*Inheritance – Types -Single and Multiple Inheritance - Multilevel Inheritance - Hierarchical Inheritance - Hybrid Inheritance - Advanced Functions - Inline, Friend- Virtual - Pure Virtual function - Abstract class - UML State Chart Diagram - UML Activity Diagram*

**C++ Inheritance**

Inheritance is one of the key features of Object-oriented programming in C++. It allows us to create a new [class](https://www.programiz.com/cpp-programming/object-class) (derived class) from an existing class (base class).

**The derived class inherits the features from the base class** and can have additional features of its own. For example,

class Animal {

// eat() function

// sleep() function

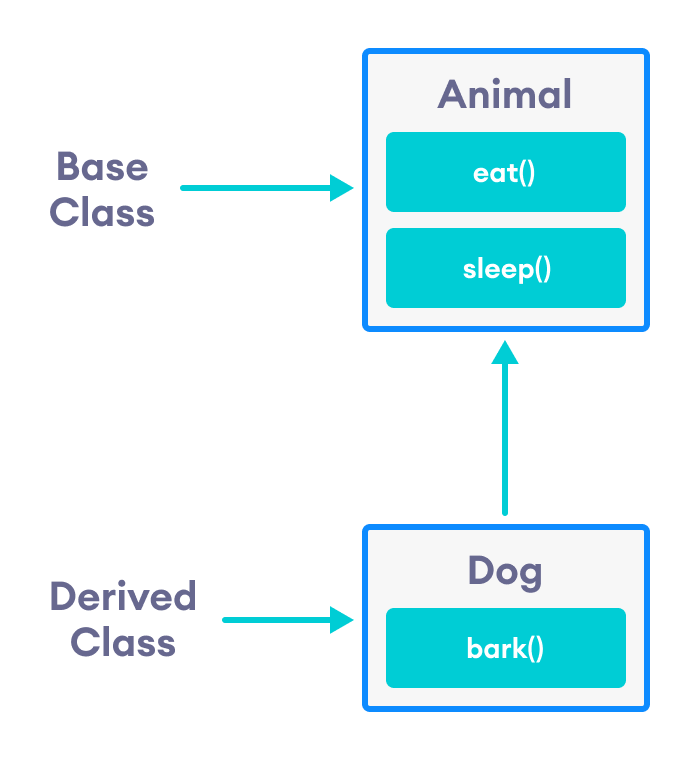
};

class Dog : public Animal {

// bark() function

};

Here, the Dog class is derived from the Animal class. Since Dog is derived from Animal, members of Animal are accessible to Dog.



Notice the use of the keyword public while inheriting Dog from Animal.

class Dog : public Animal {...};

We can also use the keywords private and protected instead of public

## is-a relationship

Inheritance is an **is-a relationship**. We use inheritance only if an **is-a relationship** is present between the two classes.

Here are some examples:

* A car is a vehicle.
* Orange is a fruit.
* A surgeon is a doctor.
* A dog is an animal.

// C++ program to demonstrate inheritance

#include <iostream>

using namespace std;

// base class

class Animal {

public:

void eat() {

cout << "I can eat!" << endl;

}

void sleep() {

cout << "I can sleep!" << endl;

}

};

// derived class

class Dog : public Animal {

public:

void bark() {

cout << "I can bark! Woof woof!!" << endl;

}

};

int main() {

// Create object of the Dog class

Dog dog1;

// Calling members of the base class

dog1.eat();

dog1.sleep();

// Calling member of the derived class

dog1.bark();

return 0;

}

**Output**

I can eat!

I can sleep!

I can bark! Woof woof!!

Here, dog1 (the object of derived class Dog) can access members of the base class Animal. It's because Dog is inherited from Animal.

## C++ protected Members

The access modifier protected is especially relevant when it comes to C++ inheritance.

Like private members, protected members are inaccessible outside of the class. However, they can be accessed by **derived classes** and **friend classes/functions**.

We need protected members if we want to hide the data of a class, but still want that data to be inherited by its derived classes.

### Example

// C++ program to demonstrate protected members

#include <iostream>

#include <string>

using namespace std;

// base class

class Animal {

private:

string color;

protected:

string type;

public:

void eat() {

cout << "I can eat!" << endl;

}

void sleep() {

cout << "I can sleep!" << endl;

}

void setColor(string clr) {

color = clr;

}

string getColor() {

return color;

}

};

// derived class

class Dog : public Animal {

public:

void setType(string tp) {

type = tp;

}

void displayInfo(string c) {

cout << "I am a " << type << endl;

cout << "My color is " << c << endl;

}

void bark() {

cout << "I can bark! Woof woof!!" << endl;

}

};

int main() {

// Create object of the Dog class

Dog dog1;

// Calling members of the base class

dog1.eat();

dog1.sleep();

dog1.setColor("black");

// Calling member of the derived class

dog1.bark();

dog1.setType("mammal");

// Using getColor() of dog1 as argument

// getColor() returns string data

dog1.displayInfo(dog1.getColor());

return 0;

}

**Output**

I can eat!

I can sleep!

I can bark! Woof woof!!

I am a mammal

My color is blacK

Here, the variable type is protected and is thus accessible from the derived class Dog. We can see this as we have initialized type in the Dog class using the function setType().

On the other hand, the private variable color cannot be initialized in Dog.

class Dog : public Animal {

public:

void setColor(string clr) {

// Error: member "Animal::color" is inaccessible

color = clr;

}

};

Also, since the protected keyword hides data, we cannot access type directly from an object of Dog or Animal class.

// Error: member "Animal::type" is inaccessible

dog1.type = "mammal";

## Access Modes in C++ Inheritance

In our previous tutorials, we have learned about C++ access specifiers such as [public, private, and protected](https://www.programiz.com/cpp-programming/public-protected-private-inheritance).

So far, we have used the public keyword in order to inherit a class from a previously-existing base class. However, we can also use the private and protected keywords to inherit classes. For example,

class Animal {

// code

};

class Dog : private Animal {

// code

};

class Cat : protected Animal {

// code

};

The various ways we can derive classes are known as **access modes**. These access modes have the following effect:

1. **public:** If a derived class is declared in public mode, then the members of the base class are inherited by the derived class just as they are.
2. **private:** In this case, all the members of the base class become private members in the derived class.
3. **protected:** The public members of the base class become protected members in the derived class.

The private members of the base class are always private in the derived class.

To learn more, visit our [C++ public, private, protected inheritance](https://www.programiz.com/cpp-programming/public-protected-private-inheritance) tutorial.

## Advantage of C++ Inheritance

**Code reusability:** Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

## Types Of Inheritance

**C++ supports five types of inheritance:**

* Single inheritance
* Multiple inheritance
* Hierarchical inheritance
* Multilevel inheritance
* Hybrid inheritance

## Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

**class** derived\_class\_name :: visibility-mode base\_class\_name

{

     // body of the derived class.

}

**Where,**

**derived\_class\_name:** It is the name of the derived class.

**visibility mode:** The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

**base\_class\_name:** It is the name of the base class.

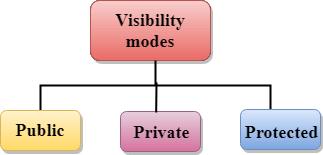
* When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
* When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

## How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

**Visibility modes can be classified into three categories:**



* **Public**: When the member is declared as public, it is accessible to all the functions of the program.
* **Private**: When the member is declared as private, it is accessible within the class only.
* **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

## Visibility of Inherited Members

|  |  |  |  |
| --- | --- | --- | --- |
| **Base class visibility** | **Derived class visibility** | | |
| **Public** | **Private** | **Protected** |
| Private | Not Inherited | Not Inherited | Not Inherited |
| Protected | Protected | Private | Protected |
| Public | Public | Private | Protected |

## C++ Single Inheritance

**Single inheritance** is defined as the inheritance in which a derived class is inherited from the only one base class.

C++ Inheritance

Where 'A' is the base class, and 'B' is the derived class.

## ++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Account {
4. **public**:
5. **float** salary = 60000;
6. };
7. **class** Programmer: **public** Account {
8. **public**:
9. **float** bonus = 5000;
10. };
11. **int** main(**void**) {
12. Programmer p1;
13. cout<<"Salary: "<<p1.salary<<endl;
14. cout<<"Bonus: "<<p1.bonus<<endl;
15. **return** 0;
16. }

Output:

Salary: 60000

Bonus: 5000

## C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Animal {
4. **public**:
5. **void** eat() {
6. cout<<"Eating..."<<endl;
7. }
8. };
9. **class** Dog: **public** Animal
10. {
11. **public**:
12. **void** bark(){
13. cout<<"Barking...";
14. }
15. };
16. **int** main(**void**) {
17. Dog d1;
18. d1.eat();
19. d1.bark();
20. **return** 0;
21. }

Output:

Eating...

Barking...

1. #include <iostream>
2. **using** **namespace** std;
3. **class** A
4. {
5. **int** a = 4;
6. **int** b = 5;
7. **public**:
8. **int** mul()
9. {
10. **int** c = a\*b;
11. **return** c;
12. }
13. };
15. **class** B : **private** A
16. {
17. **public**:
18. **void** display()
19. {
20. **int** result = mul();
21. cout <<"Multiplication of a and b is : "<<result<<  endl;
22. }
23. };
24. **int** main()
25. {
26. B b;
27. b.display();
29. **return** 0;
30. }

Output:

Multiplication of a and b is : 20

In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

## C++ Multilevel Inheritance

**Multilevel inheritance** is a process of deriving a class from another derived class.

C++ Inheritance

## C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Animal {
4. **public**:
5. **void** eat() {
6. cout<<"Eating..."<<endl;
7. }
8. };
9. **class** Dog: **public** Animal
10. {
11. **public**:
12. **void** bark(){
13. cout<<"Barking..."<<endl;
14. }
15. };
16. **class** BabyDog: **public** Dog
17. {
18. **public**:
19. **void** weep() {
20. cout<<"Weeping...";
21. }
22. };
23. **int** main(**void**) {
24. BabyDog d1;
25. d1.eat();
26. d1.bark();
27. d1.weep();
28. **return** 0;
29. }

Output:

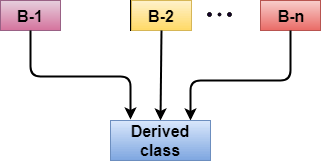
Eating...

Barking...

Weeping...

## C++ Multiple Inheritance

**Multiple inheritance** is the process of deriving a new class that inherits the attributes from two or more classes.



**Syntax of the Derived class:**

**class** D : visibility B-1, visibility B-2, ?

{

    // Body of the class;

}

Let's see a simple example of multiple inheritance.

#include <iostream>

**using** **namespace** std;

**class** A

{

**protected**:

**int** a;

**public**:

**void** get\_a(**int** n)

    {

        a = n;

    }

};

**class** B

{

**protected**:

**int** b;

**public**:

**void** get\_b(**int** n)

    {

        b = n;

    }

};

**class** C : **public** A,**public** B

{

**public**:

**void** display()

    {

cout << "The value of a is : " <<a<<  endl;

        cout << "The value of b is : " <<b<<  endl;

        cout<<"Addition of a and b is : "<<a+b;

    }

};

**int** main()

{

   C c;

   c.get\_a(10);

   c.get\_b(20);

   c.display();

**return** 0;

}

Output:

The value of a is : 10

The value of b is : 20

Addition of a and b is : 30

## Ambiquity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Let's understand this through an example:

1. #include <iostream>
2. **using** **namespace** std;
3. **class** A
4. {
5. **public**:
6. **void** display()
7. {
8. cout << "Class A" <<  endl;
9. }
10. };
11. **class** B
12. {
13. **public**:
14. **void** display()
15. {
16. cout << "Class B" <<  endl;
17. }
18. };
19. **class** C : **public** A, **public** B
20. {
21. **void** view()
22. {
23. display();
24. }
25. };
26. **int** main()
27. {
28. C c;
29. c.display();
30. **return** 0;
31. }

Output:

error: reference to 'display' is ambiguous

display();

* The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

1. **class** C : **public** A, **public** B
2. {
3. **void** view()
4. {
5. A :: display();         // Calling the display() function of class A.
6. B :: display();         // Calling the display() function of class B.
8. }
9. };

An ambiguity can also occur in single inheritance.

Consider the following situation:

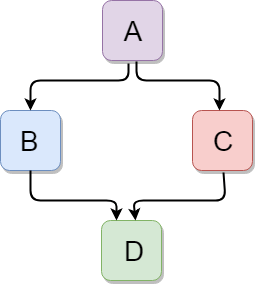
1. **class** A
2. {
3. **public**:
4. **void** display()
5. {
6. cout<<?Class A?;
7. }
8. } ;
9. **class** B  : public A
10. {
11. **public**:
12. **void** display()
13. {
14. cout<<?Class B?;
15. }
16. } ;

In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

1. **int** main()
2. {
3. B b;
4. b.display();               // Calling the display() function of B class.
5. b.B :: display();       // Calling the display() function defined in B class.
6. }

## C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



Let's see a simple example:

1. #include <iostream>
2. **using** **namespace** std;
3. **class** A
4. {
5. **protected**:
6. **int** a;
7. **public**:
8. **void** get\_a()
9. {
10. cout << "Enter the value of 'a' : " <<  endl;
11. cin>>a;
12. }
13. };
15. **class** B : **public** A
16. {
17. **protected**:
18. **int** b;
19. **public**:
20. **void** get\_b()
21. {
22. cout << "Enter the value of 'b' : " <<  endl;
23. cin>>b;
24. }
25. };
26. **class** C
27. {
28. **protected**:
29. **int** c;
30. **public**:
31. **void** get\_c()
32. {
33. cout << "Enter the value of c is : " <<  endl;
34. cin>>c;
35. }
36. };
38. **class** D : **public** B, **public** C
39. {
40. **protected**:
41. **int** d;
42. **public**:
43. **void** mul()
44. {
45. get\_a();
46. get\_b();
47. get\_c();
48. cout << "Multiplication of a,b,c is : " <<a\*b\*c<<  endl;
49. }
50. };
51. **int** main()
52. {
53. D d;
54. d.mul();
55. **return** 0;
56. }

Output:

Enter the value of 'a' :

10

Enter the value of 'b' :

20

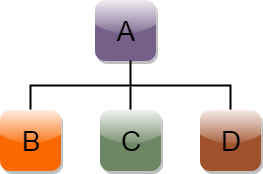
Enter the value of c is :

30

Multiplication of a,b,c is : 6000

## C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



**Syntax of Hierarchical inheritance:**

1. **class** A
2. {
3. // body of the class A.
4. }
5. **class** B : **public** A
6. {
7. // body of class B.
8. }
9. **class** C : **public** A
10. {
11. // body of class C.
12. }
13. **class** D : **public** A
14. {
15. // body of class D.
16. }

Let's see a simple example:

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Shape                 // Declaration of base class.
4. {
5. **public**:
6. **int** a;
7. **int** b;
8. **void** get\_data(**int** n,**int** m)
9. {
10. a= n;
11. b = m;
12. }
13. };
14. **class** Rectangle : **public** Shape  // inheriting Shape class
15. {
16. **public**:
17. **int** rect\_area()
18. {
19. **int** result = a\*b;
20. **return** result;
21. }
22. };
23. **class** Triangle : **public** Shape    // inheriting Shape class
24. {
25. **public**:
26. **int** triangle\_area()
27. {
28. **float** result = 0.5\*a\*b;
29. **return** result;
30. }
31. };
32. **int** main()
33. {
34. Rectangle r;
35. Triangle t;
36. **int** length,breadth,base,height;
37. cout << "Enter the length and breadth of a rectangle: " <<  endl;
38. cin>>length>>breadth;
39. r.get\_data(length,breadth);
40. **int** m = r.rect\_area();
41. cout << "Area of the rectangle is : " <<m<<  endl;
42. cout << "Enter the base and height of the triangle: " <<  endl;
43. cin>>base>>height;
44. t.get\_data(base,height);
45. **float** n = t.triangle\_area();
46. cout <<"Area of the triangle is : "  << n<< endl;
47. **return** 0;
48. }

Output:

Enter the length and breadth of a rectangle:

23

20

Area of the rectangle is : 460

Enter the base and height of the triangle:

2

5

Area of the triangle is : 5

**Inline Functions in C++**

C++ provides inline functions to reduce the function call overhead. An inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

**Syntax:**

inline return-type function-name(parameters)

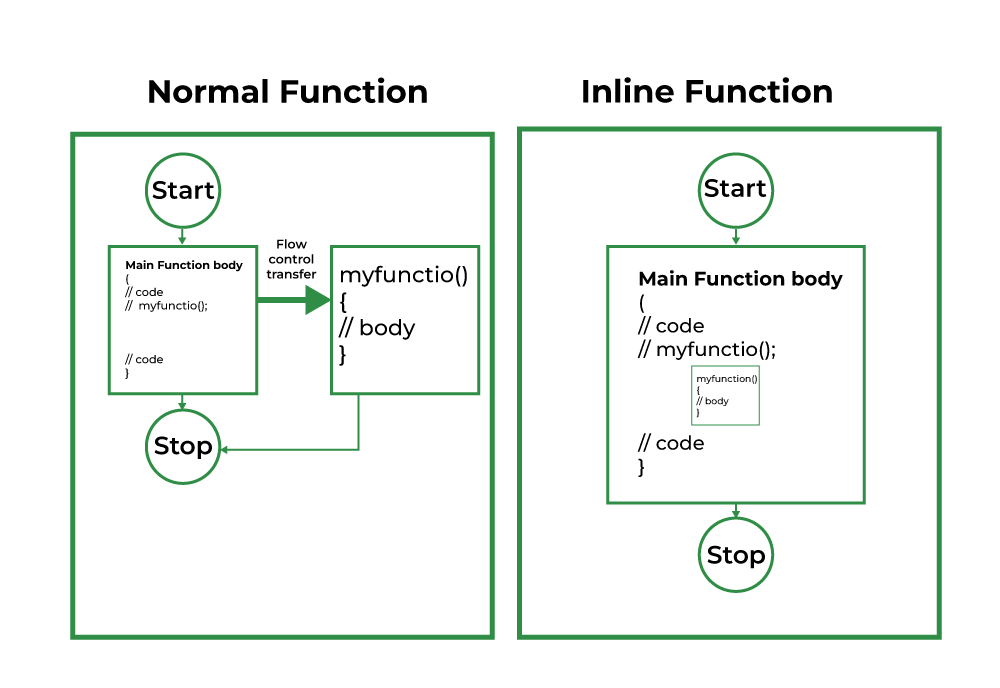
{

// function code

}

**The compiler may not perform inlining in such circumstances as:**

1. If a function contains a loop. (*for, while and do-while*)
2. If a function contains static variables.
3. If a function is recursive.
4. If a function return type is other than void, and the return statement doesn’t exist in a function body.
5. If a function contains a switch or goto statement.



**Example:**

|  |
| --- |
| #include <iostream>  using namespace std;  inline int cube(int s) { return s \* s \* s; }  int main()  {      cout << "The cube of 3 is: " << cube(3) << "\n";      return 0;  } |

**Output**

The cube of 3 is: 27

## ****Inline function and classes****

It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare an inline function in the class then just declare the function inside the class and define it outside the class using the inline keyword.

**Syntax:**

class S

{

public:

inline int square(int s) // redundant use of inline

{

// this function is automatically inline

// function body

}

};

The above style is considered a bad programming style. The best programming style is to just write the prototype of the function inside the class and specify it as an inline in the function definition.

**For Example:**

class S

{

public:

int square(int s); // declare the function

};

inline int S::square(int s) // use inline prefix

{

}

**// C++ Program to demonstrate inline functions and classes**

#include <iostream>

using namespace std;

class operation {

int a, b, add, sub, mul;

float div;

public:

void get();

void sum();

void difference();

void product();

void division();

};

inline void operation ::get()

{ cout << "Enter first value:";

cin >> a;

cout << "Enter second value:";

cin >> b;

}

inline void operation ::sum()

{

add = a + b;

cout << "Addition of two numbers: " << a + b << "\n";

}

inline void operation ::difference()

{

sub = a - b;

cout << "Difference of two numbers: " << a - b << "\n";

}

inline void operation ::product()

{

mul = a \* b;

cout << "Product of two numbers: " << a \* b << "\n";

}

inline void operation ::division()

{

div = a / b;

cout << "Division of two numbers: " << a / b << "\n";

}

int main()

{

cout << "Program using inline function\n";

operation s;

s.get();

s.sum();

s.difference();

s.product();

s.division();

return 0;

}

**Output:**

Enter first value: 45

Enter second value: 15

Addition of two numbers: 60

Difference of two numbers: 30

Product of two numbers: 675

Division of two numbers: 3

## ****Inline functions Advantages:****

1. Function call overhead doesn’t occur.
2. It also saves the overhead of push/pop variables on the stack when a function is called.
3. It also saves the overhead of a return call from a function.
4. When you inline a function, you may enable the compiler to perform context-specific optimization on the body of the function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of the calling context and the called context.

## ****Inline function Disadvantages:****

1. The added variables from the inlined function consume additional registers, After the in-lining function if the variable number which is going to use the register increases then they may create overhead on register variable resource utilization. This means that when the inline function body is substituted at the point of the function call, the total number of variables used by the function also gets inserted. So the number of registers going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause overhead on register utilization.
2. If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of the same code.
3. Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.
4. The inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because the compiler would be required to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

# C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

## Declaration of friend function in C++

1. **class** class\_name
2. {
3. **friend** data\_type function\_name(argument/s);            // syntax of friend function.
4. };

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

**Characteristics of a Friend function:**

* The function is not in the scope of the class to which it has been declared as a friend.
* It cannot be called using the object as it is not in the scope of that class.
* It can be invoked like a normal function without using the object.
* It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
* It can be declared either in the private or the public part.

## C++ friend function Example

Let's see the simple example of C++ friend function used to print the length of a box.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Box
4. {
5. **private**:
6. **int** length;
7. **public**:
8. Box(): length(0) { }
9. **friend** **int** printLength(Box); //friend function
10. };
11. **int** printLength(Box b)
12. {
13. b.length += 10;
14. **return** b.length;
15. }
16. **int** main()
17. {
18. Box b;
19. cout<<"Length of box: "<< printLength(b)<<endl;
20. **return** 0;
21. }

**Output:**

Length of box: 10

**Let's see a simple example when the function is friendly to two classes.**

1. #include <iostream>
2. **using** **namespace** std;
3. **class** B;          // forward declarartion.
4. **class** A
5. {
6. **int** x;
7. **public**:
8. **void** setdata(**int** i)
9. {
10. x=i;
11. }
12. **friend** **void** min(A,B);         // friend function.
13. };
14. **class** B
15. {
16. **int** y;
17. **public**:
18. **void** setdata(**int** i)
19. {
20. y=i;
21. }
22. **friend** **void** min(A,B);                    // friend function
23. };
24. **void** min(A a,B b)
25. {
26. **if**(a.x<=b.y)
27. std::cout << a.x << std::endl;
28. **else**
29. std::cout << b.y << std::endl;
30. }
31. **int** main()
32. {
33. A a;
34. B b;
35. a.setdata(10);
36. b.setdata(20);
37. min(a,b);
38. **return** 0;
39. }

**Output:**

10

In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes A and B.

### C++ Friend class

A friend class can access both private and protected members of the class in which it has been declared as friend.

**Let's see a simple example of a friend class.**

1. #include <iostream>
3. **using** **namespace** std;
5. **class** A
6. {
7. **int** x =5;
8. **friend** **class** B;           // friend class.
9. };
10. **class** B
11. {
12. **public**:
13. **void** display(A &a)
14. {
15. cout<<"value of x is : "<<a.x;
16. }
17. };
18. **int** main()
19. {
20. A a;
21. B b;
22. b.display(a);
23. **return** 0;
24. }

**Output:**

value of x is : 5

In the above example, class B is declared as a friend inside the class A. Therefore, B is a friend of class A. Class B can access the private members of class A.

**Virtual Function in C++**

A virtual function is a member function which is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the function.

* Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
* They are mainly used to achieve [Runtime polymorphism](https://www.geeksforgeeks.org/polymorphism-in-c/)
* Functions are declared with a **virtual**keyword in base class.
* The resolving of function call is done at runtime.

**Rules for Virtual Functions**

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have [virtual destructor](https://www.geeksforgeeks.org/virtual-destructor/) but it cannot have a virtual constructor.

// CPP program to illustrate

// concept of Virtual Functions

#include<iostream>

using namespace std;

class base {

public:

virtual void print()

{

cout << "print base class\n";

}

void show()

{

cout << "show base class\n";

}

};

class derived : public base {

public:

void print()

{

cout << "print derived class\n";

}

void show()

{

cout << "show derived class\n";

}

};

int main()

{

base \*bptr;

derived d;

bptr = &d;

// Virtual function, binded at runtime

bptr->print();

// Non-virtual function, binded at compile time

bptr->show();

return 0;

}

**Output:**

print derived class

show base class

* It is a run-time polymorphism.
* Both the base class and the derived class have the same function name, and the base class is assigned with an address of the derived class object then also pointer will execute the base class function.
* If the function is made virtual, then the compiler will determine which function is to execute at the run time on the basis of the assigned address to the pointer of the base class.

### ****Limitations of Virtual Functions:****

* **Slower:**The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.
* **Difficult to Debug:**In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

**Pure Virtual Functions**

**Characteristics of a pure virtual function**

* A pure virtual function is a "do nothing" function. Here "do nothing" means that it just provides the template, and derived class implements the function.
* It can be considered as an empty function means that the pure virtual function does not have any definition relative to the base class.
* Programmers need to redefine the pure virtual function in the derived class as it has no definition in the base class.
* A class having pure virtual function cannot be used to create direct objects of its own. It means that the class is containing any pure virtual function then we cannot create the object of that class. This type of class is known as an abstract class.

**Syntax**

There are two ways of creating a virtual function:

**virtual** **void** display() = 0;

or

**virtual** **void** display() {}

1. #include <iostream>
2. **using** **namespace** std;
3. // Abstract class
4. **class** Shape
5. {
6. **public**:
7. **virtual** **float** calculateArea() = 0; // pure virtual function.
8. };
9. **class** Square : **public** Shape
10. {
11. **float** a;
12. **public**:
13. Square(**float** l)
14. {
15. a = l;
16. }
17. **float** calculateArea()
18. {
19. **return** a\*a;
20. }
21. };
22. **class** Circle : **public** Shape
23. {
24. **float** r;
25. **public**:
27. Circle(**float** x)
28. {
29. r = x;
30. }
31. **float** calculateArea()
32. {
33. **return** 3.14\*r\*r ;
34. }
35. };
36. **class** Rectangle : **public** Shape
37. {
38. **float** l;
39. **float** b;
40. **public**:
41. Rectangle(**float** x, **float** y)
42. {
43. l=x;
44. b=y;
45. }
46. **float** calculateArea()
47. {
48. **return** l\*b;
49. }
50. };
51. **int** main()
52. {
54. Shape \*shape;
55. Square s(3.4);
56. Rectangle r(5,6);
57. Circle c(7.8);
58. shape =&s;
59. **int** a1 =shape->calculateArea();
60. shape = &r;
61. **int** a2 = shape->calculateArea();
62. shape = &c;
63. **int** a3 = shape->calculateArea();
64. std::cout << "Area of the square is " <<a1<< std::endl;
65. std::cout << "Area of the rectangle is " <<a2<< std::endl;
66. std::cout << "Area of the circle is " <<a3<< std::endl;
67. **return** 0;
68. }

|  |  |
| --- | --- |
| **Virtual function** | **Pure virtual function** |
| A virtual function is a member function in a base class that can be redefined in a derived class. | A pure virtual function is a member function in a base class whose declaration is provided in a base class and implemented in a derived class. |
| The classes which are containing virtual functions are not abstract classes. | The classes which are containing pure virtual function are the abstract classes. |
| In case of a virtual function, definition of a function is provided in the base class. | In case of a pure virtual function, definition of a function is not provided in the base class. |
| The base class that contains a virtual function can be instantiated. | The base class that contains a pure virtual function becomes an abstract class, and that cannot be instantiated. |
| If the derived class will not redefine the virtual function of the base class, then there will be no effect on the compilation. | If the derived class does not define the pure virtual function; it will not throw any error but the derived class becomes an abstract class. |
| All the derived classes may or may not redefine the virtual function. | All the derived classes must define the pure virtual function. |

**Abstract Classes in C++**

*A class is abstract if it has at least one pure virtual function.*   
In the following example, Test is an abstract class because it has a pure virtual function show().

// pure virtual functions make a class abstract  
#include  
using namespace std;

class Test  
{  
int x;  
public:  
virtual void show() = 0;  
int getX() { return x; }  
};

int main(void)  
{  
Test t;  
return 0;  
}

**Output:**

Compiler Error: cannot declare variable 't' to be of abstract

type 'Test' because the following virtual functions are pure

within 'Test': note: virtual void Test::show()

## C++ Pure Virtual Functions

Pure virtual functions are used

* if a function doesn't have any use in the base class
* but the function must be implemented by all its derived classes

Let's take an example,

Suppose, we have derived Triangle, Square and Circle classes from the Shape class, and we want to calculate the area of all these shapes.

In this case, we can create a pure virtual function named calculateArea() in the Shape. Since it's a pure virtual function, all derived classes Triangle, Square and Circle must include the calculateArea() function with implementation.

A pure virtual function doesn't have the function body and it must end with = 0. For example,

class Shape {

public:

// creating a pure virtual function

virtual void calculateArea() = 0;

};

**Note:** The = 0 syntax doesn't mean we are assigning 0 to the function. It's just the way we define pure virtual functions.

## Abstract Class

A class that contains a pure virtual function is known as an abstract class. In the above example, the class Shape is an abstract class.

We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions (except pure virtual functions).

// C++ program to calculate the area of a square and a circle

#include <iostream>

using namespace std;

// Abstract class

class Shape {

protected:

float dimension;

public:

void getDimension() {

cin >> dimension;

}

// pure virtual Function

virtual float calculateArea() = 0;

};

// Derived class

class Square : public Shape {

public:

float calculateArea() {

return dimension \* dimension;

}

};

// Derived class

class Circle : public Shape {

public:

float calculateArea() {

return 3.14 \* dimension \* dimension;

}

};

int main() {

Square square;

Circle circle;

cout << "Enter the length of the square: ";

square.getDimension();

cout << "Area of square: " << square.calculateArea() << endl;

cout << "\nEnter radius of the circle: ";

circle.getDimension();

cout << "Area of circle: " << circle.calculateArea() << endl;

return 0;

}

**Output**

Enter the length of the square: 4

Area of square: 16

Enter radius of the circle: 5

Area of circle: 78.5

In this program, virtual float calculateArea() = 0; inside the Shape class is a pure virtual function.

That's why we must provide the implementation of calculateArea() in both of our derived classes, or else we will get an error.

