**Topic to Discuss in class related basic C++**

**1. Documentation section (give a notes or reference heading)**

**2. header files (add supporting program)**

**3. main function (it's a function , we write all code inside**

**this)**

**4. variable (reference name to store a value)**

**5. keywords(reserved words), identifier (rule for create name)**

**6. data type --int,float, double, char (categories of input)**

**7. operator (math. operation)**

**8. number (0-9)**

**9. alphabet (A-Z, a-z)**

**10.symbol (@ # $ % ^ \* ! ~ ' ")**

**11.brackets --> <> () {} []**

**12.punctuation ; : , .**

**13.install turbo c and dev c software**

**14. C++ is a middle level language**

**15.In C++,insertion operator “<<” is used for output and**

**extraction operator “>>” is used for input.**

**Variable Declaration**

**Variable- it is used to store the value or data or information.**

**Syntax:**

**datatype variablename**

**Ex: int suresh= 100;**

**int raja1=55;**

**int ramu=677;**

**int suresh=100,raja1=55,ramu=677;**

**float ece=50.123456;**

**double cse;**

**cse=5676.668767857765465;**

**char name='a';**

**char hi='5';**

**char hello='#';**

**char hhh='A';**

**string name="suresh123@"**

**Keywords**

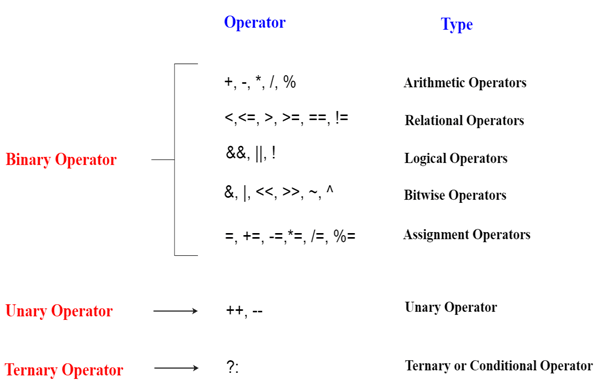
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Auto** | **break** | **case** | **char** | **const** | **continue** | **default** | **do** |
| **double** | **else** | **enum** | **extern** | **float** | **for** | **Goto** | **if** |
| **Int** | **long** | **Register** | **return** | **short** | **signed** | **Sizeof** | **static** |
| **struct** | **switch** | **Typedef** | **union** | **unsigned** | **void** | **volatile** | **while** |

**C++ Operators**

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.

There are following types of operators to perform different types of operations in C language.

* Arithmetic Operators +,- , \*,/ , X=10 %3
* Relational Operators <, <=, >, >=, X=(A==B), , Y= A != B
* Logical Operators – X= (A<B) &&(C>D)&& (A<=C), ||, X=!A
* Bitwise Operators & , |, ^, <<, >> ~
* Assignment Operator =, +=,-=,\*=/=,%=
* A=A+10, A+=10
* Unary operator
* Ternary or Conditional Operator
* Misc Operator (. Dot operator, sizeof() operator)



## Precedence of Operators in C++

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operators direction to be evaluated, it may be left to right or right to left.

Let's understand the precedence by the example given below:

1. **int** data=5+10\*10;

The "data" variable will contain 105 because \* (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C++ operators is given below:

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | ( ) [ ] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Right to left |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == !=/td> | Right to left |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Right to left |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

## DECISION MAKING STATEMENT

**In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.**

* **if statement**
* **if-else statement**
* **nested if statement**
* **if-else ladder**
* **switch case**

## if Statement

**The syntax of the if statement is:**

**if (condition) {**

**// body of if statement**

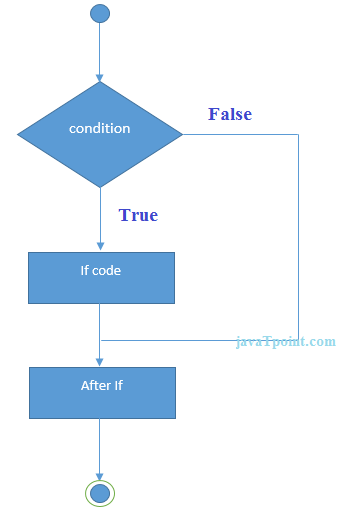
**}**

The if statement evaluates the condition inside the parentheses ( ).

If the condition evaluates to true, the code inside the body of if is executed.

If the condition evaluates to false, the code inside the body of if is skipped.

Note: The code inside { } is the body of the if statement.

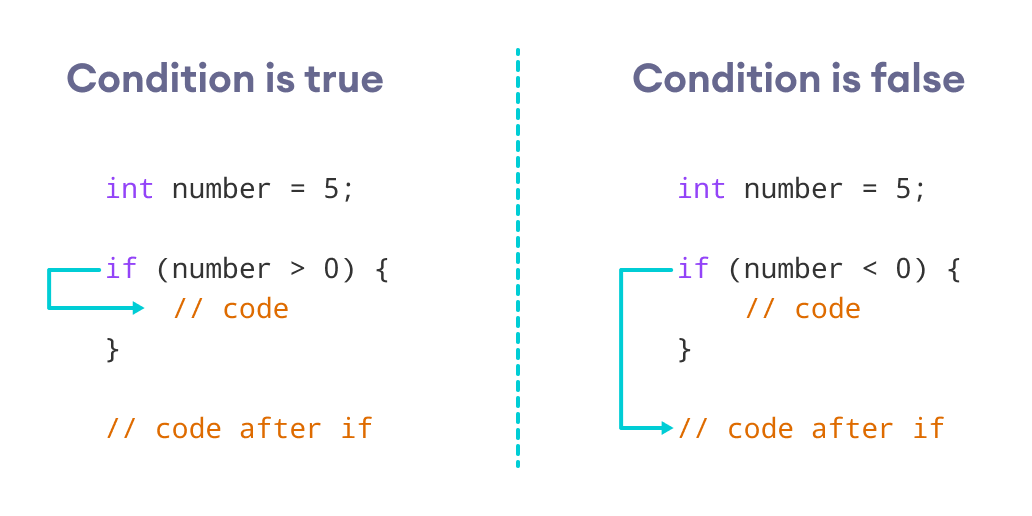


## If Example

1. **#include <iostream>**
2. **using namespace std;**
3. **int main () {**
4. **int age;**
5. **cin>>age;**
6. **if (age >=18)**
7. **{**
8. **cout<<"you allow to voting";**
9. **}**
10. **else**
11. **{**
12. **cout<<”you are not allow to voting ”;**
13. **}**
14. **return 0;**
15. **}**

**Output:/p>**

**It is even number**



// Program to print positive number entered by the user

// If the user enters a negative number, it is skipped

#include <iostream>

using namespace std;

int main() {

int number;

cout << "Enter an integer: ";

cin >> number;

// checks if the number is positive

if (number > 0) {

cout << "You entered a positive integer: " << number << endl;

}

cout << "This statement is always executed.";

return 0;

}

## if...else statement

**The if statement can have an optional else clause. Its syntax is:**

**if (condition) {**

**// block of code if condition is true**

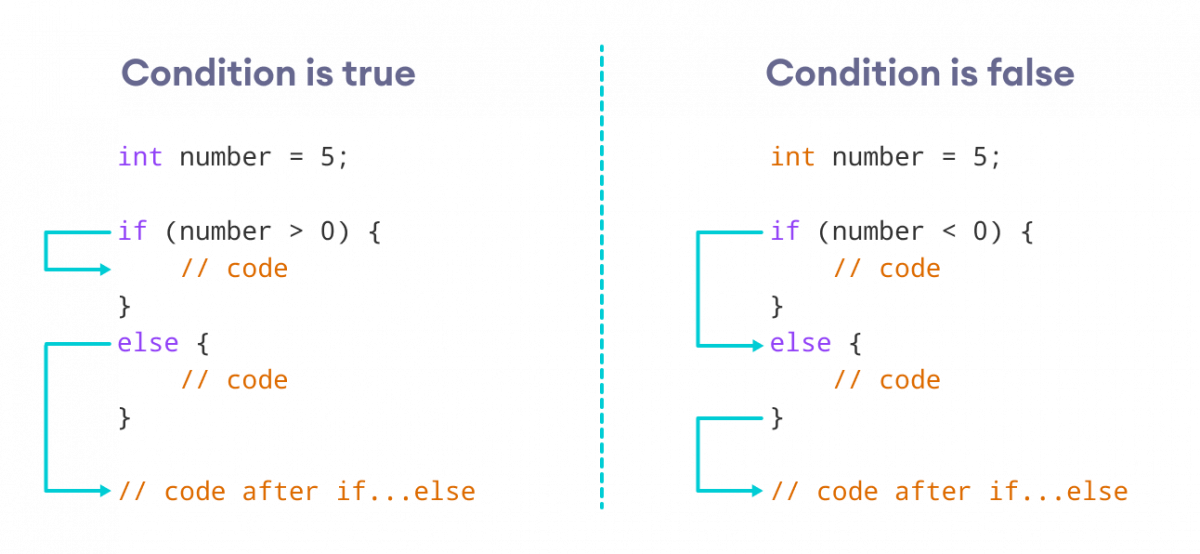
**}**

**else {**

**// block of code if condition is false**

**}**

**The if..else statement evaluates the condition inside the parenthesis.**



* **If the condition evaluates true,**
* **the code inside the body of if is executed**
* **the code inside the body of else is skipped from execution**
* **If the condition evaluates false,**
* **the code inside the body of else is executed**
* **the code inside the body of if is skipped from execution**

### Ex 2: C++ if...else Statement

// Program to check whether an integer is positive or negative

// This program considers 0 as a positive number

#include <iostream>

using namespace std;

int main() {

int number;

cout << "Enter an integer: ";

cin >> number;

if (number >= 0) {

cout << "You entered a positive integer: " << number << endl;

}

else {

cout << "You entered a negative integer: " << number << endl;

}

cout << "This line is always printed.";

return 0;

}

## Else-if ladder or statement

**The if...else statement is used to execute a block of code among two alternatives. However, if we need to make a choice between more than two alternatives, we use the if...else if...else statement.**

**The syntax of the if...else if...else statement is:**

**if (condition1) {**

**// code block 1**

**}**

**else if (condition2){**

**// code block 2**

**}**

**else if(cond3){}**

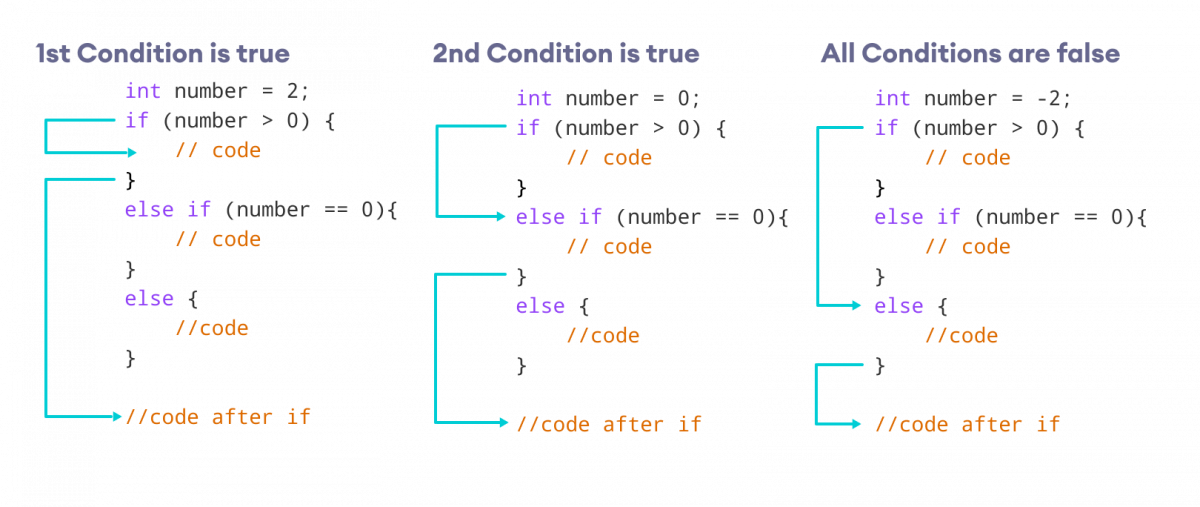
**else if (cond4){**

**// code block 3**

**}**

**Here,**

* **If condition1 evaluates to true, the code block 1 is executed.**
* **If condition1 evaluates to false, then condition2 is evaluated.**
* **If condition2 is true, the code block 2 is executed.**
* **If condition2 is false, the code block 3 is executed.**



### Ex 3: C++ else if

// Program to check whether an integer is positive, negative or zero

#include <iostream>

using namespace std;

int main() {

int number;

cout << "Enter an integer: ";

cin >> number;

if (number > 0) {

cout << "You entered a positive integer: " << number << endl;

}

else if (number < 0) {

cout << "You entered a negative integer: " << number << endl;

}

else {

cout << "You entered 0." << endl;

}

## Nested if...else statement

**Sometimes, we need to use an if statement inside another if statement. This is known as nested if statement.**

**Think of it as multiple layers of if statements. There is a first, outer if statement, and inside it is another, inner if statement. Its syntax is:**

**// outer if statement**

**if (condition1) {**

**// inner if statement**

**if (condition2) {**

**// statements**

**}**

**else {**

**//st**

**}**

**}**

**else {**

**if {**

**//st3**

**}**

**else {**

**//st4**

**}**

**}**

**Notes:**

* **We can add else and else if statements to the inner if statement as required.**
* **The inner if statement can also be inserted inside the outer else or else if statements (if they exist).**
* **We can nest multiple layers of if statements.**

### Ex 4: C++ Nested if

// C++ program to find if an integer is positive, negative or zero using nested if statements

#include <iostream>

using namespace std;

int main() {

int num;

cout << "Enter an integer: ";

cin >> num;

// outer if condition

if (num != 0) {

// inner if condition

if (num > 0) {

cout << "The number is positive." << endl;

}

// inner else condition

else {

cout << "The number is negative." << endl;

}

}

// outer else condition

else {

cout << "The number is 0 and it is neither positive nor negative." << endl;

}

cout << "This line is always printed." << endl;

return 0;

}

**switch case Statement**

**The switch statement allows us to execute a block of code among many alternatives.**

**The syntax of the switch statement in C++ is:**

**switch (expression) {**

**case constant1:**

**// code to be executed if**

**// expression is equal to constant1;**

**break;**

**case constant2:**

**// code to be executed if**

**// expression is equal to constant2;**

**break;**

**.**

**.**

**.**

**default:**

**// code to be executed if**

**// expression doesn't match any constant**

**}**

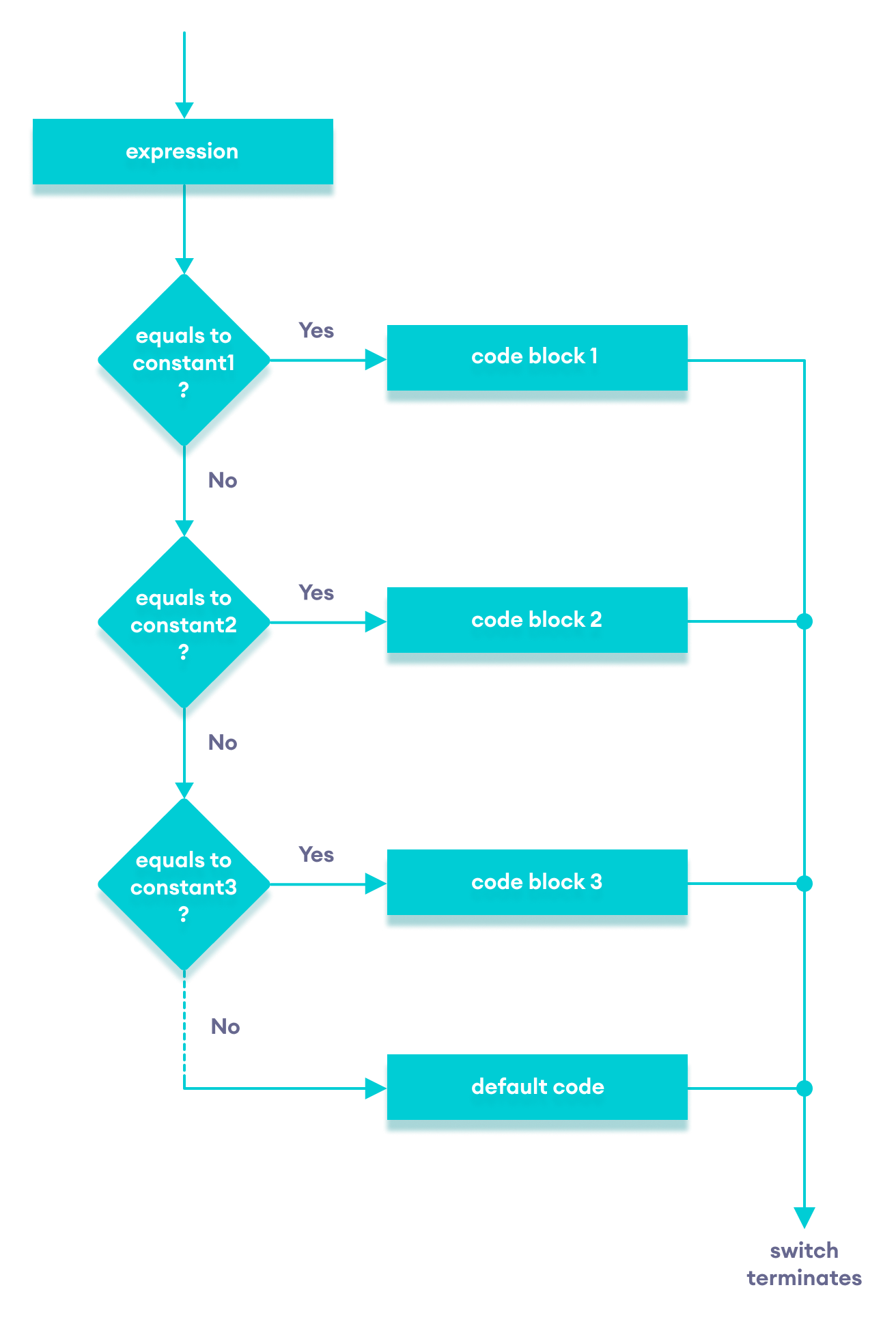
**How does the switch statement work?**

**The expression is evaluated once and compared with the values of each case label.**

* **If there is a match, the corresponding code after the matching label is executed. For example, if the value of the variable is equal to constant2, the code after case constant2: is executed until the**[**break statement**](https://www.programiz.com/cpp-programming/break-statement)**is encountered.**
* **If there is no match, the code after default: is executed.**

**Note: We can do the same thing with the if...else..if ladder. However, the syntax of the switch statement is cleaner and much easier to read and write.**

## Flowchart of switch Statement



### Ex: Create a Calculator using the switch Statement

// Program to build a simple calculator using switch Statement

#include <iostream>

using namespace std;

int main() {

char oper;

float num1, num2;

cout << "Enter an operator (+, -, \*, /): ";

cin >> oper;

cout << "Enter two numbers: " << endl;

cin >> num1 >> num2;

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:

// operator is doesn't match any case constant (+, -, \*, /)

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

break;

}

return 0;

}

**Looping Statement**

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

1. **while loop**
2. **do...while loop**
3. **for loop**

## while  Loop

**The syntax of the while loop is:**

**while (condition) {**

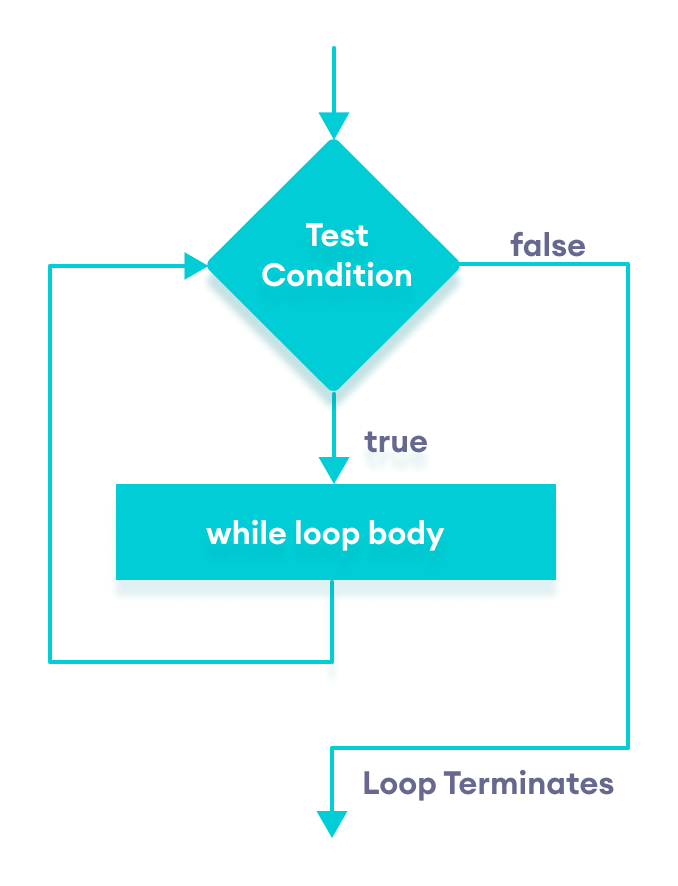
**// body of the loop**

**}**

**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.**

### Flowchart of while Loop

Flowchart of C++ while loop

### Ex 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;

}

### Ex 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;

}

## do...while Loop statement

**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;**

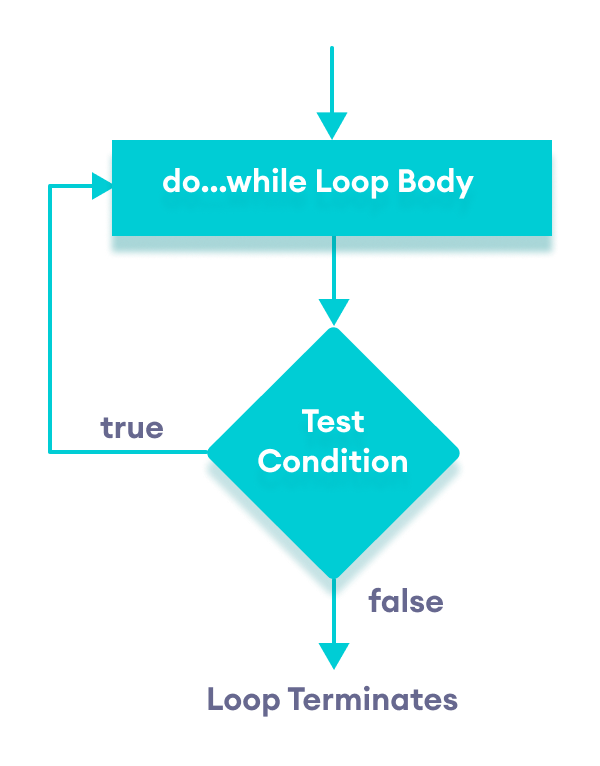
**}**

**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



### Ex 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;

}

### Ex 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;

}

## 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 loop**

**while(true) {**

**// body of the loop**

**}**

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

**// infinite do...while loop**

**int 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 loop

**The syntax of for-loop is:**

**for (initialization; condition; update) {**

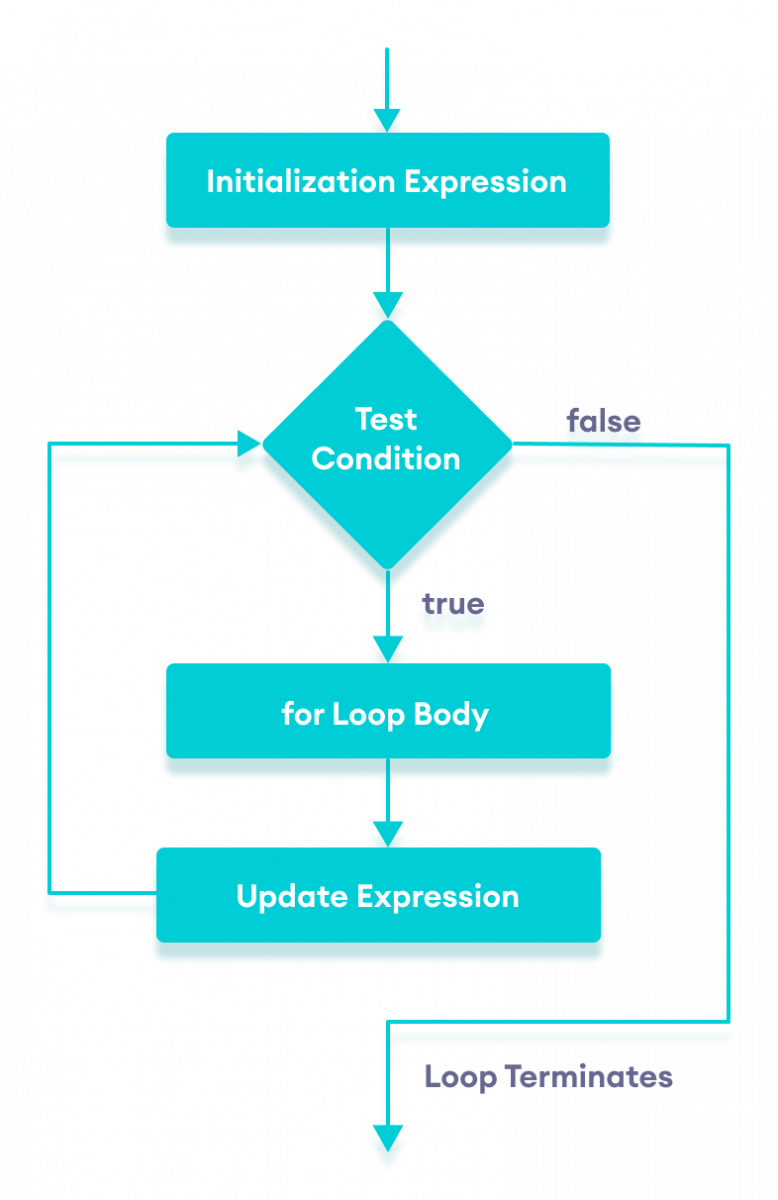
**// body of-loop**

**}**

**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.**

## Flowchart of for Loop in C++



### Ex 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;

}

### Ex 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;

}

### Example 3: Find the sum of first n Natural Numbers

// C++ program to find the sum of first n natural numbers 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 i = 1; i <= num; ++i) {

sum += i;

}

cout << "Sum = " << sum << endl;

return 0;

}

### Infinite for loop

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

**// infinite for loop**

**for(int i = 1; i > 0; i++) {**

**// block of code**

**}**

**break Statement**

In C++, the break statement terminates the loop when it is encountered.

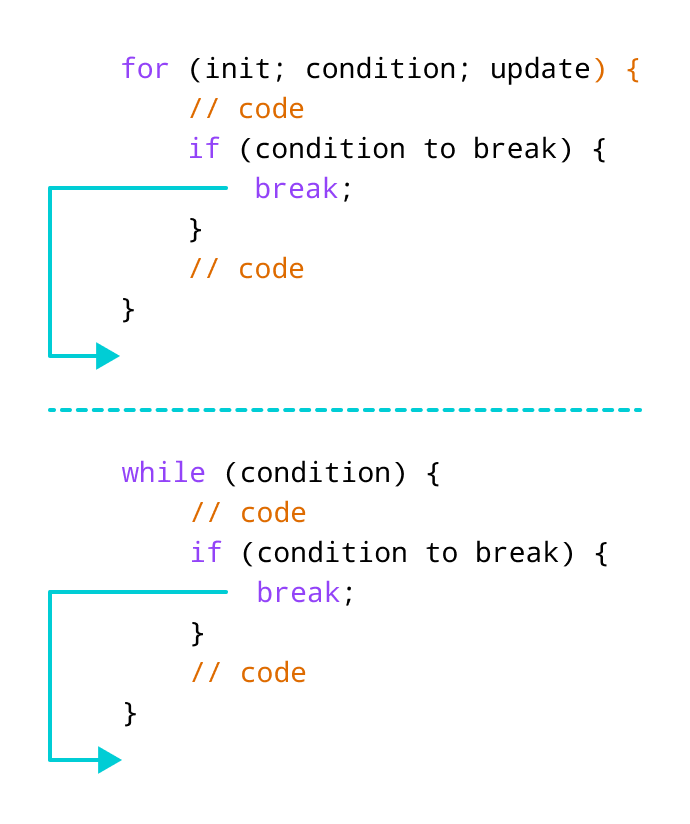
The syntax of the break statement is:

break;

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

* [C++ for loop](https://www.programiz.com/cpp-programming/for-loop)
* [C++ if...else](https://www.programiz.com/cpp-programming/for-loop)
* [C++ while loop](https://www.programiz.com/cpp-programming/do-while-loop)

## Working of C++ break Statement



## Ex 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;

}

## 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;

}

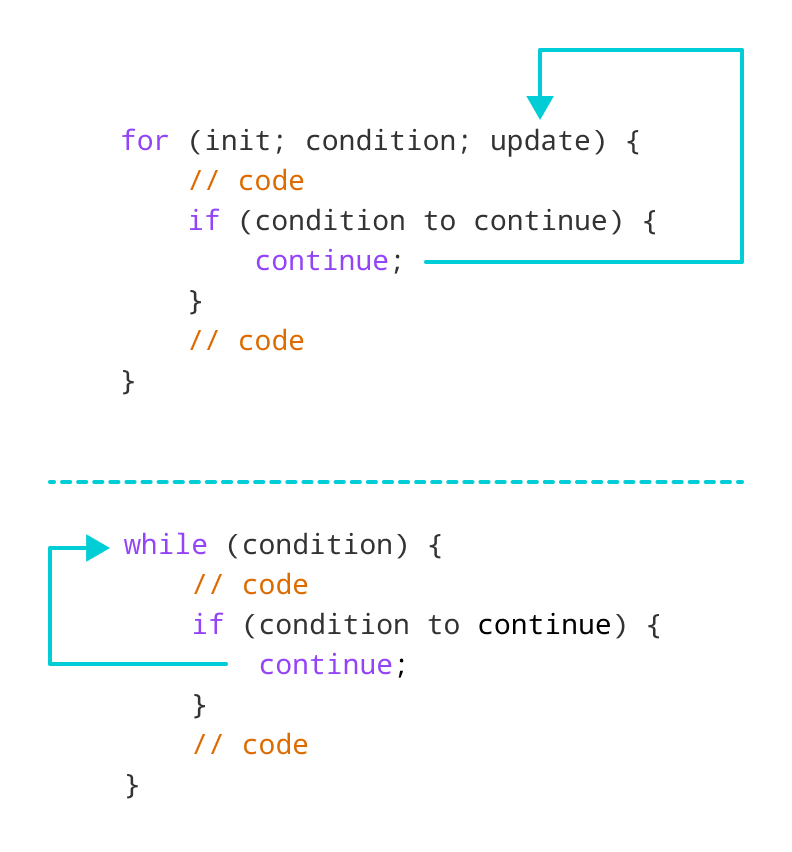
**C++ continue Statement**

In computer programming, the continue statement is used to skip the current iteration of the loop and the control of the program goes to the next iteration.

The syntax of the continue statement is:

continue;

## Working of C++ continue Statement

Working of continue statement in C++

## Example 1: continue with for loop

In a for loop, continue skips the current iteration and the control flow jumps to the update expression.

// program to print the value of i

#include <iostream>

using namespace std;

int main() {

for (int i = 1; i <= 5; i++) {

// condition to continue

if (i == 3) {

continue;

}

cout << i << endl;

}

return 0;

}

## Example 2: continue with while loop

In a while loop, continue skips the current iteration and control flow of the program jumps back to the while condition.

// program to calculate positive numbers till 50 only

// if the user enters a negative number,

// that number is skipped from the calculation

// negative number -> loop terminate

// numbers above 50 -> skip iteration

#include <iostream>

using namespace std;

int main() {

int sum = 0;

int number = 0;

while (number >= 0) {

// add all positive numbers

sum += number;

// take input from the user

cout << "Enter a number: ";

cin >> number;

// continue condition

if (number > 50) {

cout << "The number is greater than 50 and won't be calculated." << endl;

number = 0; // the value of number is made 0 again

continue;

}

}

// display the sum

cout << "The sum is " << sum << endl;

return 0;

}

**goto Statement**

In C++ programming, the goto statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.

## Syntax of goto Statement

goto label;

... .. ...

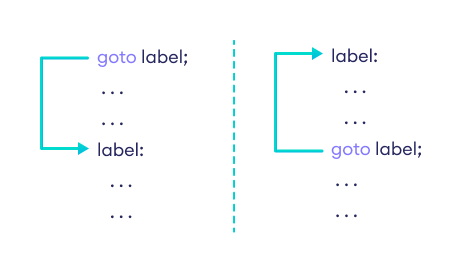
... .. ...

label:

statement;

... .. ...

In the syntax above, label is an identifier. When goto label; is encountered, the control of program jumps to label: and executes the code below it.



### Example: goto Statement

// This program calculates the average of numbers entered by the user. If the user enters a negative number, it ignores the //number and calculates the average number entered before it.

# include <iostream>

using namespace std;

int main()

{

float num, average, sum = 0.0;

int i, n;

cout << "Maximum number of inputs: ";

cin >> n;

for(i = 1; i <= n; ++i)

{

cout << "Enter n" << i << ": ";

cin >> num;

if(num < 0.0)

{

// Control of the program move to jump:

goto jump;

}

sum += num;

}

jump:

average = sum / (i - 1);

cout << "\nAverage = " << average;

return 0;

}

**Arrays**

**In C++, an array is a variable that can store multiple values of the same type. Suppose a class has 50 students, and we need to store the grades of all of them. Instead of creating 50 separate variables, we can simply create an array:**

**double grade[50];**

**Here, grade is an array that can hold a maximum of 50 elements of double type.**

**In C++, the size and type of arrays cannot be changed after its declaration.**

## Array Declaration

**dataType arrayName[arraySize];**

**For example,**

**int x[6];**

**Here,**

* **int - type of element to be stored**
* **x - name of the array**
* **6 - size of the array**

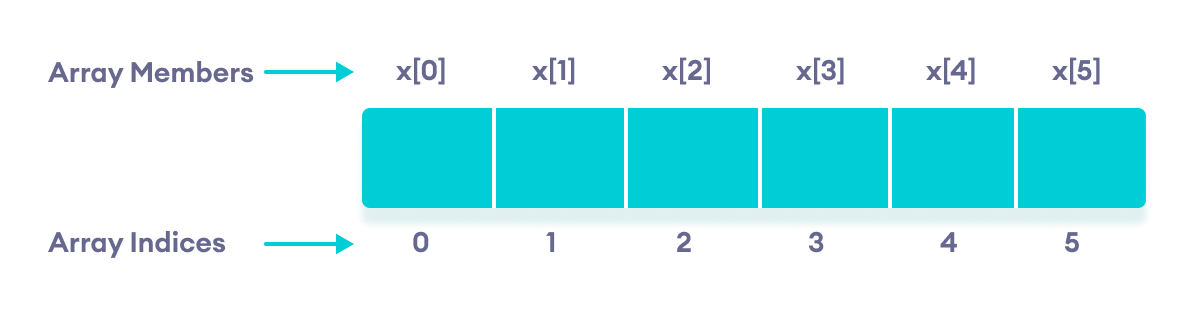
## Access Elements in C++ Array

**In C++, each element in an array is associated with a number. The number is known as an array index. We can access elements of an array by using those indices.**

**// syntax to access array elements**

**array[index];**

**Consider the array x we have seen above. int x[6];**

****

### Note:

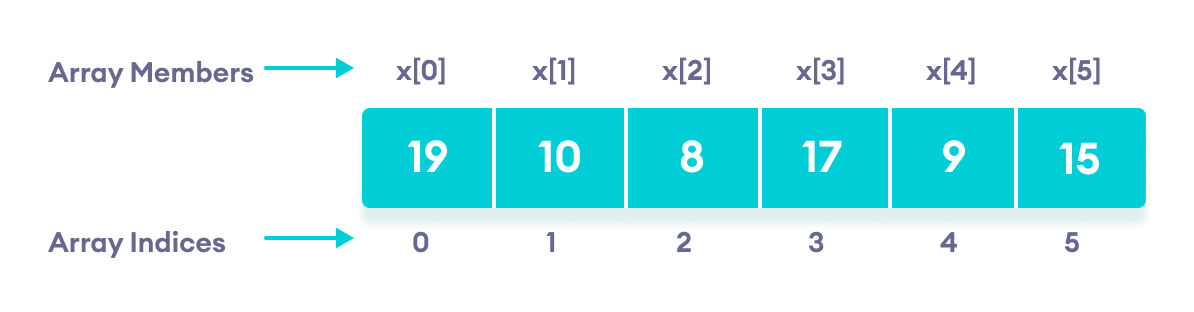
* **The array indices start with 0. Meaning x[0] is the first element stored at index 0.**
* **If the size of an array is n, the last element is stored at index (n-1). In this example, x[5] is the last element.**
* **Elements of an array have consecutive addresses. For example, suppose the starting address of x[0] is 2120.**
* **Then, the address of the next element x[1] will be 2124, the address of x[2] will be 2128, and so on.**
* **Here, the size of each element is increased by 4. This is because the size of int is 4 bytes.**

## Array Initialization

**In C++, it's possible to initialize an array during declaration. For example,**

**// declare and initialize and array**

**int x[6] = {19, 10, 8, 17, 9, 15};**

****

**Another method to initialize array during declaration:**

**// declare and initialize an array**

**int x[] = {19, 10, 8, 17, 9, 15};**

**Here, we have not mentioned the size of the array. In such cases, the compiler automatically computes the size.**

### Array With Empty Members

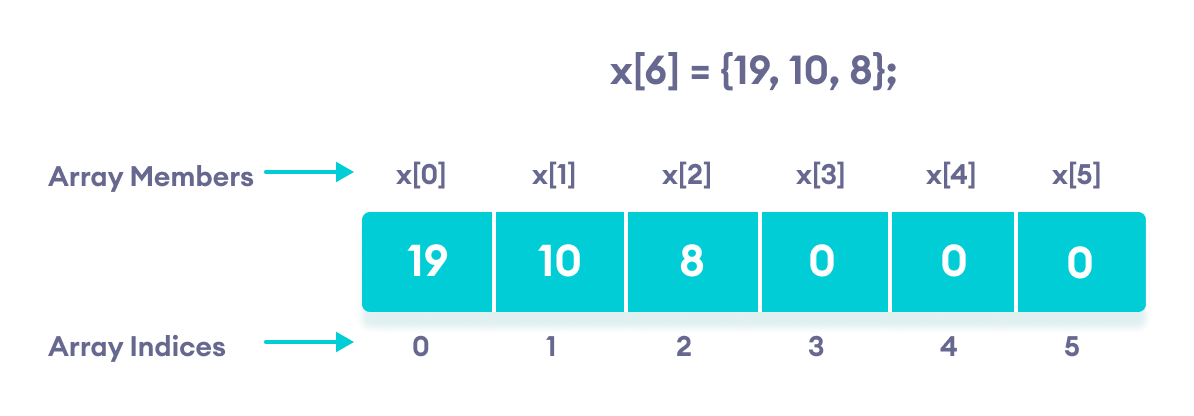
**In C++, if an array has a size n, we can store upto n number of elements in the array. However, what will happen if we store less than n number of elements. For example,**

**// store only 3 elements in the array**

**int x[6] = {19, 10, 8};**

**Here, the array x has a size of 6. However, we have initialized it with only 3 elements.**

**In such cases, the compiler assigns random values to the remaining places. Oftentimes, this random value is sim ply 0.**

****

**Empty array members are automatically assigned the value 0**

## How to insert and print array elements?

**int mark[5] = {19, 10, 8, 17, 9}**

**// change 4th element to 9**

**mark[3] = 9;**

**// take input from the user**

**// store the value at third position**

**cin >> mark[2];**

**// take input from the user**

**// insert at ith position**

**cin >> mark[i-1];**

**// print first element of the array**

**cout << mark[0];**

**// print ith element of the array**

**cout >> mark[i-1];**

## Example 1: Displaying Array Elements

**#include <iostream>**

**using namespace std;**

**int main() {**

**int numbers[5] = {7, 5, 6, 12, 35};**

**cout << "The numbers are: ";**

**// Printing array elements**

**// using range based for loop**

**for (const int &n : numbers) {**

**cout << n << " ";**

**}**

**cout << "\nThe numbers are: ";**

**// Printing array elements**

**// using traditional for loop**

**for (int i = 0; i < 5; ++i) {**

**cout << numbers[i] << " ";**

**}**

**return 0;**

**}**

## Example 2: Take Inputs from User and Store Them in an Array

**#include <iostream>**

**using namespace std;**

**int main() {**

**int numbers[5];**

**cout << "Enter 5 numbers: " << endl;**

**// store input from user to array**

**for (int i = 0; i < 5; ++i) {**

**cin >> numbers[i];**

**}**

**cout << "The numbers are: ";**

**// print array elements**

**for (int n = 0; n < 5; ++n) {**

**cout << numbers[n] << " ";**

**}**

**return 0;**

**}**

## Example 3: Display Sum and Average of Array Elements Using for Loop

**#include <iostream>**

**using namespace std;**

**int main() {**

**// initialize an array without specifying size**

**double numbers[] = {7, 5, 6, 12, 35, 27};**

**double sum = 0;**

**double count = 0;**

**double average;**

**cout << "The numbers are: ";**

**// print array elements**

**// use of range-based for loop**

**for (const double &n : numbers) {**

**cout << n << " ";**

**// calculate the sum**

**sum += n;**

**// count the no. of array elements**

**++count;**

**}**

**// print the sum**

**cout << "\nTheir Sum = " << sum << endl;**

**// find the average**

**average = sum / count;**

**cout << "Their Average = " << average << endl;**

**return 0;**

**}**

**C++ Multidimensional Arrays**

In C++, we can create an [array](https://www.programiz.com/cpp-programming/arrays) of an array, known as a multidimensional array. For example:

**int x[3][4];**

Here, x is a two-dimensional array. It can hold a maximum of 12 elements.

We can think of this array as a table with 3 rows and each row has 4 columns as shown below.



Elements in two-dimensional array in C++ Programming

## Example 1: Two Dimensional Array

// C++ Program to display all elements of an initialised two dimensional array

#include <iostream>

using namespace std;

int main() {

int test[3][2] = {{2, -5},

{4, 0},

{9, 1}};

// use of nested for loop

// access rows of the array

for (int i = 0; i < 3; ++i) {

// access columns of the array

for (int j = 0; j < 2; ++j) {

cout << "test[" << i << "][" << j << "] = " << test[i][j] << endl;

}

}

return 0;

}

## Example 2: Taking Input for Two Dimensional Array

#include <iostream>

using namespace std;

int main() {

int numbers[2][3];

cout << "Enter 6 numbers: " << endl;

// Storing user input in the array

for (int i = 0; i < 2; ++i) {

for (int j = 0; j < 3; ++j) {

cin >> numbers[i][j];

}

}

cout << "The numbers are: " << endl;

// Printing array elements

for (int i = 0; i < 2; ++i) {

for (int j = 0; j < 3; ++j) {

cout << "numbers[" << i << "][" << j << "]: " << numbers[i][j] << endl;

}

}

return 0;

}

## Character String

**This string is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.**

**The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."**

**char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};**

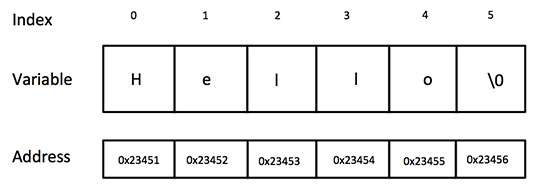
**char greeting[6] = {'H', 'e', 'l', 'l', 'o'};**

**char greeting[] = "Hello";**

**Type II : string hi =”hello”;**

**char greeting[] = "Hello";**

**Following is the memory presentation of above defined string in C/C++ −**

****

**Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string −**

**#include <iostream>**

**using namespace std;**

**int main () {**

**char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};**

**cout << "Greeting message: ";**

**cout << greeting << endl;**

**return 0;**

**}**

**When the above code is compiled and executed, it produces the following result −**

**Greeting message: Hello**

**C++ supports a wide range of functions that manipulate null-terminated strings –**

**#include <cstring>**

|  |  |
| --- | --- |
| **Sr.No** | **Function & Purpose** |
| **1** | **strcpy(s1, s2);**  **Copies string s2 into string s1.** |
| **2** | **strcat(s1, s2);**  **Concatenates string s2 onto the end of string s1.** |
| **3** | **strlen(s1);**  **Returns the length of string s1.** |
| **4** | **strcmp(s1, s2);**  **Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.** |
| **5** | **strchr(s1, ch);**  **Returns a pointer to the first occurrence of character ch in string s1.** |
| **6** | **strstr(s1, s2);**  **Returns a pointer to the first occurrence of string s2 in string s1.** |

**Following example makes use of few of the above-mentioned functions −**

[**Live Demo**](http://tpcg.io/oA6mP3)

**#include <iostream>**

**#include <cstring>**

**using namespace std;**

**int main () {**

**char str1[10] = "Hello";**

**char str2[10] = "World";**

**char str3[10];**

**int len ;**

**// copy str1 into str3**

**strcpy( str3, str1);**

**cout << "strcpy( str3, str1) : " << str3 << endl;**

**// concatenates str1 and str2**

**strcat( str1, str2);**

**cout << "strcat( str1, str2): " << str1 << endl;**

**// total lenghth of str1 after concatenation**

**len = strlen(str1);**

**cout << "strlen(str1) : " << len << endl;**

**return 0;**

**}**

**When the above code is compiled and executed, it produces result something as follows −**

**strcpy( str3, str1) : Hello**

**strcat( str1, str2): HelloWorld**

**strlen(str1) : 10**

## The String Class in C++

**The standard C++ library provides a string class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example −**

[**Live Demo**](http://tpcg.io/upuWFC)

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int main () {**

**string str1 = "Hello";**

**string str2 = "World";**

**string str3;**

**int len ;**

**// copy str1 into str3**

**str3 = str1;**

**cout << "str3 : " << str3 << endl;**

**// concatenates str1 and str2**

**str3 = str1 + str2;**

**cout << "str1 + str2 : " << str3 << endl;**

**// total length of str3 after concatenation**

**len = str3.size();**

**cout << "str3.size() : " << len << endl;**

**return 0;**

**}**

**When the above code is compiled and executed, it produces result something as follows −**

**str3 : Hello**

**str1 + str2 : HelloWorld**

**str3.size() : 10**

\

**Function**

* **A function is a block of code (group of statement) that performs a specific task.**
* **When the function is invoked (called) from any part of the program, it all executes the codes defined in the body of the function.**
* **Ex:**

**Void main()**

**{  
int a, b,c;**

**b=10; c=20;**

**a=b+c;**

**cout<<”the result is: ”<< a;**

**}**

**function syntax**

**return type function name (parameter)**

**{**

**//st.;**

**//st.;**

**}**

**Ex:**

**void suresh(void)**

{

cout<<"hello suresh";

}

Ex:

#include <iostream>

using namespace std;

void greet(int a ,int b); // declaring a function

int main() {

greet(10,20); // calling the function

cout<<"hai";

greet(40,50); // calling the function

return 0;

}

void greet(int a) { //function definition

int s3; s3=a+b; cout<<s3<<endl;

}

**Function Overloading**

wo functions can have the same name if the number and/or type of arguments passed is different.

These functions having the same name but different arguments are known as overloaded functions. For example:

// same name different arguments

int test() { }

int test(int a) { }

float test(double a) { }

int test(int a, double b) { }

Here, all 4 functions are overloaded functions.

Notice that the return types of all these 4 functions are not the same. Overloaded functions may or may not have different return types but they must have different arguments. For example,

// Error code

int test(int a) { }

double test(int b){ }

Here, both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will throw an error.

## Example 1: Overloading Using Different Types of Parameter

// Program to compute absolute value

// Works for both int and float

#include <iostream>

using namespace std;

// function with float type parameter

float absolute(float var){

if (var < 0.0)

var = -var;

return var;

}

// function with int type parameter

int absolute(int var) {

if (var < 0)

var = -var;

return var;

}

int main() {

// call function with int type parameter

cout << "Absolute value of -5 = " << absolute(-5) << endl;

// call function with float type parameter

cout << "Absolute value of 5.5 = " << absolute(5.5f) << endl;

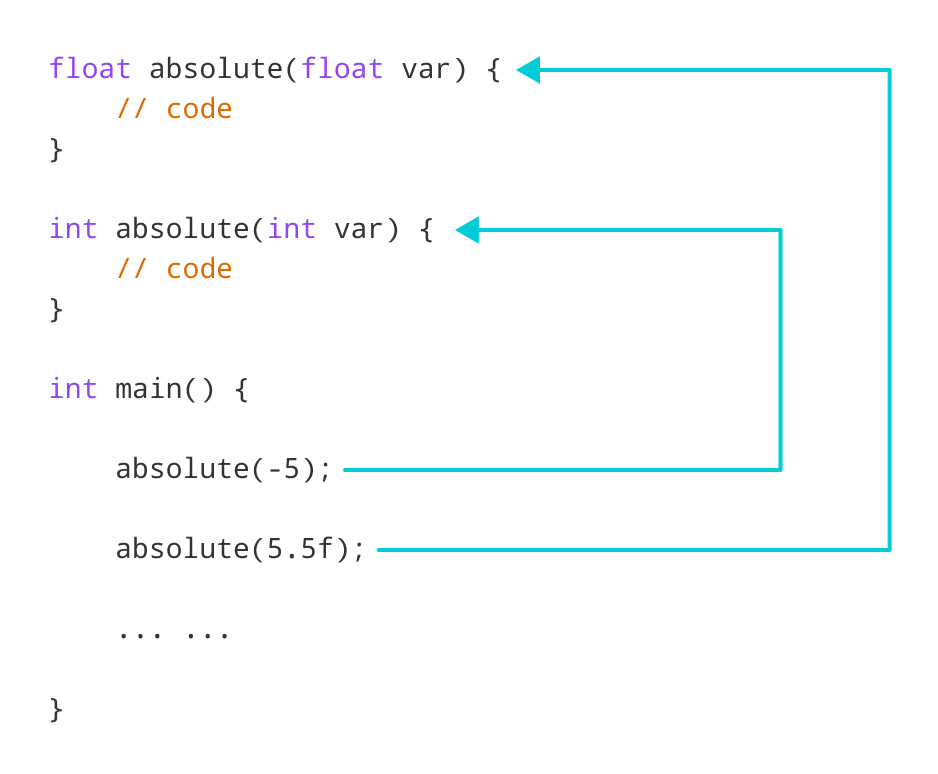
return 0;

}

**Output**

Absolute value of -5 = 5

Absolute value of 5.5 = 5.5



In this program, we overload the absolute() function. Based on the type of parameter passed during the function call, the corresponding function is called.

## Example 2: Overloading Using Different Number of Parameters

#include <iostream>

using namespace std;

// function with 2 parameters

void display(int var1, double var2) {

cout << "Integer number: " << var1;

cout << " and double number: " << var2 << endl;

}

// function with double type single parameter

void display(double var) {

cout << "Double number: " << var << endl;

}

// function with int type single parameter

void display(int var) {

cout << "Integer number: " << var << endl;

}

int main() {

int a = 5;

double b = 5.5;

// call function with int type parameter

display(a);

// call function with double type parameter

display(b);

// call function with 2 parameters

display(a, b);

return 0;

}

[Run Code](https://www.programiz.com/cpp-programming/online-compiler)

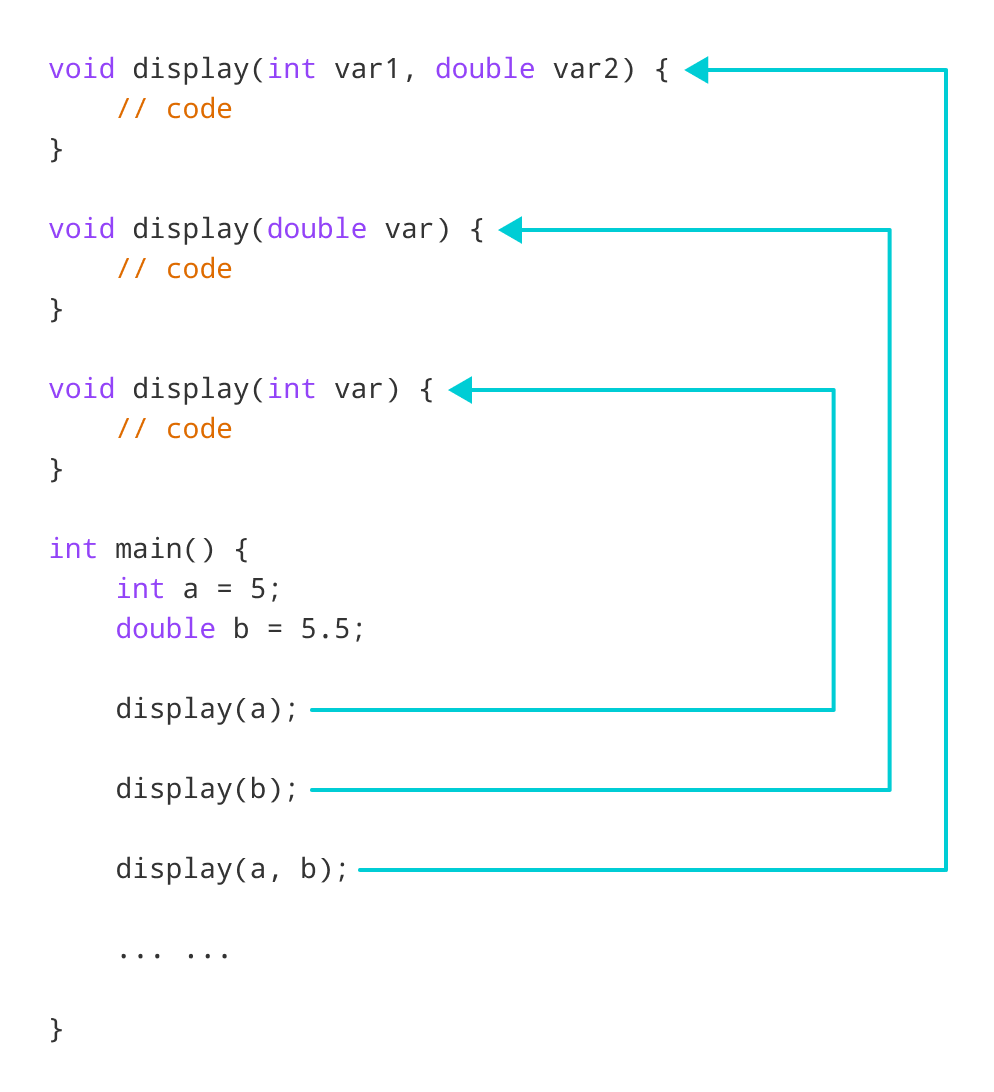
**Output**

Integer number: 5

Float number: 5.5

Integer number: 5 and double number: 5.5

Here, the display() function is called three times with different arguments. Depending on the number and type of arguments passed, the corresponding display() function is called.

Working of overloading for the display() function

The return type of all these functions is the same but that need not be the case for function overloading.

**Note:** In C++, many standard library functions are overloaded. For example, the sqrt() function can take double, float, int, etc. as parameters. This is possible because the sqrt() function is overloaded in C++.

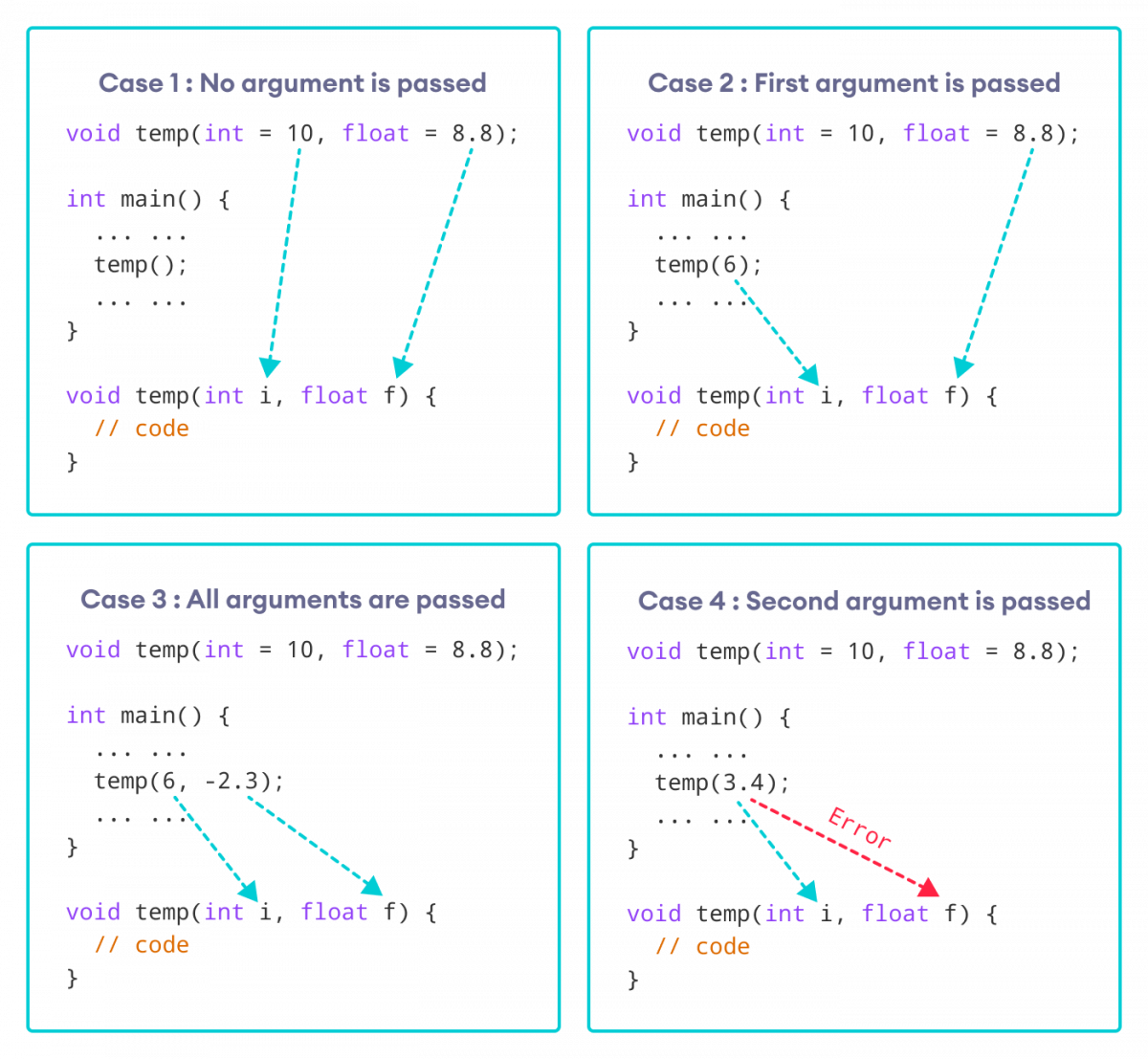
**Default Arguments (Parameters)**

we can provide default values for [function](https://www.programiz.com/cpp-programming/function) parameters.

If a function with default arguments is called without passing arguments, then the default parameters are used.

However, if arguments are passed while calling the function, the default arguments are ignored.

## Working of default arguments

How default arguments work in C++

We can understand the working of default arguments from the image above:

1. When temp() is called, both the default parameters are used by the function.
2. When temp(6) is called, the first argument becomes 6 while the default value is used for the second parameter.
3. When temp(6, -2.3) is called, both the default parameters are overridden, resulting in i = 6 and f = -2.3.
4. When temp(3.4) is passed, the function behaves in an undesired way because the second argument cannot be passed without passing the first argument.  
     
   Therefore, 3.4 is passed as the first argument. Since the first argument has been defined as int, the value that is actually passed is 3.

## Example: Default Argument

#include <iostream>

using namespace std;

// defining the default arguments

void display(char = '\*', int = 3);

int main() {

int count = 5;

cout << "No argument passed: ";

// \*, 3 will be parameters

display();

cout << "First argument passed: ";

// #, 3 will be parameters

display('#');

cout << "Both arguments passed: ";

// $, 5 will be parameters

display('$', count);

return 0;

}

void display(char c, int count) {

for(int i = 1; i <= count; ++i)

{

cout << c;

}

cout << endl;

}

[Run Code](https://www.programiz.com/cpp-programming/online-compiler)

**Output**

No argument passed: \*\*\*

First argument passed: ###

Both arguments passed: $$$$$

Here is how this program works:

1. display() is called without passing any arguments. In this case, display() uses both the default parameters c = '\*' and n = 1.
2. display('#') is called with only one argument. In this case, the first becomes '#'. The second default parameter n = 1 is retained.
3. display('#', count) is called with both arguments. In this case, default arguments are not used.

We can also define the default parameters in the function definition itself. The program below is equivalent to the one above.

#include <iostream>

using namespace std;

// defining the default arguments

void display(char c = '\*', int count = 3) {

for(int i = 1; i <= count; ++i) {

cout << c;

}

cout << endl;

}

int main() {

int count = 5;

cout << "No argument passed: ";

// \*, 3 will be parameters

display();

cout << "First argument passed: ";

// #, 3 will be parameters

display('#');

cout << "Both argument passed: ";

// $, 5 will be parameters

display('$', count);

return 0;

}

[Run Code](https://www.programiz.com/cpp-programming/online-compiler)

## Things to Remember

1. Once we provide a default value for a parameter, all subsequent parameters must also have default values. For example,
2. // Invalid
3. void add(int a, int b = 3, int c, int d);
4. // Invalid
5. void add(int a, int b = 3, int c, int d = 4);
6. // Valid

void add(int a, int c, int b = 3, int d = 4);

1. If we are defining the default arguments in the function definition instead of the function prototype, then the function must be defined before the function call.
2. // Invalid code
3. int main() {
4. // function call
5. display();
6. }
7. void display(char c = '\*', int count = 5) {
8. // code
9. }

**Recursion**

A [function](https://www.programiz.com/cpp-programming/function) that calls itself is known as a recursive function. And, this technique is known as recursion.

## Working of Recursion in C++

void recurse()

{

... .. ...

recurse();

... .. ...

}

int main()

{

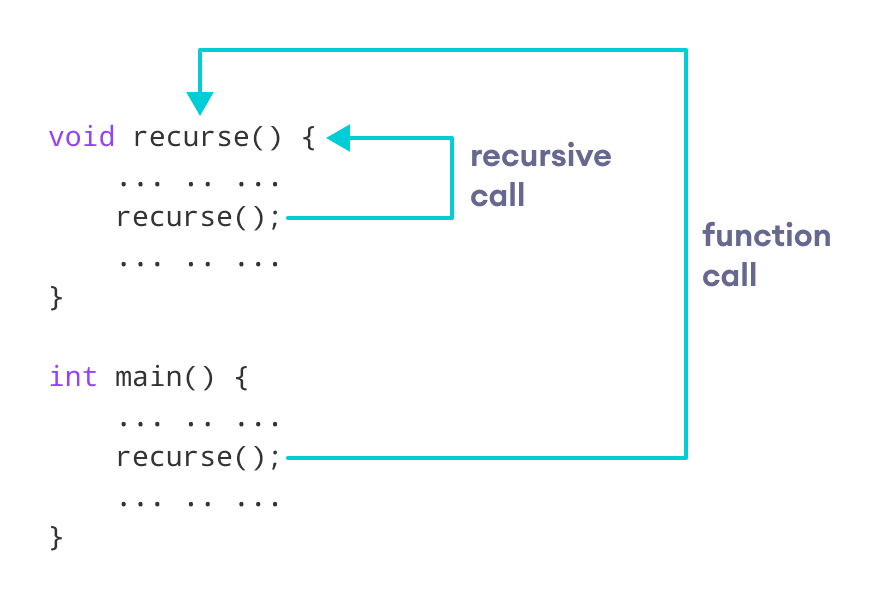
... .. ...

recurse();

... .. ...

}

The figure below shows how recursion works by calling itself over and over again.



The recursion continues until some condition is met. To prevent infinite recursion, [if...else statement](https://www.programiz.com/cpp-programming/if-else) (or similar approach) can be used where one branch makes the recursive call and the other doesn't.

## Example 1: Factorial of a Number Using Recursion

// Factorial of n = 1\*2\*3\*...\*n

#include <iostream>

using namespace std;

int factorial(int);

int main() {

int n, result;

cout << "Enter a non-negative number: ";

cin >> n;

result = factorial(n);

cout << "Factorial of " << n << " = " << result;

return 0;

}

int factorial(int n) {

if (n > 1) {

return n \* factorial(n - 1);

} else {

return 1;

}

}

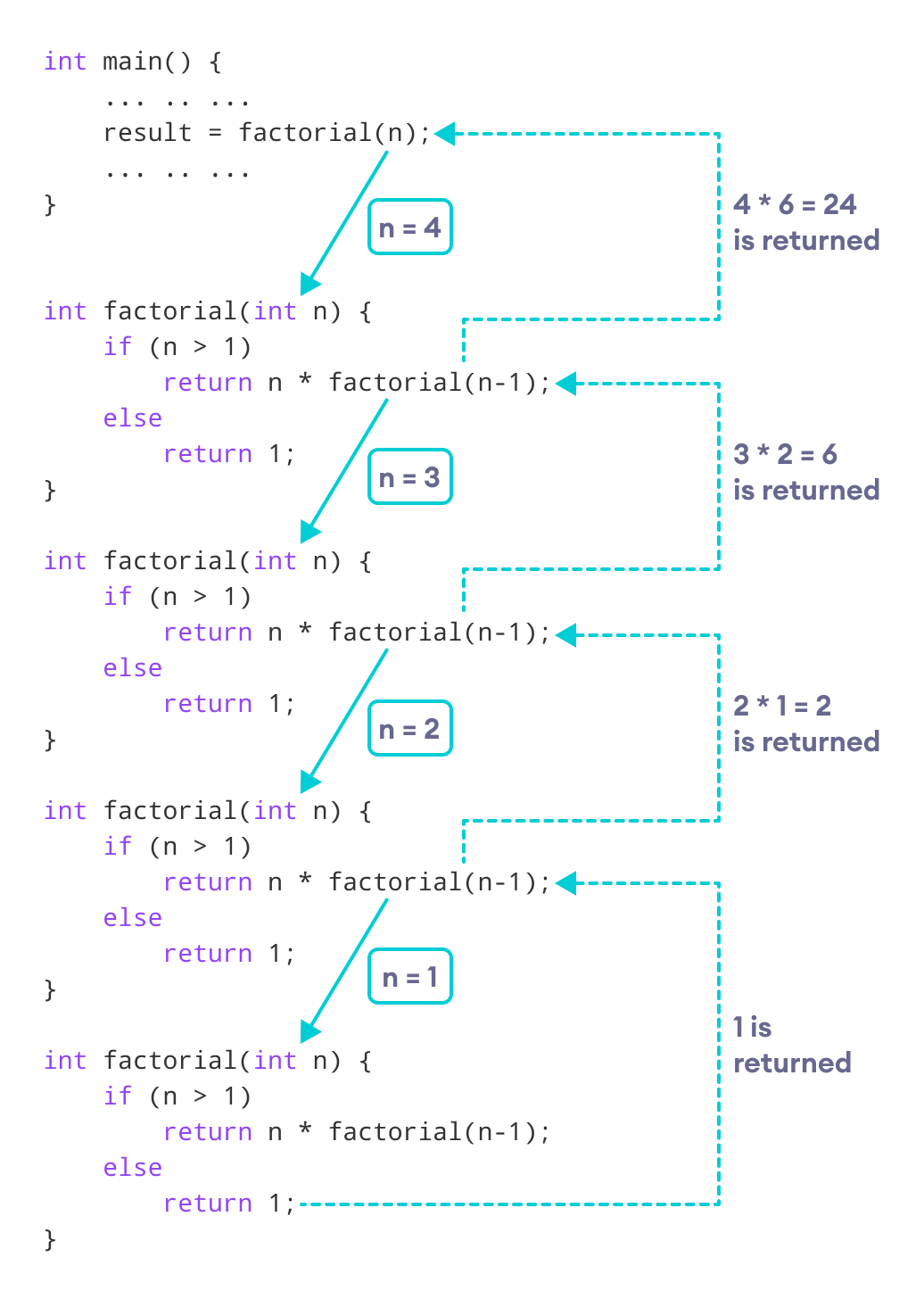
[Run Code](https://www.programiz.com/cpp-programming/online-compiler)

**Output**

Enter a non-negative number: 4

Factorial of 4 = 24

### Working of Factorial Program



As we can see, the factorial() function is calling itself. However, during each call, we have decreased the value of n by 1. When n is less than 1, the factorial() function ultimately returns the output.

### Advantages of C++ Recursion

* It makes our code shorter and cleaner.
* Recursion is required in problems concerning data structures and advanced algorithms, such as Graph and Tree Traversal.

### Disadvantages of C++ Recursion

* It takes a lot of stack space compared to an iterative program.
* It uses more processor time.
* It can be more difficult to debug compared to an equivalent iterative program.

**Return by Reference**

In C++ Programming, not only can you pass values by reference to a [function](https://www.programiz.com/cpp-programming/function) but you can also return a value by reference.

## Example: Return by Reference

#include <iostream>

using namespace std;

// global variable

int num;

// function declaration

int& test();

int main() {

  // assign 5 to num variable

  // equivalent to num = 5;

test() = 5;

cout << num;

return 0;

}

// function definition

// returns the address of num variable

int& test() {

return num;

}

**Output**

5

In program above, the return type of function test() is int&. Hence, this function returns a reference of the variable num.

The return statement is return num;. Unlike return by value, this statement doesn't return value of num, instead it returns the variable itself (address).

So, when the **variable** is returned, it can be assigned a value as done in test() = 5;

This stores **5** to the variable num, which is displayed onto the screen.

### Important Things to Remember When Returning by Reference.

* Ordinary function returns value but this function doesn't. Hence, you cannot return a constant from the function.

int& test() {

return 2;

}

* You cannot return a local variable from this function.

int& test() {

int n = 2;

return n;

}

**Wildcard Pattern Matching**

### Description

**The C++ wildcard pattern matching library is a simple to use and very efficient wildcard matching/globbing library. The library supports the following wildcard options:**

| **Wildcard** | **Meaning** |
| --- | --- |
| \* | Match zero or more characters |
| ? | Match exactly one character |

**Given a text and a wildcard pattern, implement wildcard pattern matching algorithm that finds if wildcard pattern is matched with text. The matching should cover the entire text (not partial text). The wildcard pattern can include the characters ‘?’ and ‘\*’**

* **‘?’ – matches any single character**
* **‘\*’ – Matches any sequence of characters (including the empty sequence)**

**For example:**

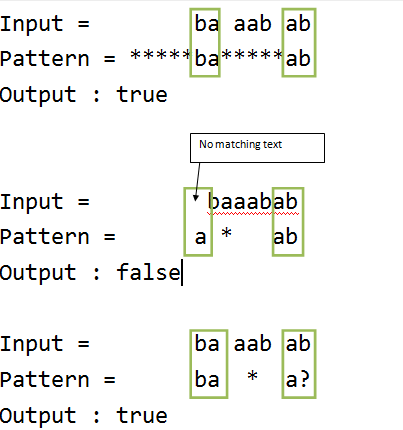
**Text = "baaabab",**

**Pattern = “\*\*\*\*\*ba\*\*\*\*\*ab", output : true**

**Pattern = "baaa?ab", output : true**

**Pattern = "ba\*a?", output : true**

**Pattern = "a\*ab", output : false**



Each occurrence of ‘?’ character in wildcard pattern can be replaced with any other character and each occurrence of ‘\*’ with a sequence of characters such that the wildcard pattern becomes identical to the input string after replacement.

Let’s consider any character in the pattern.

**Case 1: The character is ‘\*’** . Here two cases arises as follows:

1. We can ignore ‘\*’ character and move to next character in the Pattern.
2. ‘\*’ character matches with one or more characters in Text. Here we will move to next character in the string.

**Case 2: The character is ‘?’**   
We can ignore current character in Text and move to next character in the Pattern and Text.

**Case 3: The character is not a wildcard character**   
If current character in Text matches with current character in Pattern, we move to next character in the Pattern and Text. If they do not match, wildcard pattern and Text do not match.  
We can use Dynamic Programming to solve this problem:

Let **T[i][j]** is true if first i characters in given string matches the first j characters of pattern.

**POINTER**

C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined −

#include <iostream>

using namespace std;

int main () {

int var1;

char var2[10];

cout << "Address of var1 variable: ";

cout << &var1 << endl;

cout << "Address of var2 variable: ";

cout << &var2 << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Address of var1 variable: 0xbfebd5c0

Address of var2 variable: 0xbfebd5b6

## What are Pointers?

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is −

type \*var-name;

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration −

int \*ip; // pointer to an integer

double \*dp; // pointer to a double

float \*fp; // pointer to a float

char \*ch // pointer to character

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

## Using Pointers in C++

There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations −

#include <iostream>

using namespace std;

int main () {

int var = 20; // actual variable declaration.

int \*ip; // pointer variable

ip = &var; // store address of var in pointer variable

cout << "Value of var variable: " << var << endl;

cout<< “Address of var variable: “<<&var<< endl;

// print the address stored in ip pointer variable

cout << "Address stored in ip variable: " << ip << endl;

cout<< “Address of ip variable: “<< &ip<<endl;

// access the value at the address available in pointer

cout << "Value of \*ip variable: ";

cout << \*ip << endl;

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

Value of var variable: 20

Address of var variable: 0xbfc601ac

Address stored in ip variable: 0xbfc601ac

Address of ip variable: 0xbfc601b0

Value of \*ip variable: 20

## Pointers in C++

Pointers have many but easy concepts and they are very important to C++ programming. There are following few important pointer concepts which should be clear to a C++ programmer −

|  |  |
| --- | --- |
| **Sr.No** | **Concept & Description** |
| 1 | [Null Pointers](https://www.tutorialspoint.com/cplusplus/cpp_null_pointers.htm)  C++ supports null pointer, which is a constant with a value of zero defined in sev  eral standard libraries. |
| 2 | [Pointer Arithmetic](https://www.tutorialspoint.com/cplusplus/cpp_pointer_arithmatic.htm)There are four arithmetic operators that can be used on pointers: ++, --, +, - |
| 3 | [Pointers vs Arrays](https://www.tutorialspoint.com/cplusplus/cpp_pointers_vs_arrays.htm)  There is a close relationship between pointers and arrays. |
| 4 | [Array of Pointers](https://www.tutorialspoint.com/cplusplus/cpp_array_of_pointers.htm)  You can define arrays to hold a number of pointers. |
| 5 | [Pointer to Pointer](https://www.tutorialspoint.com/cplusplus/cpp_pointer_to_pointer.htm)  C++ allows you to have pointer on a pointer and so on. |
| 6 | [Passing Pointers to Functions](https://www.tutorialspoint.com/cplusplus/cpp_passing_pointers_to_functions.htm)  Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function. |
| 7 | [Return Pointer from Functions](https://www.tutorialspoint.com/cplusplus/cpp_return_pointer_from_functions.htm)  C++ allows a function to return a pointer to local variable , static variable and dynamically allocated memory as well.0 |

**POINTER TO POINTER**

A pointer to a pointer is a form of multiple indirection or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int −

int \*\*var;

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example −

#include <iostream>

using namespace std;

int main () {

int var;

int \*ptr;

int \*\*pptr;

var = 3000;

// take the address of var

ptr = &var;

// take the address of ptr using address of operator &

pptr = &ptr;

// take the value using pptr

cout << "Value of var :" << var << endl;

cout << "Value available at \*ptr :" << \*ptr << endl;

cout << "Value available at \*\*pptr :" << \*\*pptr << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Value of var :3000

Value available at \*ptr :3000

Value available at \*\*pptr :3000

**POINTER ARITHMETIC**

consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer −

ptr++

the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer. This operation will move the pointer to next memory location without impacting actual value at the memory location. If ptr points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

## Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array −

[Live Demo](http://tpcg.io/g0pNAF)

#include <iostream>

using namespace std;

const int MAX = 3;

int main () {

int var[MAX] = {10, 100, 200};

int \*ptr;

// let us have array address in pointer.

ptr = var;

for (int i = 0; i < MAX; i++) {

cout << "Address of var[" << i << "] = ";

cout << ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// point to the next location

ptr++;

}

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

Address of var[0] = 0xbfa088b0

Value of var[0] = 10

Address of var[1] = 0xbfa088b4

Value of var[1] = 100

Address of var[2] = 0xbfa088b8

Value of var[2] = 200

## Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below −

[Live Demo](http://tpcg.io/ujgz9K)

#include <iostream>

using namespace std;

const int MAX = 3;

int main () {

int var[MAX] = {10, 100, 200};

int \*ptr;

// let us have address of the last element in pointer.

ptr = &var[MAX-1];

for (int i = MAX; i > 0; i--) {

cout << "Address of var[" << i << "] = ";

cout << ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// point to the previous location

ptr--;

}

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

Address of var[3] = 0xbfdb70f8

Value of var[3] = 200

Address of var[2] = 0xbfdb70f4

Value of var[2] = 100

Address of var[1] = 0xbfdb70f0

Value of var[1] = 10

## Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1] −

[Live Demo](http://tpcg.io/T8eHu9)

#include <iostream>

using namespace std;

const int MAX = 3;

int main () {

int var[MAX] = {10, 100, 200};

int \*ptr;

// let us have address of the first element in pointer.

ptr = var;

int i = 0;

while ( ptr <= &var[MAX - 1] ) {

cout << "Address of var[" << i << "] = ";

cout << ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// point to the previous location

ptr++;

i++;

}

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

Address of var[0] = 0xbfce42d0

Value of var[0] = 10

Address of var[1] = 0xbfce42d4

Value of var[1] = 100

Address of var[2] = 0xbfce42d8

Value of var[2] = 200

**Pointers vs Arrays**

Pointers and arrays are strongly related. In fact, pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing. Consider the following program −

[Live Demo](http://tpcg.io/Z3TAMf)

#include <iostream>

using namespace std;

const int MAX = 3;

int main () {

int var[MAX] = {10, 100, 200};

int \*ptr;

// let us have array address in pointer.

ptr = var;

for (int i = 0; i < MAX; i++) {

cout << "Address of var[" << i << "] = ";

cout << ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// point to the next location

ptr++;

}

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

Address of var[0] = 0xbfa088b0

Value of var[0] = 10

Address of var[1] = 0xbfa088b4

Value of var[1] = 100

Address of var[2] = 0xbfa088b8

Value of var[2] = 200

**Array of Pointers**

#include <iostream>

using namespace std;

const int MAX = 3;

int main () {

int var[MAX] = {10, 100, 200};

for (int i = 0; i < MAX; i++) {

cout << "Value of var[" << i << "] = ";

cout << var[i] << endl;

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200

There may be a situation, when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer −

int \*ptr[MAX];

This declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers which will be stored in an array of pointers as follows −

[Live Demo](http://tpcg.io/qgpsAZ)

#include <iostream>

using namespace std;

const int MAX = 3;

int main () {

int var[MAX] = {10, 100, 200};

int \*ptr[MAX];

for (int i = 0; i < MAX; i++) {

ptr[i] = &var[i]; // assign the address of integer.

}

for (int i = 0; i < MAX; i++) {

cout << "Value of var[" << i << "] = ";

cout << \*ptr[i] << endl;

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200

You can also use an array of pointers to character to store a list of strings as follows −

[Live Demo](http://tpcg.io/0kooHI)

#include <iostream>

using namespace std;

const int MAX = 4;

int main () {

const char \*names[MAX] = { "Zara Ali", "Hina Ali", "Nuha Ali", "Sara Ali" };

for (int i = 0; i < MAX; i++) {

cout << "Value of names[" << i << "] = ";

cout << (names + i) << endl;

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Value of names[0] = 0x7ffd256683c0

Value of names[1] = 0x7ffd256683c8

Value of names[2] = 0x7ffd256683d0

Value of names[3] = 0x7ffd256683d8

## Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows −

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

## Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box −

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Both of the objects Box1 and Box2 will have their own copy of data members.

## Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear −

[Live Demo](http://tpcg.io/JeEWd6)

#include <iostream>

using namespace std;

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.height = 5.0;

Box1.length = 6.0;

Box1.breadth = 7.0;

// box 2 specification

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// volume of box 1

volume = Box1.height \* Box1.length \* Box1.breadth;

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.height \* Box2.length \* Box2.breadth;

cout << "Volume of Box2 : " << volume <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Volume of Box1 : 210

Volume of Box2 : 1560

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

## Classes and Objects in Detail

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below −

|  |  |
| --- | --- |
| **Sr.No** | **Concept & Description** |
| 1 | [Class Member Functions](https://www.tutorialspoint.com/cplusplus/cpp_class_member_functions.htm)  A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. |
| 2 | [Class Access Modifiers](https://www.tutorialspoint.com/cplusplus/cpp_class_access_modifiers.htm)  A class member can be defined as public, private or protected. By default members would be assumed as private. |
| 3 | [Constructor & Destructor](https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm)  A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted. |
| 4 | [Copy Constructor](https://www.tutorialspoint.com/cplusplus/cpp_copy_constructor.htm)  The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. |
| 5 | [Friend Functions](https://www.tutorialspoint.com/cplusplus/cpp_friend_functions.htm)  A **friend** function is permitted full access to private and protected members of a class. |
| 6 | [Inline Functions](https://www.tutorialspoint.com/cplusplus/cpp_inline_functions.htm)  With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function. |
| 7 | [this Pointer](https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm)  Every object has a special pointer **this** which points to the object itself. |
| 8 | [Pointer to C++ Classes](https://www.tutorialspoint.com/cplusplus/cpp_pointer_to_class.htm)  A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it. |
| 9 | [Static Members of a Class](https://www.tutorialspoint.com/cplusplus/cpp_static_members.htm)  Both data members and function members of a class can be declared as static. |

Class Member Functions

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Let us take previously defined class to access the members of the class using a member function instead of directly accessing them −

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

double getVolume(void);// Returns box volume

};

Member functions can be defined within the class definition or separately using **scope resolution operator, :** −. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define **Volume()** function as below −

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

double getVolume(void) {

return length \* breadth \* height;

}

};

If you like, you can define the same function outside the class using the **scope resolution operator** (::) as follows −

double Box::getVolume(void) {

return length \* breadth \* height;

}

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (**.**) on a object where it will manipulate data related to that object only as follows −

Box myBox; // Create an object

myBox.getVolume(); // Call member function for the object

Let us put above concepts to set and get the value of different class members in a class −

[Live Demo](http://tpcg.io/ok5zpD)

#include <iostream>

using namespace std;

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

// Member functions declaration

double getVolume(void);

void setLength( double len );

void setBreadth( double bre );

void setHeight( double hei );

};

// Member functions definitions

double Box::getVolume(void) {

return length \* breadth \* height;

}

void Box::setLength( double len ) {

length = len;

}

void Box::setBreadth( double bre ) {

breadth = bre;

}

void Box::setHeight( double hei ) {

height = hei;

}

// Main function for the program

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Volume of Box1 : 210

Volume of Box2 : 1560

Class Access Modifiers

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the labeled **public, private,** and **protected** sections within the class body. The keywords public, private, and protected are called access specifiers.

A class can have multiple public, protected, or private labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is private.

class Base {

public:

// public members go here

protected:

// protected members go here

private:

// private members go here

};

## The public Members

A **public** member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function as shown in the following example −

[Live Demo](http://tpcg.io/eLDaBs)

#include <iostream>

using namespace std;

class Line {

public:

double length;

void setLength( double len );

double getLength( void );

};

// Member functions definitions

double Line::getLength(void) {

return length ;

}

void Line::setLength( double len) {

length = len;

}

// Main function for the program

int main() {

Line line;

// set line length

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

// set line length without member function

line.length = 10.0; // OK: because length is public

cout << "Length of line : " << line.length <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Length of line : 6

Length of line : 10

## The private Members

A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.

By default all the members of a class would be private, for example in the following class **width** is a private member, which means until you label a member, it will be assumed a private member −

class Box {

double width;

public:

double length;

void setWidth( double wid );

double getWidth( void );

};

Practically, we define data in private section and related functions in public section so that they can be called from outside of the class as shown in the following program.

[Live Demo](http://tpcg.io/OJaPdZ)

#include <iostream>

using namespace std;

class Box {

public:

double length;

void setWidth( double wid );

double getWidth( void );

private:

double width;

};

// Member functions definitions

double Box::getWidth(void) {

return width ;

}

void Box::setWidth( double wid ) {

width = wid;

}

// Main function for the program

int main() {

Box box;

// set box length without member function

box.length = 10.0; // OK: because length is public

cout << "Length of box : " << box.length <<endl;

// set box width without member function

// box.width = 10.0; // Error: because width is private

box.setWidth(10.0); // Use member function to set it.

cout << "Width of box : " << box.getWidth() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Length of box : 10

Width of box : 10

## The protected Members

A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

You will learn derived classes and inheritance in next chapter. For now you can check following example where I have derived one child class **SmallBox** from a parent class **Box**.

Following example is similar to above example and here **width** member will be accessible by any member function of its derived class SmallBox.

[Live Demo](http://tpcg.io/wVR5ie)

#include <iostream>

using namespace std;

class Box {

protected:

double width;

};

class SmallBox:Box { // SmallBox is the derived class.

public:

void setSmallWidth( double wid );

double getSmallWidth( void );

};

// Member functions of child class

double SmallBox::getSmallWidth(void) {

return width ;

}

void SmallBox::setSmallWidth( double wid ) {

width = wid;

}

// Main function for the program

int main() {

SmallBox box;

// set box width using member function

box.setSmallWidth(5.0);

cout << "Width of box : "<< box.getSmallWidth() << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Width of box : 5

**Constructor and Destructor**

## The Class Constructor

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explains the concept of constructor −

[Live Demo](http://tpcg.io/Qaz3NN)

#include <iostream>

using namespace std;

class Line {

public:

void setLength( double len );

double getLength( void );

Line(); // This is the constructor

private:

double length;

};

// Member functions definitions including constructor

Line::Line(void) {

cout << "Object is being created" << endl;

}

void Line::setLength( double len ) {

length = len;

}

double Line::getLength( void ) {

return length;

}

// Main function for the program

int main() {

Line line;

// set line length

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Object is being created

Length of line : 6

## Parameterized Constructor

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example −

[Live Demo](http://tpcg.io/AzaYgr)

#include <iostream>

using namespace std;

class Line {

public:

void setLength( double len );

double getLength( void );

Line(double len); // This is the constructor

private:

double length;

};

// Member functions definitions including constructor

Line::Line( double len) {

cout << "Object is being created, length = " << len << endl;

length = len;

}

void Line::setLength( double len ) {

length = len;

}

double Line::getLength( void ) {

return length;

}

// Main function for the program

int main() {

Line line(10.0);

// get initially set length.

cout << "Length of line : " << line.getLength() <<endl;

// set line length again

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Object is being created, length = 10

Length of line : 10

Length of line : 6

## Using Initialization Lists to Initialize Fields

In case of parameterized constructor, you can use following syntax to initialize the fields −

Line::Line( double len): length(len) {

cout << "Object is being created, length = " << len << endl;

}

Above syntax is equal to the following syntax −

Line::Line( double len) {

cout << "Object is being created, length = " << len << endl;

length = len;

}

If for a class C, you have multiple fields X, Y, Z, etc., to be initialized, then use can use same syntax and separate the fields by comma as follows −

C::C( double a, double b, double c): X(a), Y(b), Z(c) {

....

}

## The Class Destructor

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor −

[Live Demo](http://tpcg.io/z57ZJK)

#include <iostream>

using namespace std;

class Line {

public:

void setLength( double len );

double getLength( void );

Line(); // This is the constructor declaration

~Line(); // This is the destructor: declaration

private:

double length;

};

// Member functions definitions including constructor

Line::Line(void) {

cout << "Object is being created" << endl;

}

Line::~Line(void) {

cout << "Object is being deleted" << endl;

}

void Line::setLength( double len ) {

length = len;

}

double Line::getLength( void ) {

return length;

}

// Main function for the program

int main() {

Line line;

// set line length

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Object is being created

Length of line : 6

Object is being deleted

Copy Constructor

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to −

* Initialize one object from another of the same type.
* Copy an object to pass it as an argument to a function.
* Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one.If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here −

classname (const classname &obj) {

// body of constructor

}

Here, **obj** is a reference to an object that is being used to initialize another object.

[Live Demo](http://tpcg.io/8hAV1i)

#include <iostream>

using namespace std;

class Line {

public:

int getLength( void );

Line( int len ); // simple constructor

Line( const Line &obj); // copy constructor

~Line(); // destructor

private:

int \*ptr;

};

// Member functions definitions including constructor

Line::Line(int len) {

cout << "Normal constructor allocating ptr" << endl;

// allocate memory for the pointer;

ptr = new int;

\*ptr = len;

}

Line::Line(const Line &obj) {

cout << "Copy constructor allocating ptr." << endl;

ptr = new int;

\*ptr = \*obj.ptr; // copy the value

}

Line::~Line(void) {

cout << "Freeing memory!" << endl;

delete ptr;

}

int Line::getLength( void ) {

return \*ptr;

}

void display(Line obj) {

cout << "Length of line : " << obj.getLength() <<endl;

}

// Main function for the program

int main() {

Line line(10);

display(line);

return 0;

}

When the above code is compiled and executed, it produces the following result −

Normal constructor allocating ptr

Copy constructor allocating ptr.

Length of line : 10

Freeing memory!

Freeing memory!

Let us see the same example but with a small change to create another object using existing object of the same type −

[Live Demo](http://tpcg.io/PZtS0u)

#include <iostream>

using namespace std;

class Line {

public:

int getLength( void );

Line( int len ); // simple constructor

Line( const Line &obj); // copy constructor

~Line(); // destructor

private:

int \*ptr;

};

// Member functions definitions including constructor

Line::Line(int len) {

cout << "Normal constructor allocating ptr" << endl;

// allocate memory for the pointer;

ptr = new int;

\*ptr = len;

}

Line::Line(const Line &obj) {

cout << "Copy constructor allocating ptr." << endl;

ptr = new int;

\*ptr = \*obj.ptr; // copy the value

}

Line::~Line(void) {

cout << "Freeing memory!" << endl;

delete ptr;

}

int Line::getLength( void ) {

return \*ptr;

}

void display(Line obj) {

cout << "Length of line : " << obj.getLength() <<endl;

}

// Main function for the program

int main() {

Line line1(10);

Line line2 = line1; // This also calls copy constructor

display(line1);

display(line2);

return 0;

}

When the above code is compiled and executed, it produces the following result −

Normal constructor allocating ptr

Copy constructor allocating ptr.

Copy constructor allocating ptr.

Length of line : 10

Freeing memory!

Copy constructor allocating ptr.

Length of line : 10

Freeing memory!

Freeing memory!

Freeing memory!

**Friend Functions**

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows −

class Box {

double width;

public:

double length;

friend void printWidth( Box box );

void setWidth( double wid );

};

To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne −

friend class ClassTwo;

Consider the following program −

[Live Demo](http://tpcg.io/rcFdmm)

#include <iostream>

using namespace std;

class Box {

double width;

public:

friend void printWidth( Box box );

void setWidth( double wid );

};

// Member function definition

void Box::setWidth( double wid ) {

width = wid;

}

// Note: printWidth() is not a member function of any class.

void printWidth( Box box ) {

/\* Because printWidth() is a friend of Box, it can

directly access any member of this class \*/

cout << "Width of box : " << box.width <<endl;

}

// Main function for the program

int main() {

Box box;

// set box width without member function

box.setWidth(10.0);

// Use friend function to print the wdith.

printWidth( box );

return 0;

}

When the above code is compiled and executed, it produces the following result −

Width of box : 10

**Inline Functions**

**inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline**specifier.

Following is an example, which makes use of inline function to return max of two numbers −

[Live Demo](http://tpcg.io/O6tJUp)

#include <iostream>

using namespace std;

inline int Max(int x, int y) {

return (x > y)? x : y;

}

// Main function for the program

int main() {

cout << "Max (20,10): " << Max(20,10) << endl;

cout << "Max (0,200): " << Max(0,200) << endl;

cout << "Max (100,1010): " << Max(100,1010) << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Max (20,10): 20

Max (0,200): 200

Max (100,1010): 1010

**Data Encapsulation in C++**

All C++ programs are composed of the following two fundamental elements −

* **Program statements (code)** − This is the part of a program that performs actions and they are called functions.
* **Program data** − The data is the information of the program which gets affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

**Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected**and **public** members. By default, all items defined in a class are private. For example −

class Box {

public:

double getVolume(void) {

return length \* breadth \* height;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

## Data Encapsulation Example

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example −

#include <iostream>

using namespace std;

class Adder {

public:

// constructor

Adder(int i = 0) {

total = i;

}

// interface to outside world

void addNum(int number) {

total += number;

}

// interface to outside world

int getTotal() {

return total;

};

private:

// hidden data from outside world

int total;

};

int main() {

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Total 60

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal**are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

## Designing Strategy

Most of us have learnt to make class members private by default unless we really need to expose them. That's just good **encapsulation**.

This is applied most frequently to data members, but it applies equally to all members, including virtual functions.