

CS-1004 Object Oriented programming

Week 9

Instructor:

ATIYA

LECTURER

(COMPUTER SCIENCE DEPARTMENT)

NATIONAL UNIVERSITY- FAST (KHI CAMPUS)

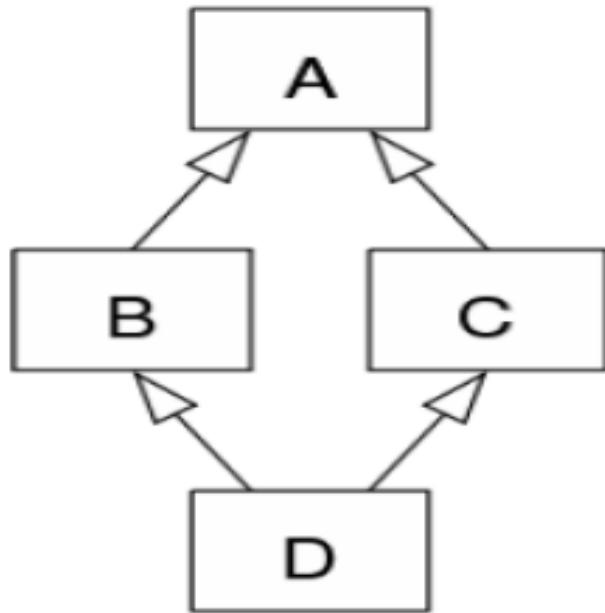
EMAIL: ATIYA.JOKHIO@NU.EDU.PK

ROOM: (LECTURER ROOM#19, CS DEPT BASEMENT-2)

Diamond problem

- ✓ In case of hybrid inheritance, a Diamond problem may arise. The “dreaded diamond” refers to a class structure in which a particular class appears more than once in a class’s inheritance hierarchy.
- ✓ Consider the situation where we have one class **A**. This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.

diamond problem



- A is the base class.
- B and C inherit from A ($B \rightarrow A$ and $C \rightarrow A$).
- D inherits from both B and C ($D \rightarrow B$ and $D \rightarrow C$).
- The issue arises because D gets **two copies of A**, leading to ambiguity.

```
class A{
    int a;
};
class B : public A{
};
class C : public A{
};
class D : public B , public C{
};
int main(){
    cout << sizeof(A) << endl;
    cout << sizeof(B) << endl;
    cout << sizeof(C) << endl;
    cout << sizeof(D) << endl;
}
```

```
4
4
4
8
```

```
-----
```

```
Process exited after 7.799 seconds with return value 0
Press any key to continue . . .
```

```
class A{
public:
int a;};
class B : public A{
    public:
    int b;};
class C : public A{
    public:
    int c;};
class D : public B , public C{
    public:
    int d;};
int main(){
    cout << sizeof(A) << endl;
    cout << sizeof(B) << endl;
    cout << sizeof(C) << endl;
    cout << sizeof(D) << endl;}
```

```
4
8
8
20
```

```
-----
Process exited after 7.704 seconds with return value 0
Press any key to continue . . .
```

diamond problem

```
class A{
    public:
    int a;};
class B : public A{
    public:
    int b;};
class C : public A{
    public:
    int c;};
class D : public B , public C{
    public:
    int d;};
```


diamond problem

```
int main(){  
    D d;  
    d.a;  
    return 0;}
```

This problem is also known as ***diamond problem***

From above example we can see that “A” is inherited two times in D means an object of class “D” will contain two attributes of “a” (D::C::a and D::B::a).

Message

In function 'int main()':

[Error] request for member 'a' is ambiguous

[Note] candidates are: int A::a

[Note] int A::a

Diamond problem

```
1  #include <iostream>
2  using namespace std;
3  class A {
4      void displayA()
5      {
6          cout<<"A display "<<endl;
7      }
8  };
9
10 class B : public A{
11     void displayB()
12     {
13         cout<<"b display "<<endl;
14     }
15 };
16
17 class C : public A{
18     void displayC()
19     {
20         cout<<"c display "<<endl;
21     }
```

```
21     }
22 };
23
24 class D : public B, public C{
25
26 };
27 int main(){
28     D objd;
29     objd.displayA();
30     return 0;
31 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
daimond.cpp: In function 'int main()':
daimond.cpp:29:10: error: request for member 'display' is ambiguous
29 |     objd.display();
   |           ^~~~~~
daimond.cpp:4:7: note: candidates are: 'void A::display()'
4 |     void display()
   |           ^~~~~~
daimond.cpp:4:7: note: 'void A::display()'
```

This problem is also known as ***diamond problem***

As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

Memory layout of class “D” object:

Possible “D” object layout
A
B
A
C
D

Supposed memory layout of class “D” object

How to resolve this issue?

To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as :

How to resolve this issue?

1. There will be new virtual base pointers which would be inserted by compiler when we are using virtual inheritance.
2. This virtual base pointer will provide an indirect access to class “A” object
3. Identify the problematic class and use virtual keyword in all child classes that are associated with problematic class. In previous example problematic class is A that occurs more than once so inherit it by using virtual keyword

virtual base class / virtual inheritance

Syntax 1:

```
class B : virtual public A
{
};
```

Syntax 2:

```
class C : public virtual A
{
};
```


virtual base class / virtual inheritance

Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

virtual base class / virtual inheritance

which single copy of its data members is shared?

both base classes (class B and class C) use virtual base.

virtual base class / virtual inheritance

- ✓ it varies from compiler to compiler
- ✓ Depends on compiler which path is selected to access grand parents' data from grand child
- ✓ Usually, Last updated copy of its data members is shared by all the base classes that use virtual base
- ✓ The right most class being inherited is the one whose data member is going to come in derived class

Solution

```
#include <iostream>
using namespace std;
class A {
    public:
    void displayA()
    {
        cout<<"A display "<<endl;
    }
};

class B : virtual public A{
    void displayB()
    {
        cout<<"b display "<<endl;
    }
};
```

```
class C : virtual public A{
    void displayC()
    {
        cout<<"c display "<<endl;
    }
};

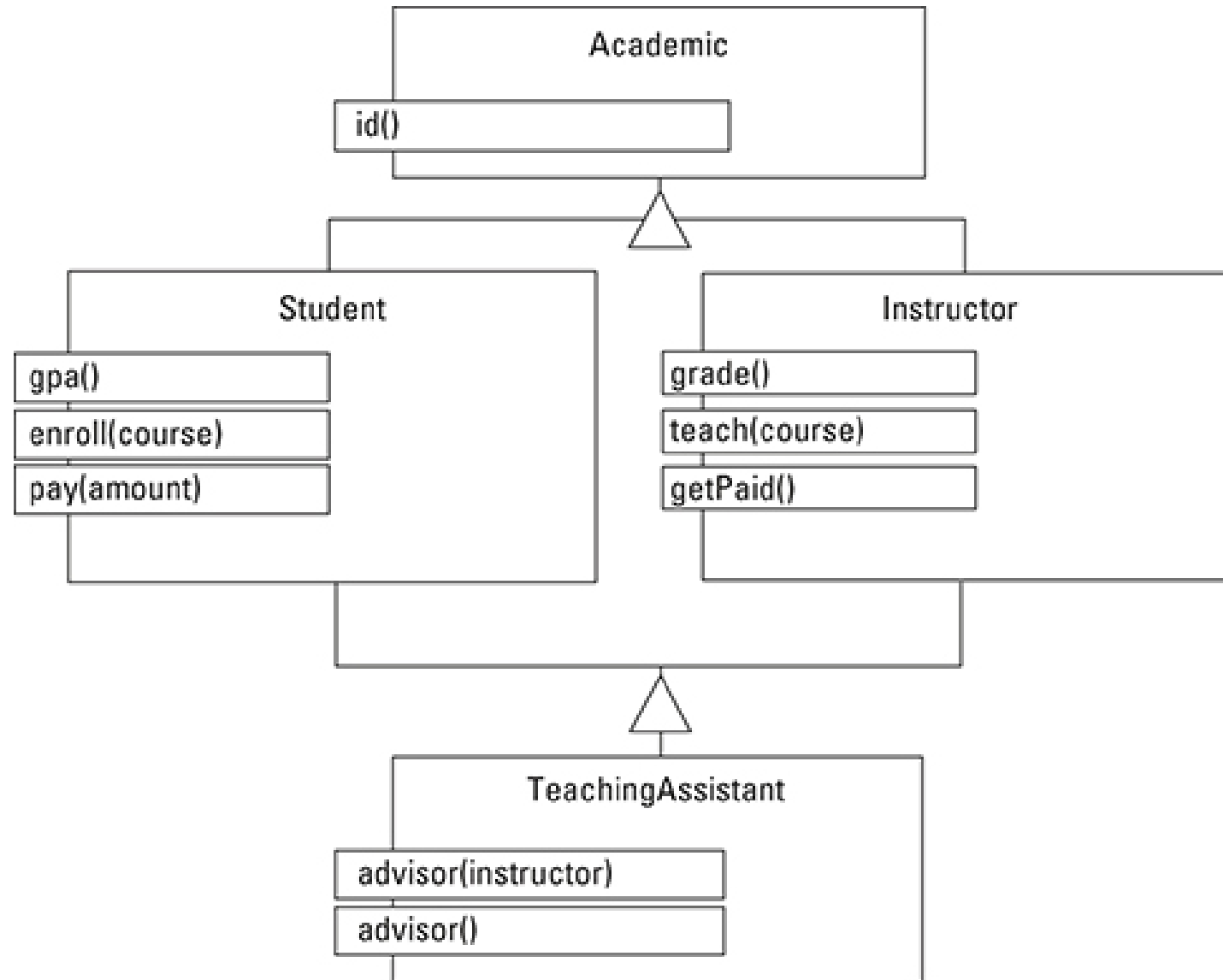
class D : public B, public C{
};

int main(){
    D objd;
    objd.displayA();
    return 0;
}
```

```
C:\Users\HP\Documents\example-01>a.exe
A display

C:\Users\HP\Documents\example-01>
```

Example



```

class Academic
{
    public:
        int id(int){
            cout<<"calling from teaching Assistant"<<endl;
            return 0;
        }
};

class Student : virtual public Academic
{
    public:
        double gpa();
        bool  enroll( );
        bool  pay(double dollars);
};

class Instructor : virtual  public Academic
{
    public:
        double grade();
        bool  teach();
        bool  getPaid(double dollars);
};

```

Solution

```

class TeachingAssistant:public Student, public Instructor
{
    public:
        Instructor advisor(Instructor&);
        Instructor advisor();
};

int main (){
    TeachingAssistant TA;
    cout << "TA's student id is " << TA.id(2); // no error now
    return 0;
}

```

virtual base class / virtual inheritance

```
class A {  
public:  
    int a;  
    A() {  
        a = 10;    }  
};  
class B : public virtual A {  
public:  
    B() {  
        a = 1;    }  
};  
class C : public virtual A {  
public:  
    C() {  
        a = 2;    }  
};  
class D : public B , public C {  
};
```

virtual base class / virtual inheritance

```
int main()
{
    D object; // object creation of class d
    cout << "a = " << object.a << endl;

    return 0;
}
```


virtual base class / virtual inheritance

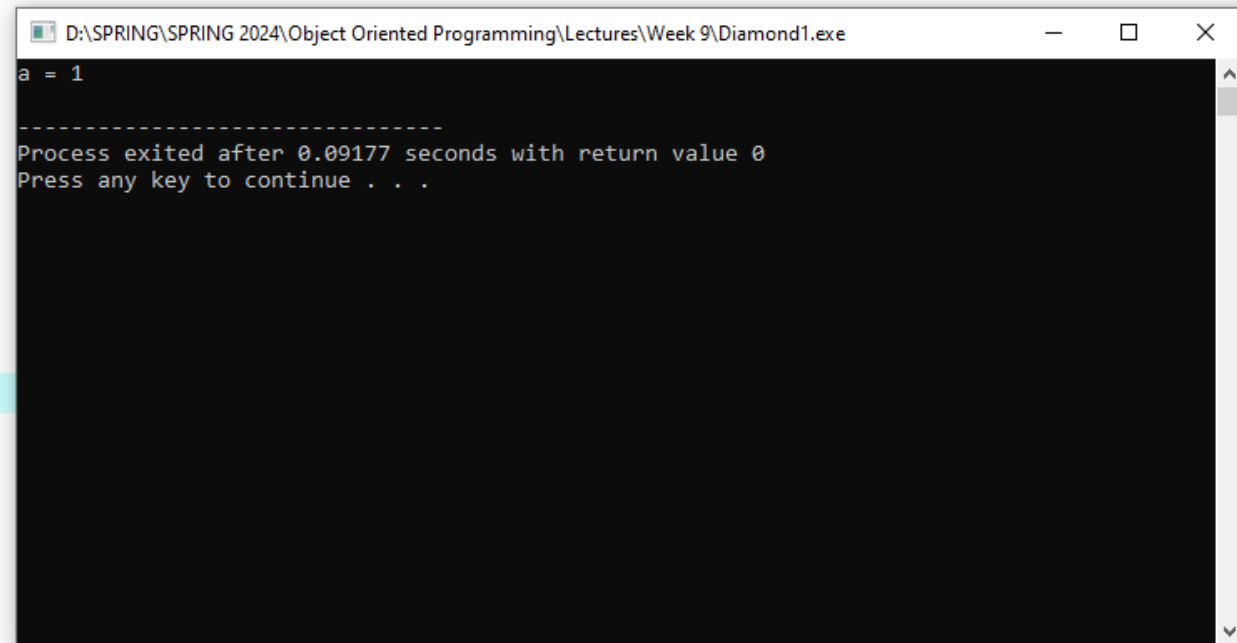
```
a = 2  
  
-----  
Process exited after 0.1109 seconds with return value 0  
Press any key to continue . . .
```

virtual base class / virtual inheritance

```
class A {
public:
    int a;
    A() {
        a = 10; }
};
class B : public virtual A {
public:
    B() {
        a = 1; }
};
class C : public virtual A {
public:
    C() {
        a = 2; }
};
class D : public C , public B {

};
int main()
{
    D object; // object creation of class d
    cout << "a = " << object.a << endl;

    return 0;
}
```



```
D:\SPRING\SPRING 2024\Object Oriented Programming\Lectures\Week 9\Diamond1.exe
a = 1
-----
Process exited after 0.09177 seconds with return value 0
Press any key to continue . . .
```

virtual base class / virtual inheritance

```
class D : public C , public B {  
    public:  
    D() {  
        a = 5;} // now the last updated value will be 5  
    getval(){  
        cout << a;}};
```

Explanation

You created object of class D compiler will choose one path from grand child to grand parent either from Class B or Class C

In this case compiler will choose right most inherited class that is class B and print the value accordingly that's why it will hide the value updated in Class C

If you want to access the updated value of class C you have to create the object of class C instead of class D it works normally

Without virtual (parameterized Constructor)

```
class A {  
public:  
    A(int a) {  
        cout << a << endl;    }  
};  
class B : public A {  
public:  
    B():A(2) { }  
};  
class C : public A {  
public:  
    C():A(3) { }  
};  
class D : public B ,public C {  
public:  
    D() { }  
};  
int main(){  
    D object; // object creation of class d  
    return 0;}
```

2







3

Process exited after 0.1011 seconds with return value 0

Press any key to continue . . .

With virtual

```
class A {
public:
    A(int a) {
        cout << a << endl;  };
class B : virtual public A {
public:
    B():A(2) {  };
class C : virtual public A {
public:
    C():A(3) { };
class D : public B ,public C {
public:
    D() {      };
int main(){
    D object; // object creation of class d
    return 0;}
```

<div>  Compiler (7)  Resources  Compile Log  Debug  Find Results  Close </div>			
Line	Col	File	Message
		D:\SPRING\SPRING 2024\Object Oriented Programm...	In constructor 'D::D0':
15	6	D:\SPRING\SPRING 2024\Object Oriented Programming\...	[Error] no matching function for call to 'A::A()'
15	6	D:\SPRING\SPRING 2024\Object Oriented Programming\...	[Note] candidates are:
5	2	D:\SPRING\SPRING 2024\Object Oriented Programming\...	[Note] A::A(int)
5	2	D:\SPRING\SPRING 2024\Object Oriented Programming\...	[Note] candidate expects 1 argument, 0 provided
3	7	D:\SPRING\SPRING 2024\Object Oriented Programming\...	[Note] A::A(const A&)
3	7	D:\SPRING\SPRING 2024\Object Oriented Programminq\...	[Note] candidate expects 1 arqument, 0 provided

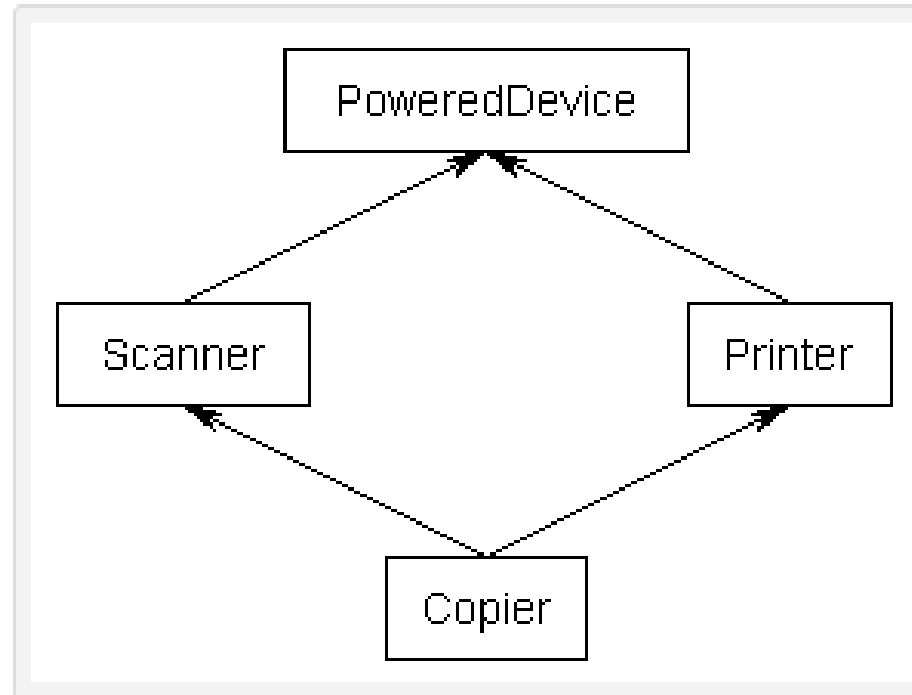
Now D is responsible to give value to its indirect parent class

```
#include <iostream>
using namespace std;
class A {
public:
    A(int a) {
        cout << a << endl;    };
};
class B : virtual public A {
public:
    B():A(2)//compiler will ignore this but you cannot skip this line for hirarichal inheritance {
};
class C : virtual public A {
public:
    C():A(3)//compiler will ignore this but you cannot skip this line for hirarichal inheritance {
};
```

```
class D : public B ,public C {  
    public:  
    D():A(1) {        }  
};  
  
int main()  
{  
    D object; // object creation of class d  
    return 0;  
}
```

```
1
-----
Process exited after 0.09129 seconds with return value 0
Press any key to continue . . .
```

Identify the problematic class?



Identify the problematic class through parametrized constructor ?

```
class PoweredDevice{
public:
    PoweredDevice(int power){
        cout << "PoweredDevice: " << power << "\n";}};

class Scanner: public PoweredDevice{
public:
    Scanner(int scanner, int power)
        : PoweredDevice(power) {
        cout << "Scanner: " << scanner << "\n"; }
};
```

Identify the problematic class through parametrized constructor ?

```
class Printer: public PoweredDevice{
public:
    Printer(int printer, int power)
        : PoweredDevice(power) {
        cout << "Printer: " << printer << '\n'; }
};

class Copier: public Scanner, public Printer{
public:
    Copier(int scanner, int printer, int power)
        : Scanner(scanner, power), Printer(printer, power) {
    }
};
```

Identify the problematic class through parametrized constructor ?

```
int main()
{
    Copier copier(1, 2, 3);

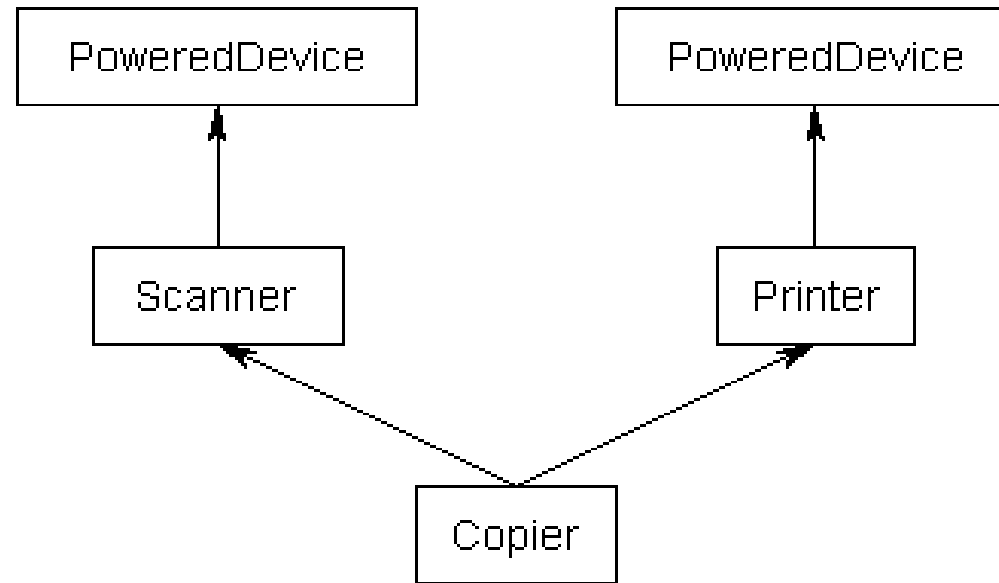
    return 0;}
```

If you were to create a Copier class object, by default you would end up with two copies of the PoweredDevice class -- one from Printer, and one from Scanner. This has the following structure:

Identify the problematic class through parametrized constructor ?

```
PoweredDevice: 3  
Scanner: 1  
PoweredDevice: 3  
Printer: 2  
  
-----  
Process exited after 0.2705 seconds with return value 0  
Press any key to continue . . .
```


Identify the problematic class through parametrized constructor ?



Solution By using Virtual Base class

```
#include <iostream>
class PoweredDevice {
public:
    PoweredDevice(int power) { std::cout << "PoweredDevice: " << power
<< '\n'; }; };
    class Scanner: virtual public PoweredDevice
    {
public:
    Scanner(int scanner, int power)
        : PoweredDevice(power) // this line is required to create Scanner
objects, but ignored in this case because we are working with copier
object
        { std::cout << "Scanner: " << scanner << '\n'; }; };
};
```

Solution By using Virtual Base class

```
class Printer: virtual public PoweredDevice // note: PoweredDevice is
now a virtual base class

{

public:

    Printer(int printer, int power)

        : PoweredDevice(power) // this line is required to create
Printer objects, but ignored in this case because we are working with
copier object

    { std::cout << "Printer: " << printer << '\n';

    }

};
```

Solution By using Virtual Base class

```
class Copier: public Scanner, public Printer
{
public:
    Copier( int scanner ,int printer, int power)
        : PoweredDevice(power), // PoweredDevice is constructed here
        Scanner(scanner, power),Printer(printer, power)
    {
    }
};

int main()
{
    Copier copier(1, 2, 3);
    return 0;}
```

Solution By using Virtual Base class

```
PoweredDevice: 3  
Scanner: 1  
Printer: 2  
  
-----  
Process exited after 0.3112 seconds with return value 0  
Press any key to continue . . .
```

Solution By using Virtual Base class

Now, when you create a Copier class object, you will get only one copy of PoweredDevice per Copier that will be shared by both Scanner and Printer.

However, this leads to one more problem: if Scanner and Printer share a PoweredDevice base class, who is responsible for creating it? The answer, as it turns out, is Copier. The Copier constructor is responsible for creating PoweredDevice. Consequently, this is one time when Copier is allowed to call a non-immediate-parent constructor directly if we not use virtual here we cannot access non-immediate-parent directly it will cause an compile time error

Friend Function and Friend Classes

Friend Function

- Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.
- Similarly, protected members can only be accessed by derived classes and are inaccessible from outside. For example,

```
class MyClass {  
    private:  
        int member1;  
}  
  
int main() {  
    MyClass obj;  
  
    // Error! Cannot access private members from here.  
    obj.member1 = 5;  
}
```


Friend Function

- There is a feature in C++ called **friend functions** that break this rule and allow us to access member functions from outside the class.
- A friend function can access the private and protected data of a class.
- We declare a friend function using the friend keyword inside the body of the class.
- A friend function does not use the dot (.) or arrow (→) like member functions

```
class className {  
    ... ..  
    friend returnType functionName(arguments);  
    ... ..  
}
```

Friend Functions

➤ A **friend function** of a class is defined outside that class's scope yet has the right to access the non-public (and public) members of the class.

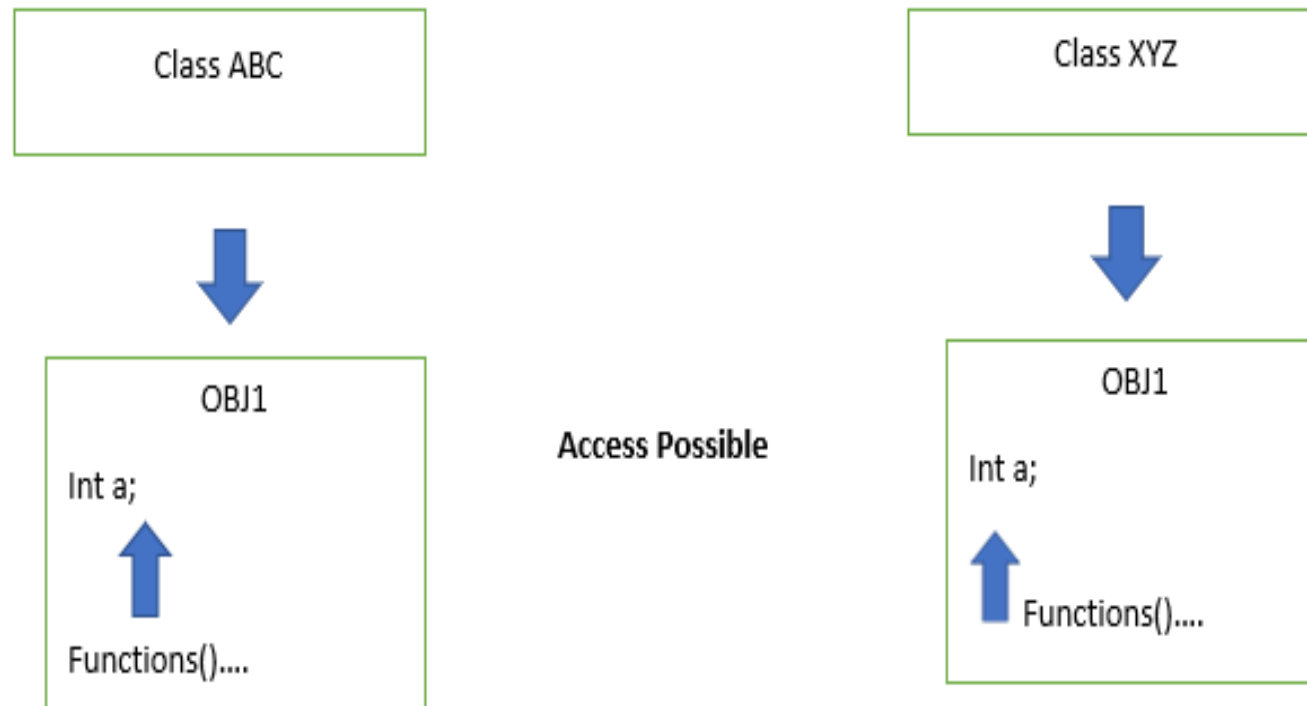
Friend functions can be

- Standalone functions,
- entire classes or
- member functions of other classes may be declared to be friends of another class.
- The functions which are not member functions of the class yet they can access all private members of the class are called friend functions.
- The compiler knows a given function is a friend function by the use of the keyword **friend**.

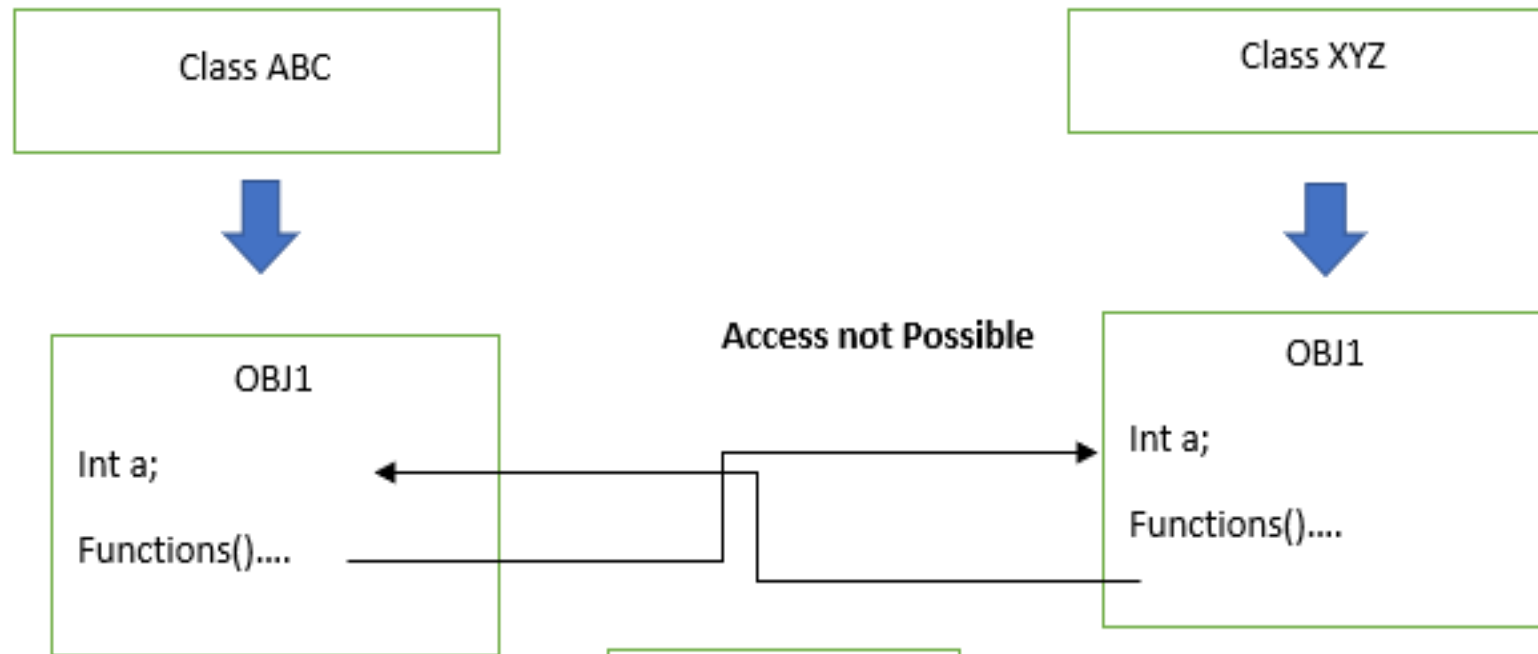
Why they are needed?

They are needed in situations where we have written code for some function in one class and it need to be used by other classes as well for example, suppose we wrote the code to compute a complex mathematical formula in one class but later it was required by other classes as well, in that case we will make that function friend of all other classes.

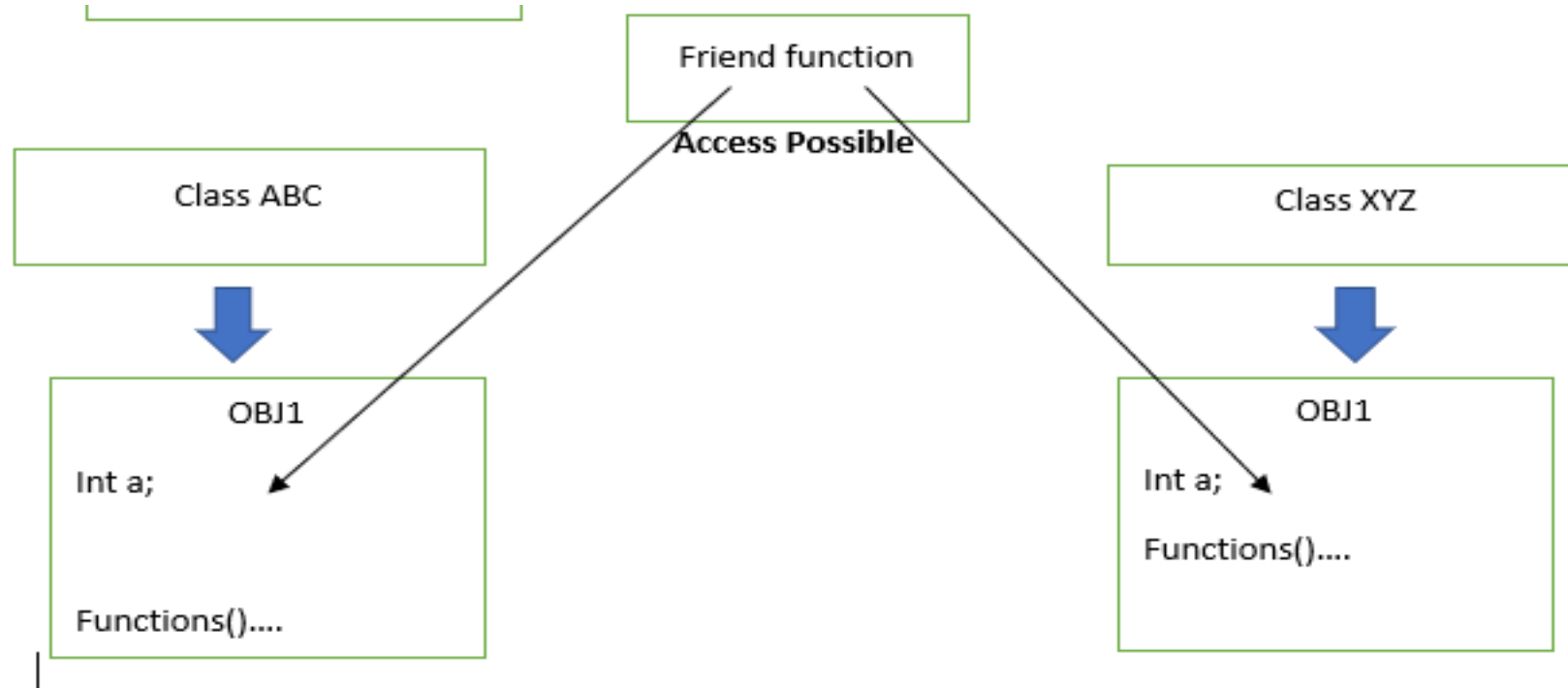
Why they are needed?



Why they are needed?



Why they are needed?



Are friend functions against the concept of Object Oriented Programming?

- One of the important concepts of OOP is data, but a non-member function cannot access an object's private or protected data.
- But, sometimes this restriction may force programmer to write long and complex codes. So, there is mechanism built in C++ programming to access private or protected data from non-member functions. This is done using a friend function or/and a friend class.
- It can be said that friend functions are against the principle of object oriented programming because they violate the principle of encapsulation which clearly says that each object methods and functions should be encapsulated in it. But there we are making our private member accessible to other outside functions.

Friend Functions

- Prototypes of friend functions appear in the class definition. But friend functions are NOT member functions.
- Friend functions can be placed anywhere in the class without any effect Access specifiers don't affect friend functions or Classes.

A friend function can be:

1. A global function
2. A member function of another class

Consider the following class

```
class X {  
private:  
int a, b;  
public:  
void MemberFunction (); }
```

Suppose we have a global function DoSomething that need to access the private members of class X, when we will try to access them compiler will generate error as outside world cannot access private members of a class except its member functions.

Friend Functions

```
void DoSomething(X obj) {  
obj.a = 3; //Error  
obj.b = 4; //Error }
```

In order to access the member variables of the class, we must make function friend of that class

Friend Functions

```
class X {  
private:  
int a, b;  
public:  
friend void DoSomething(X obj); }
```

Now the function DoSomething can access data members of class X

```
void DoSomething(X obj) {  
obj.a = 3;  
obj.b = 4; }
```

Friend Functions

```
class X{
```

```
...
```

```
private:
```

```
friend void DoSomething(X);
```

```
public:
```

```
friend void DoAnything(X);
```

```
...
```

```
};
```

While the definition of the friend function is:

```
void DoSomething(X obj)
```

```
{ obj.a = 3; // No Error
```

```
obj.b = 4; // No Error
```

```
...
```

```
}
```

friend keyword is not given in definition.

Friend Function is not defined as Member function defined Outside class

- ```
class MyClass { // The class
 public: // Access specifier
 void myMethod(); // Method/function declaration
};

// Method/function definition outside the class
void MyClass::myMethod() {
 cout << "Hello World!";
}

int main() {
 MyClass myObj; // Create an object of MyClass
 myObj.myMethod(); // Call the method
 return 0;
}
```

# global Friend Function Example1

---

```
#include <iostream>
using namespace std;
class Sample {
private:
 int num;

public:
 void setData(int n) {
 num = n;
 }

 // Friend function declaration
 friend void display(Sample obj);
};
```

```
// Friend function definition
void display(Sample obj) {
 cout << "The value of num is: " << obj.num
 << endl;
}

int main() {
 Sample s;
 s.setData(10);
 display(s); // Calling the friend function

 return 0;
}
```

# global Friend Function Example2

---

```
// C++ program to demonstrate the working
of friend function
```

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

class Distance {
 private:
 int meter;
 // friend function
 friend int addFive(Distance);
 public:
 Distance() : meter(0) {}
};
```

```
// friend function definition
int addFive(Distance d) {

 //accessing private members from the
friend function
 d.meter += 5;
 return d.meter;
}

int main() {
 Distance D;

 cout << "Distance: " << addFive(D);
 return 0;
}
```

# global Friend Function Example3

---

```
#include <iostream>
using namespace std;
class Box {
 double width;
public:
 friend void printWidth(Box box);
 void setWidth(double wid);};
void Box::setWidth(double wid) {
 width = wid;}
void printWidth(Box box) {
 cout << "Width of box : " << box.width << endl;}
```



# global friend function

---

```
int main() {
 Box box;
 box.setWidth(10.0);
 printWidth(box);
 return 0;
}
```

# global friend function

---

```
Width of box : 10

Process exited after 0.2686 seconds with return value 0
Press any key to continue . . .
```

# friend function of another class

---

member functions of other classes may be declared to be friends of another class

```
friend return_type class_name::function_name (arguments); // for a member
function of another class
```

# friend function of another class

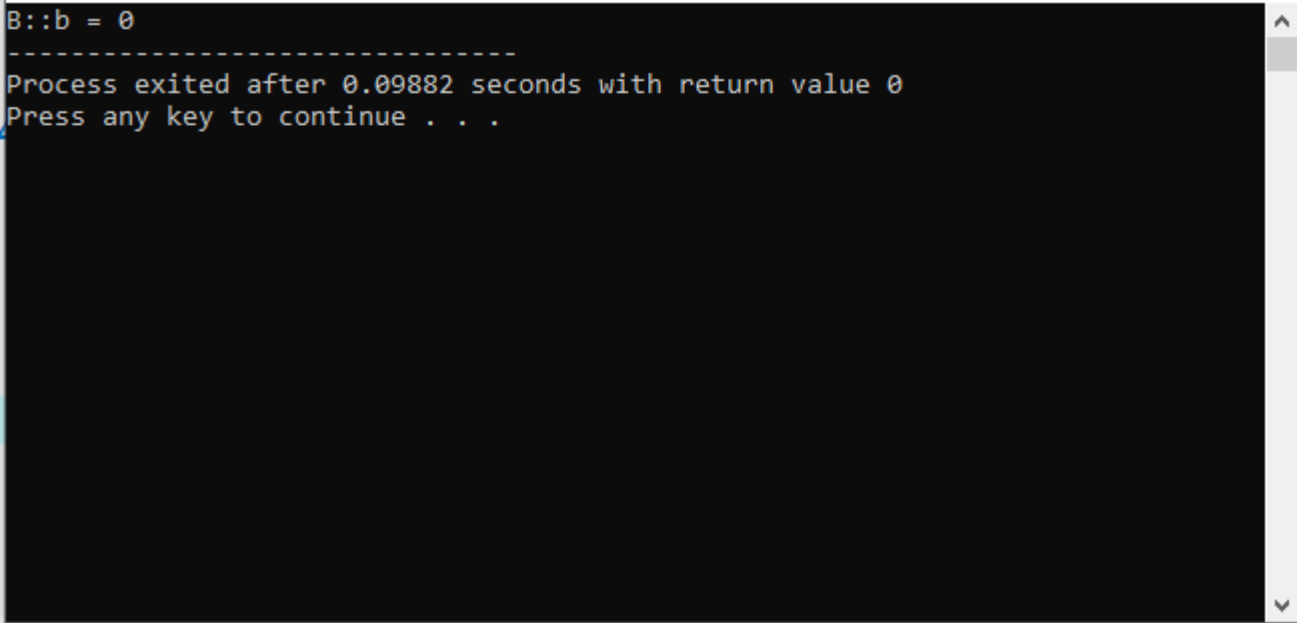
---

```
class B; // forward declaration of B needed by A
class A {
public:
void showB (B);
};
class B {
int b;
public:
B() {
b = 0; }
friend void A::showB(B x);}; // Friend function
void A::showB(B x) {
cout << "B::b = " << x.b;}
```

# friend function of another class

---

```
int main(){
 A a;
 B x;
 a.showB(x);
 return 0;}
```



```
B::b = 0

Process exited after 0.09882 seconds with return value 0
Press any key to continue . . .
```

# Inline-Function

---

The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called.

One of the major objectives of using functions in a program is to save memory space, which becomes appreciable when a function is likely to be called many times.

An inline function is a function that is expanded in line when it is invoked thus saving time.

# Why we need?

---

However, every time a function is called, it takes a lot of extra time in executing tasks such as jumping to the calling function.

When a function is small, a substantial percentage of execution time may be spent in such overheads and sometimes maybe the time taken for jumping to the calling function will be greater than the time taken to execute that function.

program example

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

inline int mul(int x, int y)
{
 return(x*y);
}

int main(){
 int a=4, b=2;
 count<<mul(a,b);
 return 0;
}
```

The diagram illustrates the concept of inline functions. It shows a C++ code snippet with a function `mul` and its call in `main`. Red annotations highlight the `return(x*y);` line in the function definition, labeling it as the "function body". Another red annotation points to the `mul(a,b);` call in the `main` function, labeling it as the "function call". A green arrow points from the function call to the function body, with the text "The call is completely replaced by the function body" next to it, indicating that the call is substituted with the actual code of the function.

# Inline Function

---

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

- 1) If a function contains a loop. (for, while, do-while)
- 2) If a function contains static variables.
- 3) If a function is recursive.
- 5) If a function contains switch or goto statement.



# Inline Function

```
1 #include <iostream>
2 using namespace std;
3 int cube(int x) https://t4tutorials.com/
4 {
5 int result = x*x*x;
6 return result;
7 }
8 int main(void)
9 {
10 int x=2;
11 cout<<cube(x);
12 }
```

Diagram illustrating a simple function call. A green box highlights the `cube` function definition (lines 3-7). A blue box highlights the function call `cube(x)` in the `main` function (line 11). A green arrow labeled "Request" points from the function call to the function definition. A green arrow labeled "Response" points from the function definition back to the function call.

Figure: Simple Function

```
1 #include <iostream>
2 using namespace std;
3 inline int cube(int x) https://t4tutorials.com/
4 {
5 int result = x*x*x;
6 return result;
7 }
8 int main(void)
9 {
10 int x=2;
11 cout<<cube(x);
12 }
```

Diagram illustrating an inline function. A green box highlights the `inline` function definition (lines 3-7). A blue box highlights the function call `cube(x)` in the `main` function (line 11). A green arrow labeled "Request" points from the function call to the function definition. A green arrow labeled "Response" points from the function definition back to the function call. An orange box highlights the function call `cube(x)` and the text "no need to request the inline function" (line 11). An orange box highlights the function definition (lines 3-7) and the text "no need to request the inline function" (line 11). An orange box highlights the function call `cube(x)` and the text "no need to request the inline function" (line 11).

Figure: Inline Function

# Inline Functions

---

```
inline int Area(int len, int hi)
{
 return len * hi;
}

int main()
{
 cout << Area(10,20);
}
```

**After Compilation the code in main () will look like:**

---

```
int main()
```

```
{
```

```
cout << len * hi;
```

**OR**

```
cout << 10 * 20;
```

```
}
```

## Example of inline function.

---

```
class Student{
private:
int rollNo;
public:
void setRollNo(int aRollNo){
// by default it is inline because it is defined inside the class body.
rollNo = aRollNo;
}
};
```

## Another example with functions defined outside the class body:

---

```
class Student{
private:
... public:
inline void setRollNo(int aRollNo);
};
void Student::setRollNo(int aRollNo){
// no need to put inline in definition, as it is declared in the class as inline.
rollNo = aRollNo;
}
```

**Another example without using inline keyword in functions declaration:**

---

```
class Student{
private:
...
public:
void setRollNo(int aRollNo);
};
inline void Student::setRollNo(int aRollNo){
//keyword place in the start of signature
rollNo = aRollNo;
}
```

---

**OR**

```
void inline Student::setRollNo(int aRollNo){
//keyword place after function return type
rollNo = aRollNo;
}
```

**Another example of defining an inline function where the keyword is place both in declaration and definition:**

---

```
class Student{
private:
...
public:
inline void setRollNo(int aRollNo);
};
inline void Student::setRollNo(int aRollNo){
rollNo = aRollNo;
}
```



# Advantages of Inline Functions

---

- 1) Function call overhead doesn't occur
- 2) It also saves overhead of a return call from a function
- 3) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return

# Disadvantages of Inline Functions

---

- 1) The added variables from the inline function consumes additional registers
- 2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code
- 3) 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
- 4) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed

# Inline friend definitions

---

Friend functions can be defined (given a function body) inside class declarations. These functions are inline functions, and like member inline functions they behave as though they were defined immediately after all class members have been seen but before the class scope is closed (the end of the class declaration). Friend functions that are defined inside class declarations are in the scope of the enclosing class.

**Remember** friends are not member functions you cannot access it through the object of class.

# Inline friend definitions

---

```
class Box {
 double width;

 public:
 friend void printWidth(Box box){
 cout << "Width of box : " << box.width << endl;}
 void setWidth(double wid){
 width = wid;}};

int main() {
 Box box;

 box.setWidth(10.0);

 printWidth(box);

 return 0;}
```

# Inline friend definitions

---

```
Width of box : 10

Process exited after 0.2927 seconds with return value 0
Press any key to continue . . .
```

# Addition of members of two different classes using friend Function (Multiple friends)

---

```
#include <iostream>

using namespace std;

class B;

class A {
 private:
 int numA;
 public:
 A(): numA(12) { }
 friend int add(A, B);};
```

# Multiple friends

---

```
class B {
 private:
 int numB;

 public:
 B(): numB(1) { }
 friend int add(A , B);};
int add(A objectA, B objectB){
 return (objectA.numA + objectB.numB);}
```

# Multiple friends

---

```
int main()
{
 A objectA;
 B objectB;
 cout<<"Sum: "<< add(objectA, objectB);
 return 0;
}
```



# Multiple friends

---

```
Sum: 13

Process exited after 0.01784 seconds with return value 13
Press any key to continue . . .
```

# Function Class

---

- A **friend class** is a class that can access the private and protected members of a class in which it is declared as **friend**. This is needed when we want to allow a particular class to access the private and protected members of a class.
- For example: we have two classes XYZ and ABC. The XYZ class has two private data members ch and num, this class declares ABC as friend class. This means that ABC can access the private members of XYZ,

```
friend class ABC;
```

# Syntax

---

```
... ..
class B;
class A
{
 // class B is a friend class of class A
 friend class B;

}
class B
{

}
```

# Friend Classes:

---

- ✓ When a class is made a friend class, all the member functions of that class becomes friend functions.
- ✓ In this program, all member functions of class B will be friend functions of class A.
- ✓ Thus, any member function of class B can access the private and protected data of class A.
- ✓ But, member functions of class A cannot access the data of class B.

# Function Class

---

- `#include <iostream>`
- `using namespace std;`
- `class XYZ {`
- `private:`
- `char ch='A';`
- `int num = 11;`
- `public:`
- `friend class ABC;`
- `};`
- `class ABC {`
- `public:`
- `void disp(XYZ obj){`
- `cout<<obj.ch<<endl;`
- `cout<<obj.num<<endl;`
- `}`
- `};`
- `int main() {`
- `ABC obj;`
- `XYZ obj2;`
- `obj.disp(obj2);`
- `return 0;`
- `}`

# Example code

---

```
#include <iostream>
using namespace std;
class YourClass {
friend class YourOtherClass; // Declare a friend class
public:
 YourClass() : topSecret(0){}
 void printMember() { cout << topSecret << endl; }
private:
 int topSecret;};
class YourOtherClass {
public:
 void change(YourClass& yc, int x){
 yc.topSecret = x;}};
```

# Example code

---

```
int main() {
 YourClass c1;
 YourOtherClass oc1;
 c1.printMember();
 oc1.change(c1, 5);
 c1.printMember();}
```

# Example code

---

```
0
5

Process exited after 0.1587 seconds with return value 0
Press any key to continue . . .
```



# Friendship is not **mutual**/symmetric

---

- ✓ Friendship is granted, not taken—i.e.,
- ✓ For class B to be a friend of class A, class A must explicitly declare that class B is its friend.
- ✓ Friendship is not mutual unless explicitly specified
- ✓ In the above example, member functions of YourClass cannot access the private members of YourOtherClass.

# Friendship is not transitive

---

- if class A is a friend of class B, and class B is a friend of class C, you cannot infer that class A is a friend of class C
- Friendship is not transitive, so classes that are friends of **YourOtherClass** cannot access **YourClass's** private members.

# Friendship is not inherited

---

- ✓ In C++, friendship is not inherited. If a base class has a friend function, then the function doesn't become a friend of the derived class(es).
- ✓ Friendship is not inherited, meaning that classes derived from **YourOtherClass** cannot access **YourClass**'s private members.

# *Overloaded friend Functions*

---

- ✓ It's possible to specify overloaded functions as friends of a class.
- ✓ Each function intended to be a friend must be explicitly declared in the class definition as a friend of the class.

# *Overloaded friend Functions*

---

```
class Distance {
 int meter;
 friend int addFive(Distance);
 friend int addFive(Distance , int);
public:
 Distance() : meter(0) {};
int addFive(Distance d) {
 d.meter += 5;
 return d.meter;}
int addFive(Distance d , int i) {
 d.meter += i;
 return d.meter;}
}
```

# *Overloaded friend Functions*

---

```
int main() {
 Distance D;
 cout << "Distance: " << addFive(D) << endl;;
 cout << "Distance overloaded: " << addFive(D,6);
 return 0;}
```

# *Overloaded friend Functions*

---

```
Distance: 5
Distance overloaded: 6

Process exited after 0.02007 seconds with return value 0
Press any key to continue . . .
```