**c++ notes unit-1**

**History of C++**

The C++ programming language has a history going back to 1979, when [Bjarne Stroustrup](http://www2.research.att.com/~bs/) was doing work for his Ph.D. thesis. One of the languages Stroustrup had the opportunity to work with was a language called Simula, which as the name implies is a language primarily designed for simulations. [The Simula 67 language](http://staff.um.edu.mt/jskl1/talk.html) - which was the variant that Stroustrup worked with - is regarded as the first language to support the object-oriented programming paradigm. Stroustrup found that this paradigm was very useful for software development, however the Simula language was far too slow for practical use.  
  
Shortly thereafter, he began work on "C with Classes", which as the name implies was meant to be a superset of the C language. His goal was to add object-oriented programming into the C language, which was and still is a language well-respected for its portability without sacrificing speed or low-level functionality. His language included [classes](http://www.cplusplus.com/doc/tutorial/classes/), basic [inheritance](http://www.cplusplus.com/doc/tutorial/inheritance/#inheritance), [inlining](http://www.cplusplus.com/doc/tutorial/functions2/" \l "inline), [default function arguments](http://www.cplusplus.com/doc/tutorial/functions2/#default_values), and strong type checking in addition to all the features of the C language.  
  
The first C with Classes compiler was called Cfront, which was derived from a C compiler called CPre. It was a program designed to translate C with Classes code to ordinary C. A rather interesting point worth noting is that Cfront was written mostly in C with Classes, making it a self-hosting compiler (a compiler that can compile itself). Cfront would later be abandoned in 1993 after it became difficult to integrate new features into it, namely C++ [exceptions](http://www.cplusplus.com/doc/tutorial/exceptions/). Nonetheless, Cfront made a huge impact on the implementations of future compilers and on the Unix operating system.  
  
In 1983, the name of the language was changed from C with Classes to C++. The ++ operator in the C language is an operator for incrementing a variable, which gives some insight into how Stroustrup regarded the language. Many new features were added around this time, the most notable of which are [virtual functions](http://www.cplusplus.com/doc/tutorial/polymorphism/#virtual), [function overloading](http://www.cplusplus.com/doc/tutorial/functions2/#function_overload), references with the & symbol, the const keyword, and single-line comments using two forward slashes (which is a feature taken from the language BCPL).  
  
In 1985, Stroustrup's reference to the language entitled *The C++ Programming Language* was published. That same year, C++ was implemented as a commercial product. The language was not officially standardized yet, making the book a very important reference. The language was updated again in 1989 to include protected and static members, as well as inheritance from several classes.  
  
In 1990, *The Annotated C++ Reference Manual* was released. The same year, Borland's Turbo C++ compiler would be released as a commercial product. Turbo C++ added a plethora of additional libraries which would have a considerable impact on C++'s development. Although Turbo C++'s last stable release was in 2006, the compiler is still widely used.  
  
In 1998, the C++ standards committee published the first international standard for [C++ ISO/IEC 14882:1998](http://www.iso.org/iso/catalogue_detail.htm?csnumber=25845), which would be informally known as C++98. *The Annotated C++ Reference Manual* was said to be a large influence in the development of the standard. [The Standard Template Library](http://www.cplusplus.com/reference/stl/), which began its conceptual development in 1979, was also included. In 2003, the committee responded to multiple problems that were reported with their 1998 standard, and revised it accordingly. The changed language was dubbed [C++03](http://www.iso.org/iso/catalogue_detail.htm?csnumber=38110).  
  
In 2005, the C++ standards committee released a technical report (dubbed TR1) detailing various features they were planning to add to the latest C++ standard. The new standard was informally dubbed C++0x as it was expected to be released sometime before the end of the first decade. Ironically, however, the new standard would not be released until mid-2011. Several technical reports were released up until then, and some compilers began adding experimental support for the new features.  
  
In mid-2011, [the new C++ standard](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372) (dubbed C++11) was finished. The [Boost library project](http://www.boost.org/) made a considerable impact on the new standard, and some of the new modules were derived directly from the corresponding Boost libraries. Some of the new features included regular expression support (details on regular expressions may be found [here](http://www.regular-expressions.info/)), a comprehensive randomization library, a new C++ time library, atomics support, a standard threading library (which up until 2011 both C and C++ were lacking), a new [for loop](http://www.cplusplus.com/doc/tutorial/control/#for) syntax providing functionality similar to foreach loops in certain other languages, the auto keyword, new container classes, better support for unions and array-initialization lists, and variadic templates.

2014--c++14

ide's we are using for c++ programming are:turbo c++ for windows,div c++,code blocks

**First c++ program in turbo c++;**

#include<iostream.h>

#include<conio.n>

void main()

{

cout<<"hello world";

}

**First c++ program in dev c++;**

#include<iostream> //header file

usingnamespace std;

// main() is where program execution begins.

int main(){

cout <<"Hello World";// prints Hello World

}

Let us look various parts of the above program:

* The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.
* The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.It contains the code that defines cout, cin, endl, etc. If we want to use them, we need them to be in our code, that's why we include it.
* The next line **// main() is where program execution begins.** is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
* The line **int main()** is the main function where program execution begins.
* The next line **cout << "This is my first C++ program.";** causes the message "This is my first C++ program" to be displayed on the screen.

## Compile & Execute C++ Program

Let's look at how to save the file, compile and run the program. Please follow the steps given below:

* Open a text editor and add the code as above.
* Save the file as: hello.cpp
* Now, compile and run your program.
* You will be able to see ' Hello World ' printed on the window

## Semicolons & Blocks in C++

In C++, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

For example, following are three different statements −

x = y;

y = y+1;

add(x, y);

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example:

{

cout <<"Hello World";// prints Hello World

}

C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where on a line you put a statement. For example:

x = y;

y = y+1;

add(x, y);

is the same as

x = y; y = y+1; add(x, y);

## ****Tokens****

A token is a group of characters that logically belong together. The programmer can write a program by using tokens. C++ uses the following types of tokens.   
Keywords, Identifiers, Literals, Operators.

## C++ Keywords

The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names.

|  |  |  |  |
| --- | --- | --- | --- |
| Asm | Else | new | this |
| Auto | Enum | operator | throw |
| Bool | Explicit | private | true |
| Break | Export | protected | try |
| Case | Extern | public | typedef |
| Catch | False | register | typeid |
| Char | Float | reinterpret\_cast | typename |
| Class | For | return | union |
| Const | Friend | short | unsigned |
| const\_cast | Goto | signed | using |
| Continue | If | sizeof | virtual |
| Default | Inline | static | void |
| Delete | Int | static\_cast | volatile |
| Do | Long | struct | wchar\_t |
| Double | Mutable | switch | while |
| dynamic\_cast | Namespace | template |  |

## ****2. Identifiers****

Symbolic names can be used in C++ for various data items used by a programmer in his program. A symbolic name is generally  known as an identifier. The identifier is a sequence of characters taken from C++ character set. The rule for the formation of an identifier are:

* An identifier can consist of alphabets, digits and/or underscores.
* It must not start with a digit
* C++ is case sensitive that is upper case and lower case letters are considered different from each other.
* It should not be a reserved word.

## ****3. Literals****

Literals (often referred to as constants) are data items that never change their value during the execution of the program. The following types of literals are available in C++.

* Integer-Constants ex: 120,142
* Character-constants ex: 'a'…..
* Floating-constants ex: 100.12
* Strings-constants ex:'lak'

## 4. Punctuators

The following characters are used as punctuators in C++.

|  |  |
| --- | --- |
| Brackets [   ] | Opening and closing brackets indicate single and multidimensional array subscript. |
| Parentheses (   ) | Opening and closing brackets indicate functions calls,; function parameters for grouping expressions etc. |
| Braces {   } | Opening and closing braces indicate the start and end of a compound statement. |
| Comma , | It is used as a separator in a function argument list. |
| Semicolon ; | It is used as a statement terminator. |
| Colon : | It indicates a labeled statement or conditional operator symbol. |
| Asterisk \* | It is used in pointer declaration or as multiplication operator. |
| Equal sign = | It is used as an assignment operator. |
| Pound sign # | It is used as pre-processor directive. |

## 5. Operators

Operators are special symbols used for specific purposes. C++ provides six types of operators. Arithmetical operators, Relational operators,  Logical operators, Unary operators, Assignment operators, Conditional operators, Comma operator

# Comments in C++

C++ comments start with /\* and end with \*/. For example:

/\* This is a comment \*/

/\* C++ comments can also

\* span multiple lines

\*/

A comment can also start with //, extending to the end of the line. For example:

#include<iostream>

usingnamespace std;

main(){

cout <<"Hello World";// prints Hello World

}

When the above code is compiled, it will ignore **// prints Hello World** and final executable will produce the following result:

Hello World

# C++ Data Types

|  |  |  |
| --- | --- | --- |
| **Type** | **Typical Bit Width** | **Typical Range** |
| Char | 1byte | -128 to 127 or 0 to 255 |
| unsigned char | 1byte | 0 to 255 |
| signed char | 1byte | -128 to 127 |
| Int | 4bytes | -2147483648 to 2147483647 |
| unsigned int | 4bytes | 0 to 4294967295 |
| signed int | 4bytes | -2147483648 to 2147483647 |
| short int | 2bytes | -32768 to 32767 |
| unsigned short int | 2bytes | 0 to 65,535 |
| signed short int | 2bytes | -32768 to 32767 |
| long int | 8bytes | -2,147,483,648 to 2,147,483,647 |
| signed long int | 4bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 4bytes | 0 to 4,294,967,295 |
| Float | 4bytes | +/- 3.4e +/- 38 (~7 digits) |
| Double | 8bytes | +/- 1.7e +/- 308 (~15 digits) |
| long double | 8bytes | +/- 1.7e +/- 308 (~15 digits) |
| wchar\_t | 2 or 4 bytes | 1 wide character |

example:

#include <iostream>

using namespace std;

int main() {

cout << "Size of char : " << sizeof(char) << endl;

cout << "Size of int : " << sizeof(int) << endl;

cout << "Size of short int : " << sizeof(short int) << endl;

cout << "Size of long int : " << sizeof(long int) << endl;

cout << "Size of float : " << sizeof(float) << endl;

cout << "Size of double : " << sizeof(double) << endl;

cout << "Size of wchar\_t : " << sizeof(wchar\_t) << endl;

}

# C++ Variable Types

A variable provides us with names storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive:

There are following basic types of variable in C++ as explained in last chapter:

|  |  |
| --- | --- |
| **Type** | **Description** |
| Bool | Stores either value true or false. |
| Char | Typically a single octet(one byte). This is an integer type. |
| Int | The most natural size of integer for the machine. |
| Float | A single-precision floating point value. |
| Double | A double-precision floating point value. |

## Variable Definition in C++

A variable definition means to tell the compiler where and how much to create the storage for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows −

type variable\_list;

ex:

int i, j, k;

char c, ch;

float f, salary;

double d;

## Variable Declaration in C++

int i=200;

float f=10.24;

# Variable Scope in C++

A scope is a region of the program and broadly speaking there are three places, where variables can be declared −

* Inside a function or a block which is called local variables,
* In the definition of function parameters which is called formal parameters.
* Outside of all functions which is called global variables.

We will learn what a function is and it's parameter in subsequent chapters. Here let us explain what local and global variables are.

## Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables:

#include<iostream>

usingnamespace std;

int main (){

// Local variable declaration:

int a, b;

int c;

// actual initialization

a =10;

b =20;

c = a + b;

cout << c;

return0;

}

### Output

30

## Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

#include<iostream>

Ud.singnamespace std;

// Global variable declaration:

int g;

int main (){

// Local variable declaration:

int a, b;

// actual initialization

a =10;

b =20;

g = a + b;

cout << g;

}

### Output

30

example2:

#include<iostream>

#include<conio.h>

using namespace std;

int a=50;

int sum()

{

int b=900;

int c=a+b;

cout<<"the value of c is"<<c;

}

int sum1()

{

int f=500;

int g=a+f;

cout<<"the value of c is"<<g;

}

int main()

{

int d,e;

d=500;

e=d+a;

cout<<"the value of e is"<<e;

sum();

sum1();

}

Operators in C++

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides the following types of operators:

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Bitwise Operators
* Assignment Operators
* Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

## Arithmetic Operators

There are following arithmetic operators supported by C++ language:

Assume variable A holds 10 and variable B holds 20, then:

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_arithmatic_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A - B will give -10 |
| \* | Multiplies both operands | A \* B will give 200 |
| / | Divides numerator by de-numerator | B / A will give 2 |
| % | Modulus Operator and remainder of after an integer division | B % A will give 0 |
| ++ | [**Increment operator**](https://www.tutorialspoint.com/cplusplus/cpp_increment_decrement_operators.htm), increases integer value by one | A++ will give 11 |
| -- | [**Decrement operator**](https://www.tutorialspoint.com/cplusplus/cpp_increment_decrement_operators.htm), decreases integer value by one | A-- will give 9 |

## example:1

#include<iostream>

using namespace std;

main()

{

int a=20, b=40, c;

cout << "the value of a is =" << a << endl;

cout << "the value of b is ="<< b << endl;

c=a+b;

cout<<"there for c is ="<< c << endl;

}

example: ***2***

## *#include<iostream>*

## *#include<conio.h>*

## *main()*

## *{*

## *using namespace std;*

## *int a,b,c,d,e,f,g;*

## *cout<<"\n Enter First Number a : ";*

## *cin>>a;*

## *cout<<"\n Enter Second Number b : ";*

## *cin>>b;*

## *c=a+b;*

## *d=a-b;*

## *e=a\*b;*

## *f=a/b;*

## *g=a%b;*

## *cout<<" Addition = "<<c<<"\n";*

## *cout<<" Subtraction = "<<d<<"\n";*

## *cout<<" Multiplication = "<<e<<"\n";*

## *cout<<" Division = "<<f<<"\n";*

## *cout<<" Modulus = "<<g<<"\n";*

## *getch();*

## *}*

## Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then:

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_relational_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

## Logical Operators

There are following logical operators supported by C++ language

Assume variable A holds 1 and variable B holds 0, then:

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_logical_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non-zero, then condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false. | !(A && B) is true. |

## Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P** | **Q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Assume if A = 60; and B = 13; now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_bitwise_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 0000 1111 |

## Assignment Operators

There are following assignment operators supported by C++ language:

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_assignment_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator | C |= 2 is same as C = C | 2 |

## Misc Operators

There are few other operators supported by C++ Language.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| Sizeof | [**sizeof operator**](https://www.tutorialspoint.com/cplusplus/cpp_sizeof_operator.htm) returns the size of a variable. For example, sizeof(a), where a is integer, will return 4. |
| Condition ? X : Y | [**Conditional operator**](https://www.tutorialspoint.com/cplusplus/cpp_conditional_operator.htm). If Condition is true ? then it returns value X : otherwise value Y |
| , | [**Comma operator**](https://www.tutorialspoint.com/cplusplus/cpp_comma_operator.htm) causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list. |
| . (dot) and -> (arrow) | [**Member operators**](https://www.tutorialspoint.com/cplusplus/cpp_member_operators.htm) are used to reference individual members of classes, structures, and unions. |
| Cast | [**Casting operators**](https://www.tutorialspoint.com/cplusplus/cpp_casting_operators.htm) convert one data type to another. For example, int(2.2000) would return 2. |
| & | [**Pointer operator &**](https://www.tutorialspoint.com/cplusplus/cpp_pointer_operators.htm) returns the address of an variable. For example &a; will give actual address of the variable. |
| \* | [**Pointer operator \***](https://www.tutorialspoint.com/cplusplus/cpp_pointer_operators.htm) is pointer to a variable. For example \*var; will pointer to a variable var. |

## Operators Precedence in C++

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has higher precedence than +, so it first gets multiplied with 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_operators_precedence.htm)

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

example:

#include<conio.h>

#include<iostream>

main()

{

using namespace std;

int a=1,b;

b=a++;

cout<<b<<" "<<a;

getch();

}

#include<conio.h>

#include<iostream>

main()

{

using namespace std;

int a=1,b;

b=a++ + ++a + ++a;

cout<<b<<" "<<a;

getch();

}

C++ decision making statements

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



C++ programming language provides following types of decision making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Statement** | **Description** |
| [**if statement**](https://www.tutorialspoint.com/cplusplus/cpp_if_statement.htm) | An if statement consists of a boolean expression followed by one or more statements. |
| [**if...else statement**](https://www.tutorialspoint.com/cplusplus/cpp_if_else_statement.htm) | An if else statement can be followed by an optional else statement, which executes when the boolean expression is false. |
| [**switch statement**](https://www.tutorialspoint.com/cplusplus/cpp_switch_statement.htm) | A switch statement allows a variable to be tested for equality against a list of values. |
| [**nested if statements**](https://www.tutorialspoint.com/cplusplus/cpp_nested_if.htm) | You can use one if or else if statement inside another if or else if statement(s). |
| [**nested switch statements**](https://www.tutorialspoint.com/cplusplus/cpp_nested_switch.htm) | You can use one swicth statement inside another switch statement(s). |

## The ? : Operator

We have covered [conditional operator ? :](https://www.tutorialspoint.com/cplusplus/cpp_conditional_operator.htm) in previous chapter which can be used to replace **if...else** statements. It has the following general form:

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

***If Example:***

#include <iostream>

#include <conio.h>

using namespace std;

int main()

{

int a;

cout<<"please enter value(1/2)";

cin>>a;

if(a==1)

tf{

cout<<"\n WINNNNNNNINGGGG you entered the winning ball";

}

if(a==2)

{

cout<<"\n you lose better luck next time";

}

getch();

}

***If-Else Example:***

#include<iostream.h>

#include<conio.h>

int main()

{

int a;

cout<<"please enter value(1/2)";

cin>>a;

if(a==1)

{ cout<<"\n you entered 1";

}

if(a==2)

{

cout<<"\n you entered 2";

}

else

{

cout<<"\n you entered"<<a;

}

getch();

}

***Switch Example:***

#include<iostream>

#include<conio.h>

int main()

{

using namespace std;

int color;

cout<<"enter the color number in a rainbow(1-7)";

cin>>color;

switch(color)

{

case 1:

{

cout<<"\n first color in a rainbow is voilet";

break;

}

case 2:

{

cout<<"\n first color in a rainbow is voilet2";

break;

}

case 3:

{

cout<<"\n first color in a rainbow is voilet3";

break;

}

case 4:

{

cout<<"\n first color in a rainbow is voilet4";

break;

}

default:

{

cout<<"you enter some other color";

}

getch();

}

}

C++ Loop Types

There may be a situation, when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



C++ programming language provides the following types of loop to handle looping requirements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| [**while loop**](https://www.tutorialspoint.com/cplusplus/cpp_while_loop.htm) | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| [**for loop**](https://www.tutorialspoint.com/cplusplus/cpp_for_loop.htm) | Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| [**do...while loop**](https://www.tutorialspoint.com/cplusplus/cpp_do_while_loop.htm) | Like a while statement, except that it tests the condition at the end of the loop body |
| [**nested loops**](https://www.tutorialspoint.com/cplusplus/cpp_nested_loops.htm) | You can use one or more loop inside any another while, for or do..while loop. |

## Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Control Statement** | **Description** |
| [**break statement**](https://www.tutorialspoint.com/cplusplus/cpp_break_statement.htm) | Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| [**continue statement**](https://www.tutorialspoint.com/cplusplus/cpp_continue_statement.htm) | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| [**goto statement**](https://www.tutorialspoint.com/cplusplus/cpp_goto_statement.htm) | Transfers control to the labeled statement. Though it is not advised to use goto statement in your program. |

for loop example:

#include<iostream.h>

#include<conio.h>

main()

{

int I;

for(i=0;i<10;i++)

{

cout<<i;

cout<<"\n";

}

getch();

}

while loop example:

#include<iostream>

#include<conio.h>

int main(){

using namespace std;

int inum;

char cans;

cout<<"do you want to find out the square of a number (y/n)";

cin>>cans;

while(cans=='y')

{

cout<<"\n enter the number";

cin>>inum;

cout<<"square of "<<inum<<"is"<<inum\*inum;

cout<<"do you want to find out the square of another number (y/n)";

cin>>cans;

}

cout<<"out of the while loop";

getch();

}

do-while loop

#include<iostream>

#include<conio.h>

int main(){

using namespace std;

int inum;

char cans;

do{

cout<<"\n enter the number";

cin>>inum;

cout<<"square of "<<inum<<"is"<<inum\*inum;

cout<<"do you want to find out the square of another number (y/n)";

cin>>cans;

}

while(cans=='y');

cout<<"out of the while loop";

getch();

}

C++ Functions

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is knows as with various names like a method or a sub-routine or a procedure etc.

## Defining a Function

The general form of a C++ function definition is as follows:

return\_type function\_name( parameter list ){

body of the function

}

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

* **Return Type**: A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword **void**.
* **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* **Function Body:** The function body contains a collection of statements that define what the function does.

## Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:

// function returning the max between two numbers

int max(int num1,int num2){

// local variable declaration

int result;

if(num1 > num2)

result = num1;

else

result = num2;

return result;

}

## Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

return\_type function\_name( parameter list );

For the above defined function max(), following is the function declaration:

int max(int num1, int num2);

Parameter names are not importan in function declaration only their type is required, so following is also valid declaration:

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

## Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

#include<iostream>

usingnamespace std;

// function declaration

int max(int num1,int num2);

int main (){

// local variable declaration:

int a =100;

int b =200;

int ret;

// calling a function to get max value.

ret = max(a, b);

cout <<"Max value is : "<< ret << endl;

}

// function returning the max between two numbers

int max(int num1,int num2){

// local variable declaration

int result;

if(num1 > num2)

result = num1;

else

result = num2;

return result;

}

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result:

Max value is : 200

## Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

|  |  |
| --- | --- |
| **Call Type** | **Description** |
| [**Call by value**](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_value.htm) | This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| [**Call by reference**](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_reference.htm) | This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

## Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example:

#include<iostream>

usingnamespace std;

int sum(int a,int b=20){

int result;

result = a + b;

return(result);

}

int main (){

// local variable declaration:

int a =100;

int b =200;

int result;

// calling a function to add the values.

result = sum(a, b);

cout <<"Total value is :"<< result << endl;

// calling a function again as follows.

result = sum(a);

cout <<"Total value is :"<< result << endl;

return0;

}

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

Total value is :300

Total value is :120

example:

#include<iostream>

#include<conio.h>

using namespace std;

int sum(int a=10,int b=20)

{

int c;

c=a+b;

cout<<"the sum of 2 no is "<<c;

cout<<"\n";

} int main()

{

sum();

sum(1,2);

sum(1);

getch();

}

call by value and call by reference:

call by value:

copy of the original variables.In call by value, **original value can not be changed** or modified. In call by value, when you passed value to the function it is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only but it not change the value of variable inside the caller function such as main().

#include <iostream>

#include<conio.h>

using namespace std;

void swap(int a,int b);

main()

{

int a=10,b=20;

cout<<"before function"<<"a=="<<a<<"b="<<b;

swap(a,b);

cout<<"after swapping"<<"a="<<a<<"b="<<b;

getch();

}

void swap(int a,int b)

{

int temp;

temp=a;

a=b;

b=temp;

}

call by reference

-----------------------

In call by reference, **original value is changed** or modified because we pass reference (address). Here, address of the value is passed in the function, so actual and formal arguments shares the same address space. Hence, any value changed inside the function, is reflected inside as well as outside the function.

#include <iostream>

#include<conio.h>

using namespace std;

void swap(int &a,int &b);

main()

{

int a=10,b=20;

cout<<"before function"<<"a=="<<a<<"b="<<b;

swap(a,b);

cout<<"after swapping"<<"a="<<a<<"b="<<b;

getch();

}

void swap(int &a,int &b)

{

int temp;

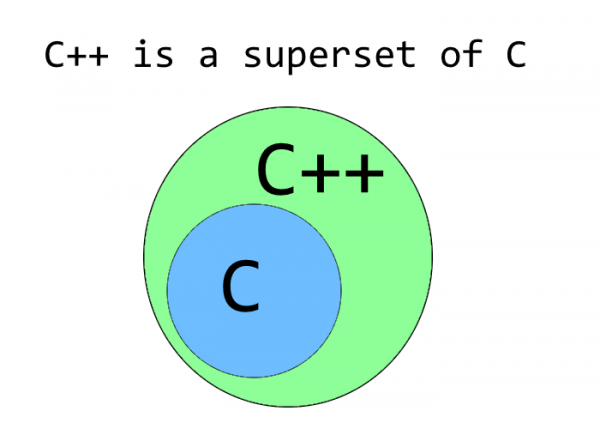
temp=a;

a=b;

b=temp;

}

differences Between C And C++



1) . C follows the procedural programming paradigm while C++ is a [multi-paradigm](http://durofy.com/programming/c-as-a-multi-paradigm-programming-language/)language(procedural as well as object oriented)

2. In case of C, the data is not secured while the data is secured(hidden) in C++

This difference is due to specific [OOP features](http://durofy.com/programming/the-basics-of-object-oriented-programming/) like Data Hiding which are not present in C.

3. C is a low-level language while C++ is a middle-level language (Relatively, Please see the discussion at the end of the post)

C is regarded as a low-level language(difficult interpretation & less user friendly) while C++ has features of both low-level(concentration on whats going on in the machine hardware) & high-level languages(concentration on the program itself) & hence is regarded as a middle-level language.

Note: This is a relative difference. See updates at end of this post.

4. C uses the top-down approach while C++ uses the bottom-up approach

In case of C, the program is formulated step by step, each step is processed into detail while in C++, the base elements are first formulated which then are linked together to give rise to larger systems.

5. C is function-driven while C++ is object-driven

Functions are the building blocks of a C program while objects are building blocks of a C++ program.

6. C++ supports function overloading while C does not

Overloading means two functions having the same name in the same program. This can be done only in C++ with the help of [Polymorphism](http://durofy.com/programming/the-basics-of-object-oriented-programming/)(an OOP feature)

7. We can use functions inside structures in C++ but not in C.

In case of C++, functions can be used inside a structure while structures cannot contain functions in C.

8. The NAMESPACE feature in C++ is absent in case of C

C++ uses NAMESPACE which avoid name collisions. For instance, two students enrolled in the same university cannot have the same roll number while two students in different universities might have the same roll number. The universities are two different namespace & hence contain the same roll number(identifier) but the same university(one namespace) cannot have two students with the same roll number(identifier)

9. The standard input & output functions differ in the two languages

C uses scanf & printf while C++ uses cin>>& cout<< as their respective input & output functions

10. C++ allows the use of reference variables while C does not

Reference variables allow two variable names to point to the same memory location. We cannot use these variables in C programming.

Object oriented programming

## C++ 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 follow 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.

syntax:

#include<iostream>

#include<conio.h>

class student

{

//data members of student

};

int main()

{

student lak,rao;

}

example 1:

#include<iostream>

usingnamespace std;

classBox

{

public:

double length;// Length of a box

double breadth;// Breadth of a box

double height;// Height of a box

};

int main()

{

BoxBox1;// Declare Box1 of type Box

BoxBox2;// Declare Box2 of type Box

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

// box 1 specification

Box1.height =4.0;

Box1.length =6.0;

Box1.breadth =3.0;

// box 2 specification

Box2.height =10.0;

Box2.length =12.0;

Box2.breadth =12.0;

// volume of box 1

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

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

// volume of box 2

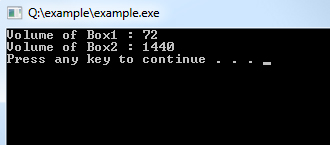
volume =Box2.height \*Box2.length \*Box2.breadth;

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

return0;

}

Program Output:



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.

### Program to enter students details and display it

Example:

#include<iostream>

usingnamespace std;

class stud

{

public:

char name[30],clas[10];

int rol,age;

void enter()

{

cout<<"Enter Student Name: "; cin>>name;

cout<<"Enter Student Age: "; cin>>age;

cout<<"Enter Student Roll number: "; cin>>rol;

cout<<"Enter Student Class: "; cin>>clas;

}

void display()

{

cout<<"\n Age\tName\tR.No.\tClass";

cout<<"\n"<<age<<"\t"<<name<<"\t"<<rol<<"\t"<<clas;

}

};

int main()

{

class stud s;

s.enter();

s.display();

cin.get(); //use this to wait for a keypress

}

Constructors in C++

**What is constructor?**  
A constructor is a member function of a class which initializes objects of a class. In C++,Constructor is automatically called when object(instance of class) create.It is special member function of the class.

**How constructors are different from a normal member function?**

A constructor is different from normal functions in following ways:

* Constructor has same name as the class itself
* Constructors don’t have return type
* A constructor is automatically called when an object is created.
* If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

**Types of Constructors**

1. [**Default Constructors:**](https://www.geeksforgeeks.org/c-internals-default-constructors-set-1/) Default constructor is the constructor which doesn’t take any argument. It has no parameters.

|  |
| --- |
| // Cpp program to illustrate the  // concept of Constructors  #include <iostream>  using namespace std;    class construct  {  public:      int a, b;            // Default Constructor      construct()      {          a = 10;          b = 20;      }  };    int main()  {          // Default constructor called automatically          // when the object is created      construct c;      cout << "a: "<< c.a << endl << "b: "<< c.b;      return 1;  } |

1. Run on IDE
2. Output:
3. a: 10
4. b: 20
5. **Note:**Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly. The default value of variables is 0 in case of automatic initialization.
   1. **Parameterized Constructors:**It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.

|  |
| --- |
| // CPP program to illustrate  // parameterized constructors  #include<iostream>  using namespace std;    class Point  {      private:          int x, y;      public:          // Parameterized Constructor          Point(int x1, int y1)          {              x = x1;              y = y1;          }            int getX()          {              return x;          }          int getY()          {              return y;          }      };    int main()  {      // Constructor called      Point p1(10, 15);        // Access values assigned by constructor      cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();        return 0;  } |

Output:

p1.x = 10, p1.y = 15

# Destructors

BY ALEX ON SEPTEMBER 6TH, 2007 | LAST MODIFIED BY ALEX ON NOVEMBER 29TH, 2017

A **destructor** is another special kind of class member function that is executed when an object of that class is destroyed. Whereas constructors are designed to initialize a class, destructors are designed to help clean up.

When an object goes out of scope normally, or a dynamically allocated object is explicitly deleted using the delete keyword, the class destructor is called (if it exists) to do any necessary clean up before the object is removed from memory. For simple classes (those that just initialize the values of normal member variables), a destructor is not needed because C++ will automatically clean up the memory for you.

However, if your class object is holding any resources (e.g. dynamic memory, or a file or database handle), or if you need to do any kind of maintenance before the object is destroyed, the destructor is the perfect place to do so, as it is typically the last thing to happen before the object is destroyed.

**Destructor naming**

Like constructors, destructors have specific naming rules:  
1) The destructor must have the same name as the class, preceded by a tilde (~).  
2) The destructor can not take arguments.  
3) The destructor has no return type.

Note that rule 2 implies that only one destructor may exist per class, as there is no way to overload destructors since they can not be differentiated from each other based on arguments.

Just like constructors, destructors should not be called explicitly. However, destructors may safely call other member functions since the object isn’t destroyed until after the destructor executes.

**A destructor example**

include <iostream>

using namespace std;

class construct

{

public:

int a, b;

// Default Constructor

construct() {

a=200,b=400;

cout<<"hello";

}

~ construct()

{

int c=a+b;

cout<<"hello how are you"<<c;

}

};

int main()

{ // Default constructor called automatically

// when the object is created

construct c;

}

#### **3)Abstraction**

Abstraction refers to showing only the essential features of the application and hiding the details. In C++, classes provide methods to the outside world to access & use the data variables, but the variables are hidden from direct access. This can be done access specifiers.

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 the user doesn't need to know about, but is needed for the class to operate properly.

#### 4) Encapsulation

It can also be said data binding. Encapsulation is all about binding the data variables and functions together in class

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;

}

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

Total 60

#### 5) Polymorphism

It is a feature, which lets us create functions with same name but different arguments, which will perform differently. That is function with same name, functioning in different way. Or, it also allows us to redefine a function to provide its new definition. You will learn how to do this in details soon in coming lessons.

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

#include <bits/stdc++.h>

using namespace std;

#### class Geeks

#### {

#### public:

#### 

#### // function with 1 int parameter

#### void func(int x)

#### {

#### cout << "value of x is " << x << endl;

#### }

#### 

#### // function with same name but 1 double parameter

#### void func(double x)

#### {

#### cout << "value of x is " << x << endl;

#### }

#### 

#### // function with same name and 2 int parameters

#### void func(int x, int y)

#### {

#### cout << "value of x and y is " << x << ", " << y << endl;

#### }

#### };

#### 

#### int main() {

#### 

#### Geeks obj1;

#### 

#### // Which function is called will depend on the parameters passed

#### // The first 'func' is called

#### obj1.func(7);

#### 

#### // The second 'func' is called

#### obj1.func(9.132);

#### 

#### // The third 'func' is called

#### obj1.func(85,64);

#### } Inheritance

Inheritance is a way to reuse once written code again and again. The class which is inherited is called base calls & the class which inherits is called derived class. So when, a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship

Base & Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

class derived-class: access-specifier base-class

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

// Derived class

class Rectangle: public Shape {

public:

int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

Rect.setWidth(5);

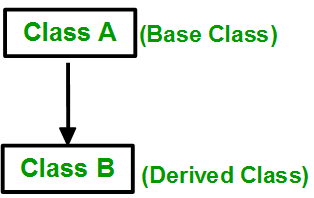
Rect.setHeight(7);

// Print the area of the object.

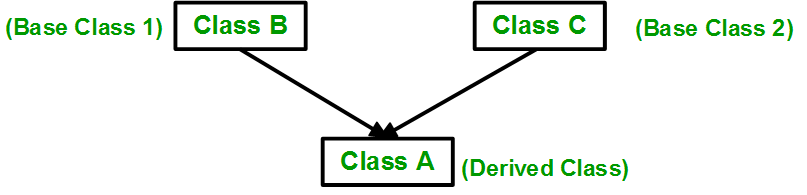
cout << "Total area: " << Rect.getArea() << endl;

}

**Types of Inheritance in C++**

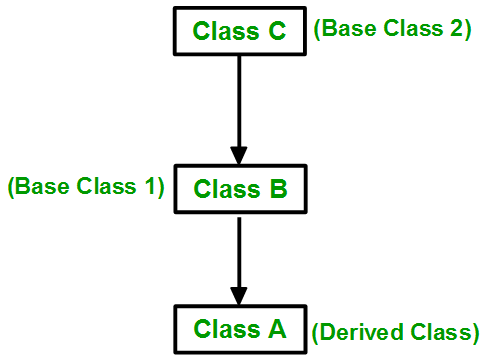
1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.  
   **Syntax**:
2. class subclass\_name : access\_mode base\_class
3. {
4. //body of subclass
5. };

|  |
| --- |
| // C++ program to explain  // Single inheritance  #include <iostream>  using namespace std;  // base class  class Vehicle {  public:  int a=200;  int b=300;  Vehicle()  {  cout << "This is a Vehicle" << endl;  }  };  // sub class derived from two base classes  class Car:public Vehicle{  public:  Car()  {  int c=a+b;  cout<<"the value of c is"<<c;  }  };  // main function  int main(){  // creating object of sub class will  // invoke the constructor of base classes  Car obj;  } |

1. Run on IDE
2. Output:
3. This is a vehicle
4. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.  
   **Syntax**:
5. class subclass\_name : access\_mode base\_class1, access\_mode base\_class2, ....
6. {
7. //body of subclass
8. };

Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

|  |
| --- |
| #include <iostream>  using namespace std;  class Vehicle {  public:  int a=10;  int b=20;  Vehicle()  {  cout << "This is a Vehicle" << endl;  }  };    // second base class  class FourWheeler {  public:  int c=30;  FourWheeler()  {  cout << "This is a 4 wheeler Vehicle" << endl;  }  };    // sub class derived from two base classes  class Car: public Vehicle, public FourWheeler {  public:  int d;  Car()  {  d=a+b+c;  cout<<"the value of d is"<<d;  }  };    // main function  int main()  {  // creating object of sub class will  // invoke the constructor of base classes  Car obj;  } |

**3)Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.

#include <iostream>

using namespace std;

// base class

class Vehicle

{

public:

int a=100;

Vehicle()

{

cout << "This is a Vehicle" << endl;

}

};

class fourWheeler: public Vehicle

{ public:

int b=200;

fourWheeler()

{

int c=a+b;

cout<<"Objects with 4 wheels are vehicles"<<c;

}

};

// sub class derived from two base classes

class Car: public fourWheeler{

public:

int d;

Car()

{

d=b+a;

cout<<"Car has 4 Wheels"<<d;

}

};

// main function

int main()

{

//creating object of sub class will

//invoke the constructor of base classes

Car obj;

}

**4.Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.

|  |
| --- |
| // C++ program to implement  // Hierarchical Inheritance |

#include <iostream>

using namespace std;

// base class

class Vehicle

{

public:

int a=100;

int b=200;

Vehicle()

{

cout << "This is a Vehicle" << endl;

}

};

// first sub class

class Car: public Vehicle

{

public:

Car()

{

int c=a+b;

cout<<"the value of c is"<<c;

}

};

// second sub class

class Bus: public Vehicle

{

public:

Bus()

{

int d=b-a;

cout<<"the value of d is"<<d;

}

};

// main function

int main()

{

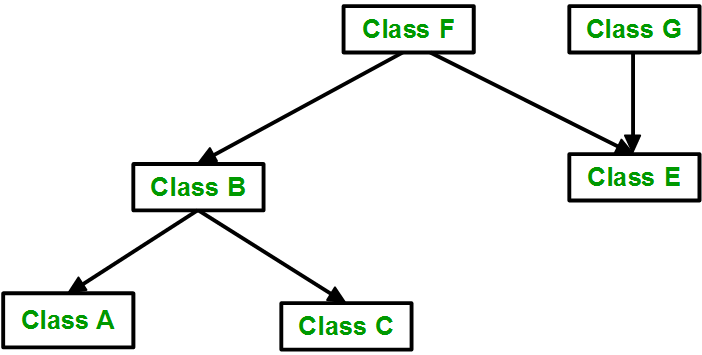
// creating object of sub class will

// invoke the constructor of base class

Car obj1;

Bus obj2;

}

**Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.Belowimage shows the combination of hierarchical and multiple inheritance:

|  |
| --- |
| // C++ program for Hybrid Inheritance    #include <iostream>  using namespace std;    // base class  class Vehicle  {  public:  int a=100;  Vehicle()  {  cout << "This is a Vehicle" << endl;  }  };    //base class  class Fare  {  public:  int b=200;  Fare()  {  cout<<"Fare of Vehicle\n";  }  };    // first sub class  class Car: public Vehicle  {    };    // second sub class  class Bus: public Vehicle, public Fare  {  public:  int d;  Bus()  {  d=a+b;  cout<<"the total cost"<<d;  }  };    // main function  int main()  {  // creating object of sub class will  // invoke the constructor of base class  Bus obj2;  } |

1. Run on IDE
2. Output:
3. This is a Vehicle
4. Fare of Vehicle

# Access Modifiers in C++

Access modifiers are used to implement important feature of Object Oriented Programming known as **Data Hiding**. Consider a real life example: What happens when a driver applies brakes? The car stops. The driver only knows that to stop the car, he needs to apply the brakes. He is unaware of how actually the car stops. That is how the engine stops working or the internal implementation on the engine side. This is what data hiding is.  
Access modifiers or Access Specifiers in a [class](https://www.geeksforgeeks.org/c-classes-and-objects/) are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

**Note**: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

Let us now look at each one these access modifiers in details:

* **Public**: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.
* // C++ program to demonstrate public
* // access modifier

#include<iostream>

using namespace std;

// class definition

class Circle

{

public:

double radius;

double compute\_area()

{

return 3.14\*radius\*radius;

}

};

// main function

int main()

{

Circle obj;

// accessing public datamember outside class

obj.radius = 5.5;

cout << "Radius is:" << obj.radius << "\n";

cout << "Area is:" << obj.compute\_area();

return 0;

}

Output:

Radius is:5.5

Area is:94.985

In the above program the data member radius is public so we are allowed to access it outside the class.

* **Private**: The class members declared as **private** can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the [friend functions](https://www.geeksforgeeks.org/friend-class-function-cpp/) are allowed to access the private data members of a class.  
  Example:
* // C++ program to demonstrate private
* // access modifier
* #include<iostream>
* using namespace std;
* class Circle
* {
* // private data member
* private:
* double radius;
* // public member function
* public:
* double compute\_area()
* { // member function can access private
* // data member radius
* return 3.14\*radius\*radius;
* }
* };
* // main function
* int main()
* {
* // creating object of the class
* Circle obj;
* // trying to access private data member
* // directly outside the class
* obj.radius = 1.5;
* cout << "Area is:" << obj.compute\_area();
* return 0;
* }

The output of above program will be a compile time error because we are not allowed to access the private data members of a class directly outside the class.  
**Output**:

In function 'int main()':

11:16: error: 'double Circle::radius' is private

double radius;

^

31:9: error: within this context

obj.radius = 1.5;

^

However we can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

// C++ program to demonstrate private

// access modifier

#include<iostream>

using namespace std;

class Circle

{

// private data member

private:

double radius;

// public member function

public:

double compute\_area(double r)

{ // member function can access private

// data member radius

radius = r;

double area = 3.14\*radius\*radius;

cout << "Radius is:" << radius << endl;

cout << "Area is: " << area;

}

};

// main function

int main()

{

// creating object of the class

Circle obj;

// trying to access private data member

// directly outside the class

obj.compute\_area(1.5);

return 0;

}

**Output**:

Radius is:1.5

Area is: 7.065

* **Protected**: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.
* // C++ program to demonstrate
* // protected access modifier

#include <bits/stdc++.h>

using namespace std;

// base class

class Parent

{

// protected data members

protected:

int a=100;

};

// sub class or derived class

class Child:public Parent

{

protected:

int b=200;

public:

void sum()

{int c=a+b;

cout << "c value" << c << endl;

}

};

// main function

int main() {

Child obj1;

obj1.sum();

}

Websites:

https://www.geeksforgeeks.org

https://www.tutorialspoint.com/cplusplus/

www.cplusplus.com

https://www.programiz.com

https://www.studytonight.com/cpp

www.learncpp.com/cpp-tutorial/

<https://www.tutorialcup.com>

sample questions:

1)explain oop’s concept with example programs?

Object &&class with program –5 marks

Encapsulation---5 marks

Polymorphism-5 marks

Abstraction---5 marks

Inheritance-20 marks

2)explain what is constructor & destructor and types with example programs?

3)what are the access modifiers with example programs?

4)what is the difference between c and c++?

5)explain local variables and global variables with example programs?

6)explain relational operators , logical operators with example programs?

7)explain decision making statements with example programs?

8)explain loop control statements with example programs?

9) explain loop statements with example programs?

10)what is function? define, declare a function?

Write a program functions with arguments.

11)explain what is call by value&call by reference with example programs?