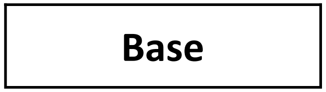
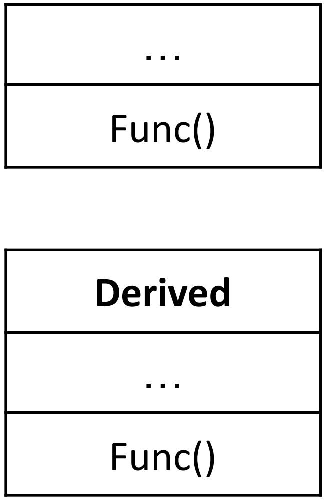
Lecture 16

Function Overriding

Overriding member functions of base class

- Sometimes a derived class needs to have the same function name as that in a base class but with different functionality.
- This is achieved by function overriding.
- Overriding a function is simple: just reimplement the function with the same name and arguments in the derived class.

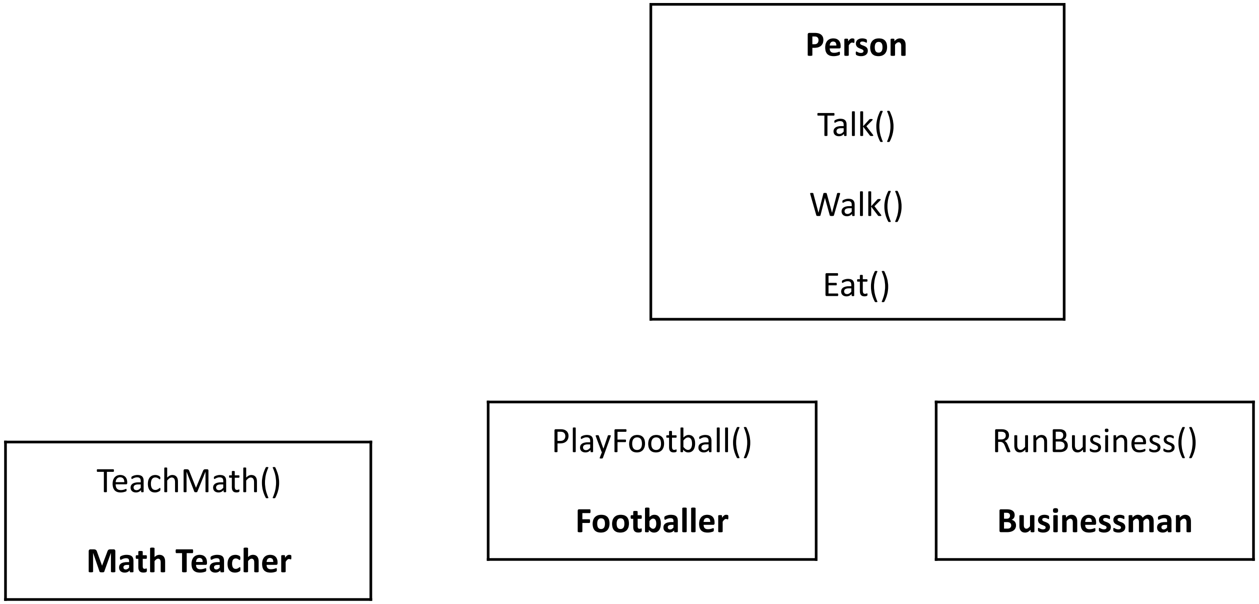




Overriding example

- Derive class can override member function of base class such that the working of function is similar to former implementation

```
class Person {
public:
        void Walk();
};
  class ParalyzedPerson : public Person {
public:
        void Walk();
};
```



Person

Talk()

Walk()

Eat()

TeachMath()

Math Teacher

PlayFootball()

Footballer

RunBusiness()

Businessman

Calling inherited functions and overriding(redefining) behavior

- By default, derived classes inherit all of the behaviors defined in a base class.
- When a member function is called with a derived class object, the compiler

first looks to see if that member exists in the derived class.

- If not, it begins walking up the inheritance chain and checking whether the member has been defined in any of the parent classes. It uses the first one it finds.

Calling inherited functions and overriding behavior

Derived class inherits member function identity() from base class and use it.

```
class Base
{
public:
Base(int value)
: m_value(value)
{
}
void identify() { std::cout << "I am a Base\n"; } int m_value;

};
```

```
class Derived: public Base
{
public:
Derived(int value)
: Base(value)
{
}
```

Calling inherited functions and overriding behavior

```
}
};
int main()
{
Base base(5);
base.identify();

Derived derived(7); derived.identify();

return 0;
```

This prints

```
I am a Base
I am a Base
```

- When derived.identify() is called, the compiler looks to see if function identify() has been defined in the Derived class. It hasn’t.

- Then it starts looking in the inherited classes (which in this case is Base). Base has defined an identify() function, so it uses that one.

- In other words, Base::identify() was used because Derived::identify() doesn't exist.

Redefining (overriding) behavior

- However, if we had defined Derived::identify() in the Derived class, it would have been used instead.
- This means that we can make functions work differently with our derived classes by **redefining (overriding)** them in the derived class!
- Below Derived class redefines the identity() member function.

```
class Derived: public Base
 {
public:
    Derived(int value)
        : Base(value)
     {
     }
   int getValue() { return m_value; }

     // Here's our modified function
void identify(){
cout << "I am a Derived";
 }
 };
```

```
int main()
{
    Base base(5);
    base.identify();

    Derived derived(7);
    derived.identify(); //overrides identity()

    return 0;
}
```

Output:

```
I am a Base
I am a Derived
```

Overriding a function: A simple Example

```
 class Base {
 public:
 void PrintNum() {
```

```
cout << 1 << endl ;
}
};
class Derived : public Base {
public:
// Override
void PrintNum() {
cout << 2 << endl ;
}
} ;
Base b;
b.PrintNum() ; // Prints 1
Derived d ;
d.PrintNum() ; // Prints 2
```

Adding to existing functionality

- Note that Derived::identify() completely hides Base::identify(), in the previous example
- Sometimes we don’t want to completely replace a base class function, but instead want to add additional functionality to it.
- It is possible to have our derived function call the base version of the function of the same name (in order to reuse code) and then add additional functionality to it.
- We redefine Derived::identify() so it first calls Base::identify() and then does its own additional stuff.
- To have a derived function call a base function of the same name, simply do a normal function call, but prefix the function with the scope

qualifier (the name of the base class and two colons).

Adding to existing functionality

- **Caution:** Calling function identify() without a scope resolution qualifier would default to the identify() in the current class, which would be Derived::identify().

- This would cause Derived::identify() to call itself, which would lead to an infinite loop!

```
class Derived: public Base
{
public:
Derived(int value)
: Base(value)
{
}
int getValue() { return
m_value; }

// Here's our modified function
```

```
void identify() {
Base::identify(); // call
Base::identify first cout << "I am a
Derived\n"; //then derived }
};
int main()
{
Base base(5);
base.identify();

Derived derived(7); derived.identify();
```

```
return 0;
}
```

Output:

```
I am a Base
I am a Base
I am a Derived
```

Overloading vs Overriding (Redefining)

- Do not mix overriding and overloading. These are two different concepts • Overloading:
  - Allow to use same name for member functions with different arguments
  - Overloading is done within the scope of one class
  - In below example, member function set() is overloaded.

- **Example:**

```
class A {
public:
void set (int a) { _a = a; }
void set (double f) {_f = f; }
void set (int a, double f) { _a = a;
_f = f; }  private:
double _f; int _a;
};
```

- Overriding (redefining):

- Allow to specialize the behavior of an existing method by providing a different implementation in the derived classes
- Redefining function is only possible with inheritance

Overloading vs overriding

- However, overriding show(int x) in the derived class hides the remaining methods with the same name of the base class.

```
class Base {
public:
 void show() { cout<<“base::show()”<<endl ; }
 void show(int x) { cout<<“base::show(int x)”<<endl ; }
};
class Derived: public Base {
public:
void show(int x) { cout<<“derived::show(int x)”<<endl ; }
};
```

```
int main() {
Derived d;
d.show(4); //OKAY: calls derived::show(int)
}
```

Overloading vs overriding

- However, overriding show(int x) in the derived class hides the remaining methods with the same name of the base class.

```
class Base {
public:
 void show() { cout<<“base::show()”<<endl ; }
 void show(int x) { cout<<“base::show(int x)”<<endl ; }
};
class Derived: public Base {
public:
void show(int x) { cout<<“derived::show(int x)”<<endl ; }
};
```

```
int main() {
Derived d;
d.show(); //error: no matching function for call to Derived::show()’ }
```

Overloading vs overriding

- We can introduce the base class functions with the same name using `using::`

```
class Base {
public:
 void show() { cout<<“base::show()”<<endl ; }
 void show(int x) { cout<<“base::show(int x)”<<endl ; }
};
class Derived: public Base {
public:
 using Base::show; //introducing base class show() methods
 void show(int x) { cout<<“derived::show(int x)”<<endl ; }
 }
};
```

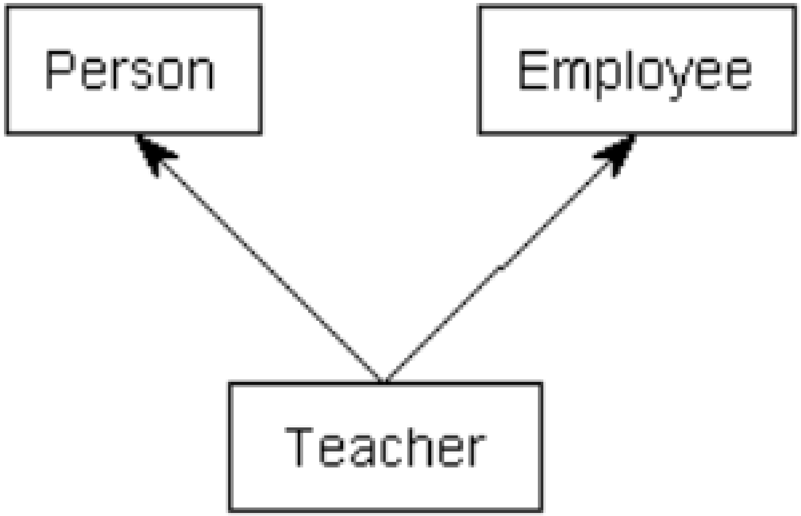
```
int main() {
Derived d;
d.show(4); // calls derived::show(int)
d.show(); // calls base::show()

}
```

Multiple Inheritance

- Multiple inheritance enables a derived class to inherit members from more than one base classes.
- Let's say we want to write a program to keep track of a bunch of **teachers**. • A teacher is a **person**.
- However, a teacher is also an **employee**.
- Multiple inheritance can be used to create a **Teacher class** that inherits properties from both **Person and Employee**.
- To use multiple inheritance, simply specify each base class (just like in single inheritance), separated by a comma. For example:

```
class Teacher: public Person, public Employee
```



Multiple Inheritance Example: Teacher class

```
class Person {
private:
•
   string m_name;
   int m_age;
 public:
   Person(string name, int age)
      : m_name(name), m_age(age)
```

```
      {
      }
   string getName() { return m_name;
   } int getAge() { return m_age; }
   };

class Teacher: public Person, public Employee
{ private:
```

```
class Employee {
private:
string m_employer;
double m_wage;
public:
```

```
  Employee(string employer, double wage) :
  m_employer(employer), m_wage(wage) {
  }
string getEmployer() { return m_employer;
} double getWage() { return m_wage; } };
```

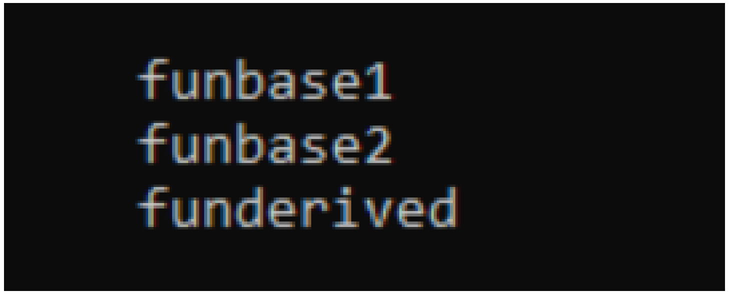
```
   int m_teachesGrade; //Data member specific to Teacher class
public:
Teacher(string name, int age, string employer, double wage, int teachesGrade) :
   Person(name, age), Employee(employer, wage), m_teachesGrade(teachesGrade) {
   }
};
```

Another Simple Example of Multiple Inheritance

```
//simple example showing multiple inheritance
class base1 {
public:
          void funbase1(void) { cout<<"funbase1"; }
};
```

```
class base2 {
public:
        void funbase2(void) { cout<<"funbase2"; }
};

class derived:public base1, public base2 {
public:
        void funderived(void) { cout<<"funderived";}
};
void main(void) {
derived der; //der inherits functionalities from both base classes
        der.funbase1();
        der.funbase2();
        der.funderived();
}
```



```
funbase1
funbase2
funderived
```

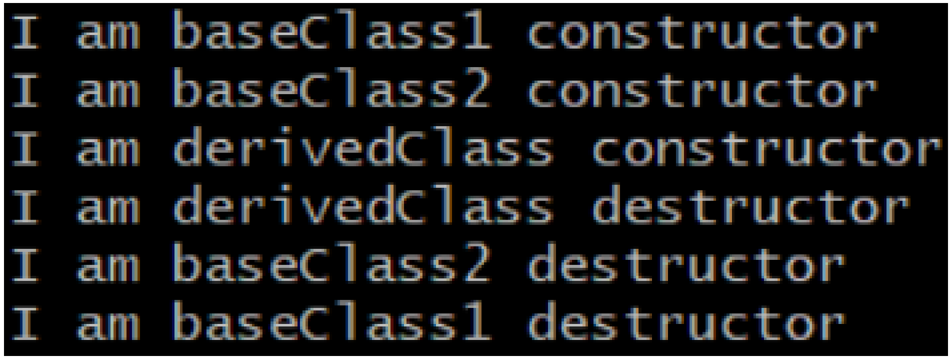
Constructor & Destructor in Multiple inheritance

```
class baseClass1 {
```

```
public:
 baseClass1() {
 cout<<"I am baseClass1 constructor"<<endl;  }

~baseClass1() {
 cout<<"I am baseClass1 destructor"<<endl;  }
};
class baseClass2 {
 public:
 baseClass2() {
 cout<<"I am baseClass2 constructor"<<endl;  }
~baseClass2() {
 cout<<"I am baseClass2 destructor"<<endl;  }
 };
class derivedClass: public baseClass1, public  baseClass2 {
 public:
 derivedClass() {
 cout<<"I am derivedClass
constructor"<<endl;
 }
 ~derivedClass() {'
 cout<<"I am derivedClass destructor"<<endl;  }
};

int main() {
derivedClass D;
return 0;
```



```
I am baseClass1 constructor
I am baseClass2 constructor
I am derivedClass constructor
I am derivedClass destructor
I am baseClass2 destructor
I am baseClass1 destructor
```

```
}
```

Inheritance Ambiguity in C++ In multiple inheritances

- Multiple inheritance introduces a lot of issues that can increase the complexity of programs and make maintenance difficult.
- When one class is derived from two or more base classes then there may be a possibility that the base classes have functions with the same name.

Derived class

Base2 Class

void func()

Base1 Class

void func()

Problems with Multiple Inheritance

- Let’s take a look at some of these situations.
- First, ambiguity can result when multiple base classes contain a function with the **same name**.
- Assume that we are creating a class WirelessAdapter. This class contains functionality of a networking device so it is derived from the NetworkDevice class
- It also contains some functionalities of a USB so it is also derived from USBDevice class.
- It means WirelessAdapter is derived from two base classes, i.e., NetworkDevice and

USBDevice.

WirelessAdapter

USBDevice

NetworkDevice int getID()

int getID()

Example: Problems with Multiple Inheritance

- The below code will not compile. Because the netGear object contains two copies of getID() function.
-

```
class USBDevice
{
private:
int m_id;
public:
   USBDevice(long id)
      : m_id(id)
   {
```

```
    }
int getID() { return m_id;
} };
```

```
class
NetworkDevice {
private:
int m_id;
public:
  NetworkDevice(long
    id) : m_id(id)
{
}
```

```
int getID() { return m_id;
} };
```

```
class WirelessAdapter: public USBDevice, public
NetworkDevice { //inherits two copies of getID().
public:
WirelessAdapter(int usbId, int networkId) :
USBDevice(usbId),
NetworkDevice(networkId) {
}
};
```

```
int main() {
WirelessAdapter netGear(54, 18); cout <<
netGear.getID();
// Which getID() do we call? }
```

Problems with Multiple Inheritance

- When netGear.getID() is compiled, the compiler looks to see if WirelessAdapter contains a function named getID(). It doesn’t.
- The compiler then looks to see if any of the base classes have a function named getID().
- The problem is that netGear actually contains two getID() functions: one

inherited from USBDevice, and one inherited from NetworkDevice.

- Consequently, this function call is ambiguous, and we receive a compiler error

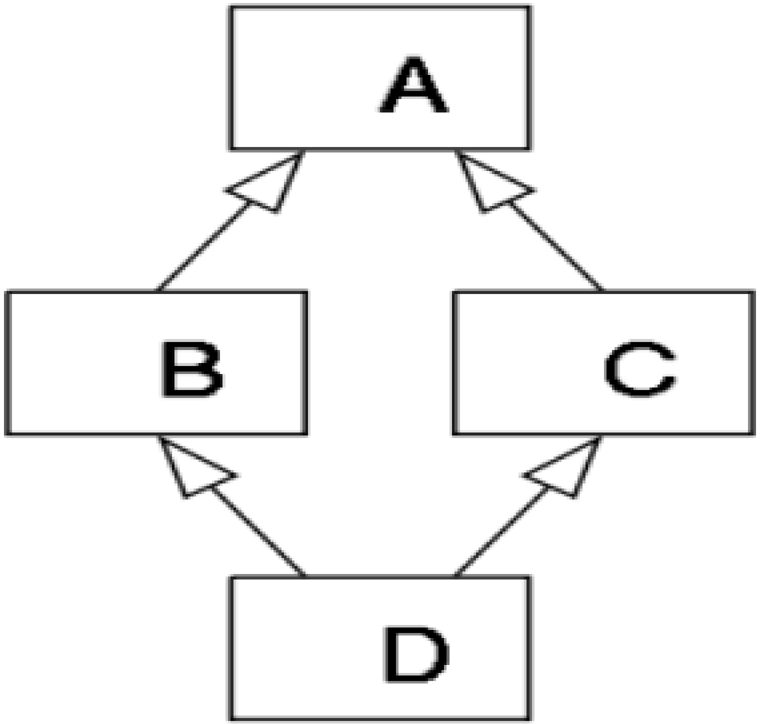
- **Solution:**
- To work around, we can explicitly specify which version we meant to call by using scope resolution operator. For example

```
cout << netGear.NetworkDevice::getID(); //displays 18
cout << netGear.USBDevice::getID(); //displays 54
```

Diamond Problem

- Another problem that arises from multiple inheritance is the **Diamond**

Problem. • if two classes B and C inherit from a class A, and a class D inherits from both B and C, then the derived class D will contain two copies of A's member variables: one via B, and one via C.



Memory View of multiple inheritance

- The derived class D will contain two copies of A's member variables: one via B, and one via C.

Class B

Members of Class

Members of Class

A Members of

A Members of

Class C

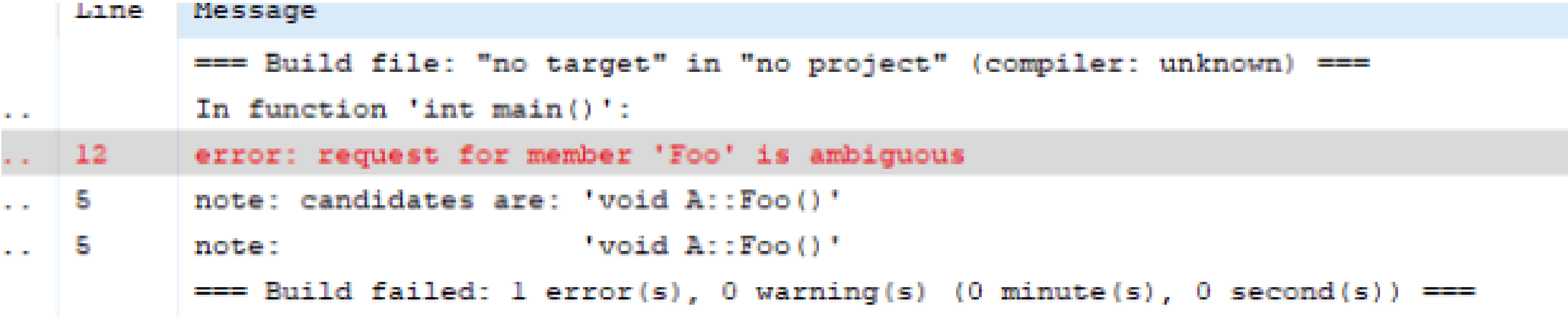
Members of Class D

Diamond Problem: Example

```
class A { public: void Foo() {} }
class B : public A {}
class C : public A {}
class D : public B, public C {}

int main() {
D d;
d.Foo();
}
```

is this B's Foo() or C's Foo() ??



```
    Line  Message
          === Build file: "no target" in "no project" (compiler: unknown) ===
..        In function 'int main()':
..  12    error: request for member 'Foo' is ambiguous
..  5     note: candidates are: 'void A::Foo()'
..  5     note:                 'void A::Foo()'
          === Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Solution: Virtual Inheritance

- **Virtual inheritance** is a C++ technique that ensures only one copy of a base class's member variables are inherited by grandchild derived classes.
- Its objective is to allow efficient use of memory and elimination of duplicate state spaces when designing inheritance hierarchies that share a common base class.

- if classes B and C inherit virtually from class A, then objects of class D will contain only one set of the ***member variables (functions?)*** from class A.

Solution (virtual inheritance)

```
class A { public: void Foo() {} }
class B : public virtual A {}
class C : public virtual A {}
class D : public B, public C {}
```

class B: virtual A means, that any class inherited from B is now responsible for creating A by itself, since B isn't going to do it automatically.

D d;

d.Foo(); // no longer ambiguous

Memory View after using Virtual

Inheritance Members of Class A

Members of Class B

Members of Class C

Members of Class D

Class exercise

- Every object of class checking consists of two data members and two member functions from the base class account:
- Similarly, every object of class savings also consists of two data members and two member functions from the base class account: • Since an object of class ibc inherits from both checking and savings, there are two implicit objects of class account contained in an ibc object.

ibc object

```
account
```

```
account::get_name()
account::get_balance()
name[]
balance
```

```
checking savings ibc
```

```
savings::get_interest()
interest
```

```
ibc::get_minimum()
minimum
```

Base class object copy in class
checking

Base class object copy in class
savings

```
account::get_name()
account::get_balance()
name[]
balance
```

```
checking::get_charges()
charges
```