label;
... .. ...
... .. ...
label:
statement;
```

Here, label cannot be the keyword. Basically label is an identifier. When goto is encountered the flow of code is transferred from that statement to label and the statements below it gets executed.

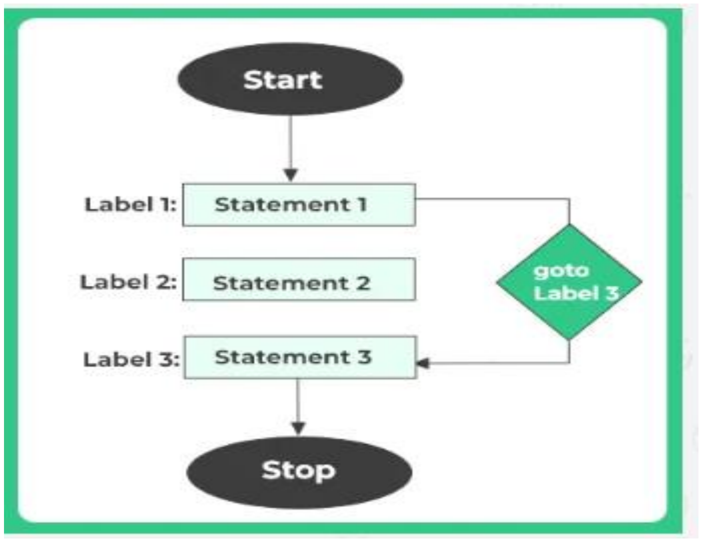
Disadvantages of goto statement :

- Since,goto statement is a unconditional statement,the efficiency of the program reduces by moving the control from one point to another unconditionally.
- The goto statement can be use to move the control up or down which may lead to infinite loop.

Example :

```
x:statement-1;
  statement-2;
  statement-3;
 goto x;
```

- If we increase the number of goto statement in a program then the tracing become difficult.



Example:

```
#include<iostream>
using namespace std;
int main ()
{
  int k = 12;
LOOP:do
    {
      if (k == 14)
                {
                  k = k + 1;
                  goto LOOP;      //goto statemet
                }
```

```
     printf ("value of k: %d\n", k);
     k++;

   }
  while (k < 25);

  return 0;
}
```
NOTE: The goto statement is highly discouraged as it increase the logic complexity of code and decrease the readability.

C++ break Statement

What is C++ break statement ?

In C++ language, the break statement works by exiting the innermost loop or switch statement that it is in. When the break statement is encountered, the program immediately exits the loop or switch statement and continues with the next statement after the loop or switch.

Syntax:

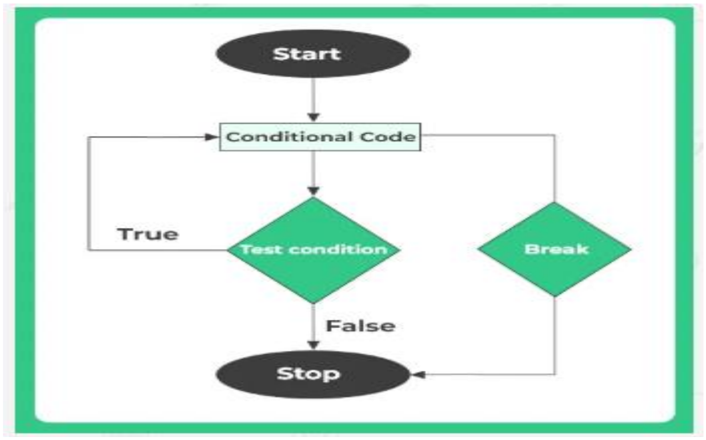
```
break;
```

The break statements are typically used when we are not sure about the number of iterations which will be used for a loop or when we want to terminate the loop based on some condition.

In C++, the break statement is used in three ways:

- Whenever a break statement is encountered within a loop, then the loop is terminated early, before the loop condition is evaluated again and and program continues with the next statement after the loop.
- The break statement can be used to exit a switch statement early.
- It can also be used to exit a loop inside a switch statement.

Note: In case of nested loop(loop inside loop), the break statement will only exit the inner loop. The outer loop will continue to run, and the program will continue to execute the next iteration of the outer loop.



Example 1:

```
#include<iostream>
using namespace std;
int main()
{
    for (int i = 0; i < 10; i++) {
     if (i == 5) {
       break;
     }
     std::cout << i << std::endl;
     }

    return 0;
}
```

Note: The break statement only exits the innermost loop or switch statement that it is in. If you want to exit multiple loops or a loop inside a switch statement, you can use goto or a flag variable to achieve this.

C++ continue statement

What is C++ continue statement ?

In C++ language, the continue statement works by skipping the remaining statements in the current iteration of a loop and immediately starting the next iteration. When the continue statement is encountered, the program skips the remaining statements in the loop body and goes back to the top of the loop to evaluate the loop condition and decide whether to continue with the next iteration.

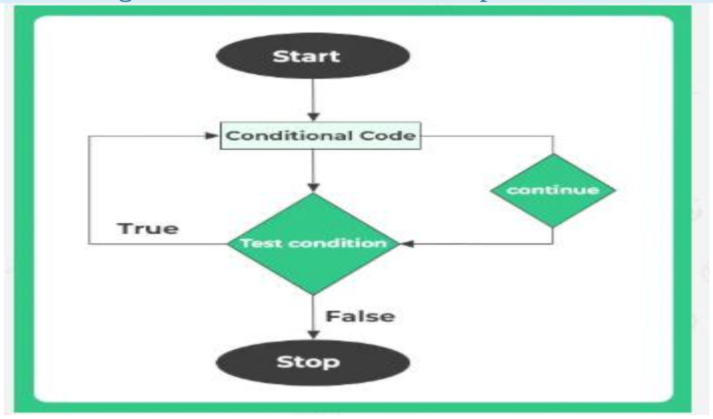
Syntax:

```
continue;
```

The continue statement is used in C++ for the following purposes:

- The continue statement can be used to skip certain statements in a loop and continue with the next iteration.
- This statement can also be used to skip certain iterations in a loop based on some condition.
- It can also be used to skip certain iterations in a nested loop.

Note: In case of nested loop(loop inside loop), the continue statement will only skip the remaining statements in the inner loop and continue with the next iteration of the inner loop.



Example 1:

```
#include <iostream>
using namespace std;

int main()
{
   for (int i = 0; i < 10; i++) {
     if (i % 2 == 0) {
      continue;
  }
   std::cout << i << std::endl;
}

    return 0;
}
```

Note: The continue statement only affects the current iteration of the loop. It does not exit the loop or switch statement like the break statement does.

Arrays in C++

Basic properties of Array

- Array index always starts with zero and ends with n-1 (n is the size)
  ex:int a[5] has the index range a[0],a[1],….a[4]
- An array is homogeneous: It stores elements of the same type only.
  ex: int a[5], integer array stores only integer elements.

Ways to define an array

Array definition by specifying the size

```
int a[5];
```

As per C99 versions, we can also declare an array of user-specified size

```
int n = 5;
int a[n];
```

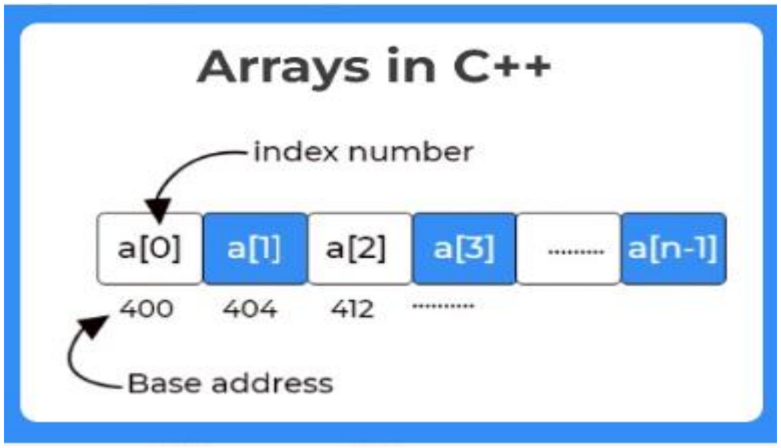
Array definition by initializing elements

```
int a[] = { 10, 20, 30, 40,50 } ;
```

Here size corresponds to the number of elements initialized, hence size = 4.

Array definition by specifying size and initializing

```
int a[5] = { 10, 20, 30, 40 ,50};
```



Initializing and accessing 1D array

```
#include < iostream >
using namespace std;
int main()
{
  int a[5]={6,9,1,5,3};//direct declartion and initialisation
  cout<<"elements are:\n";
  for(int i=0;i<5;i++)// iterate all the elements
    cout<<"a["<<i<<"]="<<a[i]<<"\n";//access though index
return 0;
}
```

Advantages

- Reduces program size
- Reduces access time
- Efficient data organization

Disadvantages

Array size is static, due to which there may be chances of wastage and shortage of memory

Array elements are addressed sequentially

- The address of each block will be sequentially located to access the memory blocks by using pointers.
- Generally, normal user access the array data with subscripted index i.e. a[n] internally compiler will access by address and pointers.

Additions Facts about Arrays

An array index cannot be negative or zero

```
a[-2];//invalid
a[0];//invalid
```

Unused space in the array is always filled with zeros

```
int a[5]={1,2,3};//a[3],a[4] are filled with zeros
```

All array elements are Garbage values before initialization of values

```
int a[5];//a[0]...a[5] are garbage values
```

Declaring an array without size is invalid

```
int a[];/invalid
```

The below array declaration sets all elements in the array to 0

```
int a[10]={0};
```

2D Arrays

- Two-dimensional arrays are represented using two indexes namely row and column
- A 2D array is viewed as an array of arrays.

- ***Size of multidimensional arrays***
    - Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the number of rows and columns.

  For example:

    - The array int a[3][3] can store total (3*3) = 9 elements.
    - Similarly array int a[2][3] can store total (2*3) = 6 elements.

Defining 2D array

```
data_type name[rows][columums];
```

ex: int a[3][3];

Initializing Two – Dimensional Arrays:

First way:

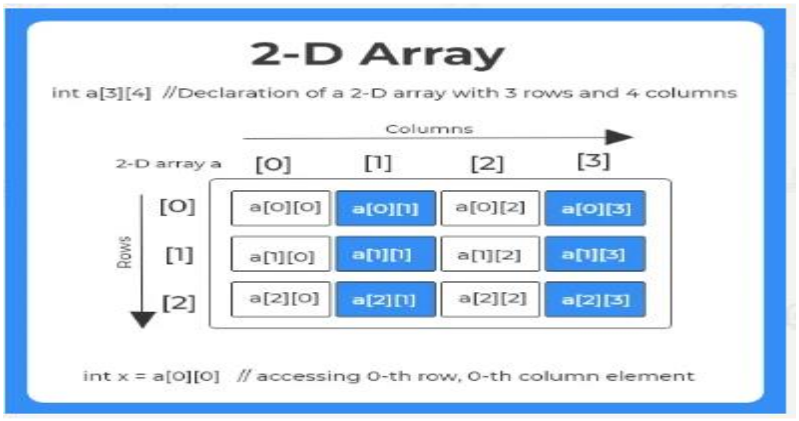
```
int a[3][3] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 };
```

The elements will be filled in the array in the order, first 3 elements from the left in the first row, next 3 elements in the second row and so on.

Second way:

```
int a[3][3] = {{0,1,2}, {3,4,5,}, {6,7,8}};
```

- This type of initialization makes use of nested braces. Each set of inner braces represents one row.
- In the above example, there are total of three rows so there are three sets of inner braces.



Accessing 2D Arrays
Elements in two-dimensional arrays are commonly referred by a[i][j] where i is the row number and 'j' is the column number.
example:

```
a[2][1];
```

- a[2][1] element is located at index 2nd row and 1st column
- A two – dimensional array can be seen as a table with 'm' rows and 'n' columns where the row number ranges from 0 to (m-1)

  example: array a[3][3] ranges from i.e a[0][0] to a[2][2]

C++ program to initialise and access a 2D array

```
#include<iostream>
using namespace std;
int main ()
{
  // an array with 5 rows and 2 columns.
  int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
  for ( int i = 0; i < 5; i++ )//iterate rows
   for ( int j = 0; j < 2; j++ ) //iterate columns
   {
     cout << "a[" << i << "][" << j << "]: ";
     cout << a[i][j]<< endl;
   }
return 0;
}
```

Advantage of Multidimensional arrays

Access time is reduced as search confines only to that particular row.
ex: to access a[3][2] search starts directly from row index 2 rather than starting from index 0 as in the case of the 1D array.

Space Efficiency: In situations where data is arranged in a grid or tabular format, using multidimensional arrays can be more space-efficient compared to using separate arrays for each dimension. This is because multidimensional arrays store all related data elements in contiguous memory locations.

Simplified Code: When dealing with complex data structures, such as multi-dimensional maps or game boards, using multidimensional arrays can lead to more concise and readable code. Accessing elements using row and column indices is intuitive and improves code clarity.

Structured Data Storage: Multidimensional arrays allow us to store data in a structured manner, mimicking real-world scenarios where data often comes in multiple dimensions. For example, in a 2D array representing a grid, each element corresponds to a specific row and column, making it easier to access and manipulate related data.

Matrix Operations: Multidimensional arrays are particularly useful when dealing with matrix operations in mathematical and scientific computations. These arrays can represent matrices, making it convenient to perform matrix addition, multiplication, and other operations.

Strings in C++

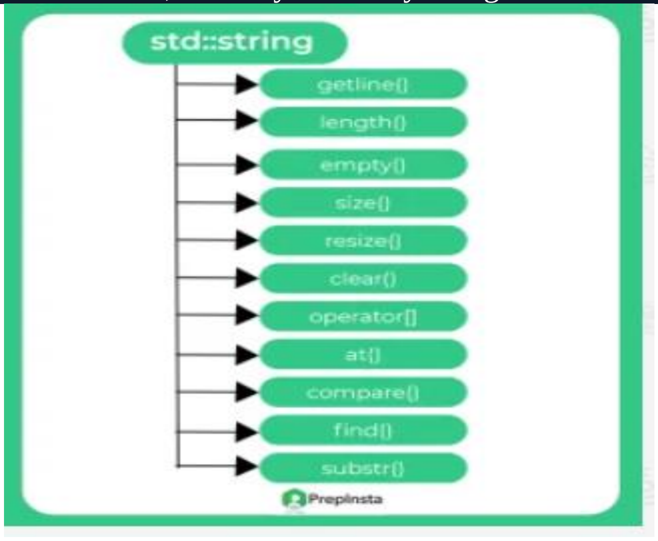
Declaration,access and initialization of String in C++

A string is defined as an array of characterswith additional predefined methods support through headers

```
std::string ="value";
```

ex:

```
std::string name ="trishaank";//direct declaration and initialization
cout<<name;//directly access by string name
```



Advantages of Using Strings in C++:

Easy Manipulation:C++ strings provide a wide range of built-in functions that make it easy to manipulate and process textual data. Concatenation, searching, replacing, and other common string operations can be performed efficiently.

Dynamic Size: Unlike character arrays, C++ strings have a dynamic size. This means you don't need to worry about allocating an exact amount of memory for your string, as it can grow or shrink as needed.

Safer Operations:C++ strings handle memory allocation and deallocation automatically, reducing the risk of buffer overflow or memory leaks, which are common issues when working with character arrays.

getline()

- It is used to ***accept a string of multiple words*** i.e it considers whitespace also a character in it
- getline() function takes two parameters.
  The first one is std::cin and the second one is the name of our string variable.

```
#include <iostream>
using namespace std;
int main ()
{
  string name;
  cout << "\nenter a name:";
•     getline (cin, name);                 //reading a string
•     cout << "\ngiven string is " << name;
•     return 0;
•   }
```

Concatenation operator(+)

We can directly ***join the strings*** using '+' operator, hence it as an overloaded operator

```
#include <iostream>
using namespace std;
int main(){
string s1="trish";
string s2="rish";
cout<<s1+s2<<endl;//trishrish
return 0;
}
```

length()

length() function ***returns the length of the string including whitespace characters.***

```
#include <iostream>
using namespace std;
int main()
{
string name; name = "My name is 3shaank";
cout << name.length() <<endl;//includes whitespaces also
return 0;
}
```

find()

find() function **finds the position of the first occurrence of a character**

```
#include <iostream>
 using namespace std;
 int main()
{
string str1 ("3sh");
cout << str1.find('s') << endl;//index of 's'
return 0;
}
```

compare()

compare() function ***compares the value of a string (str1) with another string (str2).***

1. It returns 0 if both the strings are equal.
2. It returns a positive value if either str1 > str2 or the first unmatched letter of str1 is greater than that of str2.
3. It returns a negative value if either str1 < str2 or the first unmatched letter of str1 is less than that of str2.

clear()

This function ***clears the data present strings and makes it empty.***

```
#include <iostream>
using namespace std;
int main()
{
string s = "I hate my college";
s.clear();//string became empty
cout << "Value of s is :" << s << endl;
cout<<"size of s"<<s.length();// 0 because empty
return 0;
}
```

C++ Multidimensional Arrays

Multidimensional Arrays in C++

In C++ programming language, the multi-dimensional arrays are represented using two indexes i.e. row number and column number respectively.Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the number of rows and columns.

Elements in two-dimensional arrays are commonly referred by a[i][j] where i is the row number and ‘j’ is the column number.

Declaration of multi-dimension array

```
int p[3][4];
```

In the above given declaration of the multi dimensional array, p is a 2D array which can hold 12 elements.

Initialization of multi-dimension array

First way:

```
int a[3][3] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 };
```

The elements will be filled in the array in the order, first 3 elements from the left in the first row, next 3 elements in the second row and so on.

Second way:

```
int a[3][3] = {{0,1,2}, {3,4,5,}, {6,7,8}};
```

- This type of initialization makes use of nested braces. Each set of inner braces represents one row.
- In the above example, there are total of three rows so there are three sets of inner braces.

Implementation of Multi-dimension Array in C++

Example 1:

The following code shows the implementation of multi-dimension array.

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int p[3][2] = {{2, -5},{4, 0},{9, 1}};
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 2; ++j)
        {
            cout << "p[" << i << "][" << j << "] = " << p[i][j] << endl;
        }
    }
    return 0;
}
```

Converting Number to String in C++

Converting a Number into String

In C++ programming language, there are mainly three methods for converting a number into string which are as follows :

1. Using string Stream
2. Using to_string()
3. Using boost lexical cast

Method 1 : Using string Stream

In this method, a string stream declares an object which takes a number value initially and then gets converted to a string using str().

Example:

```
#include<iostream>
#include<sstream>
#include<string>
using namespace std;
int main()
{
  int num = 20;
  ostringstream str1;
  str1 << num;
  string p = str1.str();
  cout << "The newly formed string from number is : ";
  cout << p << endl;
  return 0;
}
```

Method 2 : Using to_string()

In this method, the function to_string() takes a number(which can be any data type) and returns the number in the string data type.

Example:

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
        int i_val = 20;
        float f_val = 30.50;
        string stri = to_string(i_val);
        string strf = to_string(f_val);
        cout << "The integer in string is : ";
        cout << stri << endl;
        cout << "The float in string is : ";
        cout << strf << endl;
        return 0;
}
```

Method 3 : Using boost lexical cast

In this method, the ” lexical_cast() ” function remains the same, but in ‘boost lexical cast‘ time argument list modifies to “lexical_cast(numeric_var).

Example:

```
#include<boost/lexical_cast.hpp>
#include<iostream>
#include<string>
using namespace std;
int main()
{
        float f_val = 10.5;
        int i_val = 17;
        string strf = boost::lexical_cast(f_val);
        string stri = boost::lexical_cast(i_val);
        cout << "The float value in string is : ";
        cout << strf << endl;
        cout << "The int value in string is : ";
        cout << stri << endl;
        return 0;
}
```