Programming in Modern C++

Module M35: Multiple Inheritance

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Understood casting at run-time
- Studied `dynamic_cast` with examples
- Understood RTTI and `typeid` operator

Module Objectives

- Understand Multiple Inheritance in C++

Module Outline

1. Multiple Inheritance in C++
   - Semantics
   - Data Members and Object Layout
   - Member Functions – Overrides and Overloads
   - Access Members of Base: `protected` Access
   - Constructor & Destructor
   - Object Lifetime
2. Diamond Problem
   - Exercise
3. Design Choice
4. Module Summary

Multiple Inheritance in C++

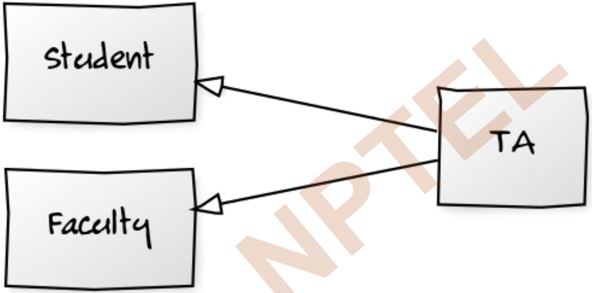
Multiple Inheritance in C++

Source:

- Is inheritance bad practice in OOP?, quora, 2019

Multiple Inheritance in C++: Hierarchy

- TA **ISA** Student; TA **ISA** Faculty

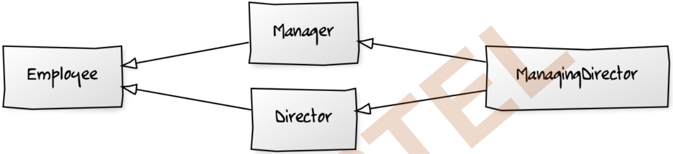


```
class Student;                                       // Base Class = Student
class Faculty;                                       // Base Class = Faculty
class TA: public Student, public Faculty; // Derived Class = TA
```

- TA inherits properties and operations of both Student as well as Faculty

Multiple Inheritance in C++: Hierarchy

- Manager **ISA** Employee, Director **ISA** Employee, ManagingDirector **ISA** Manager, ManagingDirector **ISA** Director

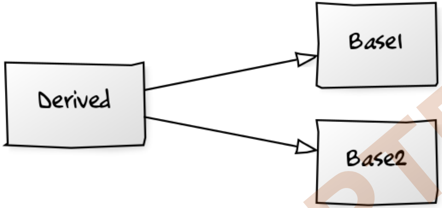


```
class Employee;                                          // Base Class = Employee -- Root
class Manager: public Employee;                          // Derived Class = Manager
class Director: public Employee;                         // Derived Class = Director
class ManagingDirector: public Manager, public Director; // Derived Class = ManagingDirector
```

- Manager inherits properties and operations of Employee
- Director inherits properties and operations of Employee
- ManagingDirector inherits properties and operations of both Manager as well as Director
- ManagingDirector, by transitivity, inherits properties and operations of Employee
- **Multiple inheritance hierarchy usually has a common base class**
- This is known as the **Diamond Hierarchy**

Multiple Inheritance in C++: Semantics

- Derived **ISA** Base1, Derived **ISA** Base2



```
class Base1;                                   // Base Class = Base1
class Base2;                                   // Base Class = Base2
class Derived: public Base1, public Base2;     // Derived Class = Derived
```

- Use keyword public after class name to denote inheritance
- Name of the Base class follow the keyword
- There may be more than two base classes
- public and private inheritance may be mixed

Multiple Inheritance in C++: Semantics

- Data Members
  - Derived class *inherits all* data members of *all* Base classes
  - Derived class may *add* data members of its own
- Member Functions
  - Derived class *inherits all* member functions of *all* Base classes
  - Derived class may *override* a member function of *any* Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of *any* Base class by *redefining* it with the *same name*; but *different signature*
- Access Specification
  - Derived class *cannot access* private members of *any* Base class
  - Derived class *can access* protected members of *any* Base class
- Construction-Destruction
  - A *constructor* of the Derived class *must first* call *all constructor*s of the Base classes to construct the Base class instances of the Derived class – Base class *constructor*s are called in *listing order*
  - The *destructor* of the Derived class *must* call the *destructor*s of the Base classes to destruct the Base class instances of the Derived class

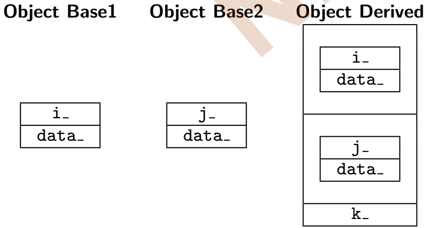
Multiple Inheritance in C++: Data Members and Object Layout

- Data Members
  - Derived class *inherits* *all* data members of *all* Base classes
  - Derived class may *add* data members of its own
- Object Layout
  - Derived class *layout* contains instances of *each* Base class
  - Further, Derived class layout will have data members of its own
  - C++ does not guarantee the relative position of the Base class instances and Derived class members

Multiple Inheritance in C++: Data Members and Object Layout

```
class Base1 { protected:
    int i_, data_;
public: // ...
};
class Base2 { protected:
    int j_, data_;
public: // ...
};
class Derived: public Base1, public Base2 { // Multiple inheritance
    int k_;
public: // ...
};
```

Object Layout



- Object Derived has two data_ members!
- Ambiguity to be resolved with base class name: Base1::data_ & Base2::data_

Multiple Inheritance in C++: Member Functions – Overrides and Overloads

- Derived **ISA** Base1, Base2
- Member Functions
  - Derived class *inherits all* member functions of *all* Base classes
  - Derived class may *override* a member function of *any* Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of *any* Base class by *redefining* it with the *same name*; but *different signature*
- Static Member Functions
  - Derived class *does not inherit* the static member functions of *any* Base class
- Friend Functions
  - Derived class *does not inherit* the friend functions of *any* Base class

Multiple Inheritance in C++: Member Functions – Overrides and Overloads

```
class Base1 { protected: int i_, data_;
public: Base1(int a, int b): i_(a), data_(b) { }
    void f(int) { cout << "Base1::f(int) \n"; }
    void g() { cout << "Base1::g() \n"; }
};
class Base2 { protected: int j_, data_;
public: Base2(int a, int b): j_(a), data_(b) { }
    void h(int) { cout << "Base2::h(int) \n"; }
};
class Derived: public Base1, public Base2 { int k_;
public: Derived(int x, int y, int u, int v, int z): Base1(x, y), Base2(u, v), k_(z) { }
    void f(int) { cout << "Derived::f(int) \n"; }       // -- Overridden Base1::f(int)
    // -- Inherited Base1::g()
    void h(string) { cout << "Derived::h(string) \n"; } // -- Overloaded Base2:: h(int)
    void e(char) { cout << "Derived::e(char) \n"; }     // -- Added Derived::e(char)
};

Derived c(1, 2, 3, 4, 5);

c.f(5);      // Derived::f(int)   -- Overridden Base1::f(int)
c.g();       // Base1::g()         -- Inherited Base1::g()
c.h("ppd");  // Derived::h(string) -- Overloaded Base2:: h(int)
c.e(’a’);    // Derived::e(char)   -- Added Derived::e(char)
```

Inheritance in C++: Member Functions – using for Name Resolution

Ambiguous Calls

```
class Base1 { public:
    Base1(int a, int b);
    void f(int) { cout << "Base1::f(int) "; }
    void g() { cout << "Base1::g() "; }
};
class Base2 { public:
    Base2(int a, int b);
    void f(int) { cout << "Base2::f(int) "; }
    void g(int) { cout << "Base2::g(int) "; }
};
class Derived: public Base1, public Base2 {
public: Derived(int x, int y, int u, int v, int z);


};
Derived c(1, 2, 3, 4, 5);

c.f(5); // Base1::f(int) or Base2::f(int)?
c.g(5); // Base1::g() or Base2::g(int)?
c.f(3); // Base1::f(int) or Base2::f(int)?
c.g();  // Base1::g() or Base2::g(int)?
```

Unambiguous Calls

```
class Base1 { public:
    Base1(int a, int b);
    void f(int) { cout << "Base1::f(int) "; }
    void g() { cout << "Base1::g() "; }
};
class Base2 { public:
    Base2(int a, int b);
    void f(int) { cout << "Base2::f(int) "; }
    void g(int) { cout << "Base2::g(int) "; }
};
class Derived: public Base1, public Base2 {
public: Derived(int x, int y, int u, int v, int z);
    using Base1::f; // Hides Base2::f
    using Base2::g; // Hides Base1::g
};
Derived c(1, 2, 3, 4, 5);

c.f(5);          // Base1::f(int)
c.g(5);          // Base2::g(int)
c.Base2::f(3);   // Base2::f(int)
c.Base1::g();    // Base1::g()
```

- Overload resolution does not work between Base1::g() and Base2::g(int)
- using hides other candidates; Explicit use of base class name can resolve (weak solution)

Multiple Inheritance in C++: Access Members of Base: `protected` Access

- Access Specification
  - Derived class *cannot access* private members of *any* Base class
  - Derived class *can access* protected members of *any* Base class

Multiple Inheritance in C++: Constructor & Destructor

- Constructor-Destructor
  - Derived class *inherits all* Constructors and Destructor of Base classes (but in a different semantics)
  - Derived class *cannot overload* a Constructor or *cannot override* the Destructor of *any* Base class
- Construction-Destruction
  - A *constructor* of the Derived class *must first* call *all constructors* of the Base classes to construct the Base class instances of the Derived class
  - Base class *constructors* are called in *listing order*
  - The *destructor* of the Derived class *must* call the *destructors* of the Base classes to destruct the Base class instances of the Derived class

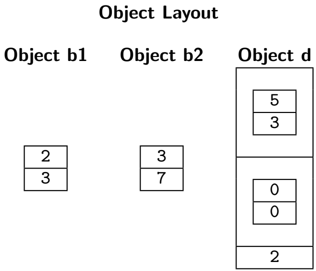
Multiple Inheritance in C++: Constructor & Destructor

```
class Base1 { protected: int i_; int data_;
public: Base1(int a, int b): i_(a), data_(b) { cout << "Base1::Base1() "; }
  ~Base1() { cout << "Base1::~Base1() "; }
};

class Base2 { protected: int j_; int data_;
public: Base2(int a = 0, int b = 0): j_(a), data_(b) { cout << "Base2::Base2() "; }
  ~Base2() { cout << "Base2::~Base2() "; }
};

class Derived: public Base1, public Base2 { int k_;
public: Derived(int x, int y, int z):
      Base1(x, y), k_(z) { cout << "Derived::Derived() "; }
      // Base1::Base1 explicit, Base2::Base2 default
  ~Derived() { cout << "Derived::~Derived() "; }
};

Base1 b1(2, 3);
Base2 b2(3, 7);
Derived d(5, 3, 2);
```



Multiple Inheritance in C++: Object Lifetime

```
class Base1 { protected: int i_; int data_;
public: Base1(int a, int b): i_(a), data_(b)
        { cout << "Base1::Base1() " << i_ << ' ' << data_ << endl; }
    ~Base1() { cout << "Base1::~Base1() " << i_ << ' ' << data_ << endl; }
};
class Base2 { protected: int j_; int data_;
public: Base2(int a = 0, int b = 0): j_(a), data_(b)
        { cout << "Base2::Base2() " << j_ << ' ' << data_ << endl; }
    ~Base2() { cout << "Base2::~Base2() " << j_ << ' ' << data_ << endl; }
};
class Derived: public Base1, public Base2 { int k_; public:
    Derived(int x, int y, int z): Base1(x, y), k_(z)
        { cout << "Derived::Derived() " << k_ << endl; }
        // Base1::Base1 explicit, Base2::Base2 default
    ~Derived() { cout << "Derived::~Derived() " << k_ << endl; }
};

Derived d(5, 3, 2);
```

Construction O/P

```
Base1::Base1(): 5, 3  // Obj. d.Base1
Base2::Base2(): 0, 0  // Obj. d.Base2
Derived::Derived(): 2 // Obj. d
```

Destruction O/P

```
Derived::~Derived(): 2 // Obj. d
Base2::~Base2(): 0, 0  // Obj. d.Base2
Base1::~Base1(): 5, 3  // Obj. d.Base1
```

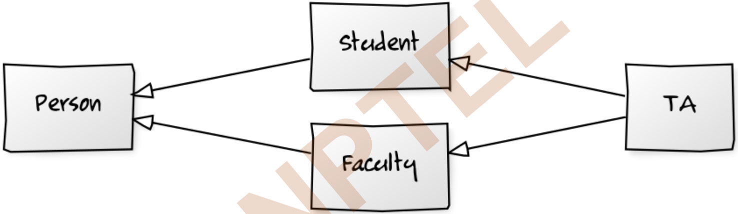
- First construct base class objects, then derived class object
- First destruct derived class object, then base class objects

Diamond Problem

Diamond Problem

Multiple Inheritance in C++: Diamond Problem

- Student **ISA** Person
- Faculty **ISA** Person
- TA **ISA** Student; TA **ISA** Faculty



```
class Person;                               // Base Class = Person -- Root
class Student: public Person;               // Base / Derived Class = Student
class Faculty: public Person;               // Base / Derived Class = Faculty
class TA: public Student, public Faculty; // Derived Class = TA
```

- Student inherits properties and operations of Person
- Faculty inherits properties and operations of Person
- TA inherits properties and operations of both Student as well as Faculty
- TA, by transitivity, inherits properties and operations of Person

Multiple Inheritance in C++: Diamond Problem

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

class Person { // data members of person
    public: Person(int x)  { cout << "Person::Person(int)" << endl; }
};
class Faculty: public Person { // data members of Faculty
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};
class Student: public Person { // data members of Student
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
};
class TA: public Faculty, public Student {
    public: TA(int x): Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
};
int main() { TA ta(30);
}
```

```
Person::Person(int)
Faculty::Faculty(int)
Person::Person(int)
Student::Student(int)
TA::TA(int)
```

- **Two instances of base class object (Person) in a TA object!**

Multiple Inheritance in C++: virtual Inheritance – virtual Base Class

```
#include<iostream>
using namespace std;
class Person { // data members of person
    public: Person(int x) { cout << "Person::Person(int)" << endl; }
    Person() { cout << "Person::Person()" << endl; } // Default ctor for virtual inheritance
};
class Faculty: virtual public Person { // data members of Faculty
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};
class Student: virtual public Person { // data members of Student
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
};
class TA: public Faculty, public Student {
    public: TA(int x): Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
};
int main()  { TA ta(30); }
```

```
Person::Person()
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int)
```

- Introduce a default constructor for root base class Person
- Prefix every inheritance of Person with virtual
- **Only one instance of base class object (Person) in a TA object!**

Multiple Inheritance in C++: virtual Inheritance with Parameterized Ctor

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

class Person {
    public: Person(int x) { cout << "Person::Person(int)" << endl; }
    Person() { cout << "Person::Person()" << endl; }
};
class Faculty: virtual public Person {
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};
class Student: virtual public Person {
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
};
class TA: public Faculty, public Student {
    public: TA(int x): Student(x), Faculty(x), Person(x) { cout << "TA::TA(int)" << endl; }
};
int main() { TA ta(30); }
```

```
Person::Person(int)
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int )
```

- **Call parameterized constructor of root base class Person from constructor of TA class**

Multiple Inheritance in C++: Ambiguity

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

class Person {
    public: Person(int x) { cout << "Person::Person(int)" << endl; }
    Person() { cout << "Person::Person()" << endl; }
    virtual ~Person();
    virtual void teach() = 0;
};
class Faculty: virtual public Person {
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
    virtual void teach();
};
class Student: virtual public Person {
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
    virtual void teach();
};
class TA: public Faculty, public Student {
    public: TA(int x):Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
    virtual void teach();
};
```

- **In the absence of `TA::teach()`, which of `Student::teach()` or `Faculty::teach()` should be inherited?**

Multiple Inheritance in C++: Exercise

```
class A {
public:
    virtual ~A() { cout << "A::~A()" << endl; }
    virtual void foo() { cout << "A::foo()" << endl; }
};
class B: public virtual A {
public:
    virtual ~B() { cout << "B::~B()" << endl; }
    virtual void foo() { cout << "B::foo()" << endl; }
};
class C: public virtual A {
public:
    virtual ~C() { cout << "C::~C()" << endl; }
    virtual void foobar() { cout << "C::foobar()" << endl; }
};
class D: public B, public C {
public:
    virtual ~D() { cout << "D::~D()" << endl; }
    virtual void foo() { cout << "D::foo()" << endl; }
    virtual void foobar() { cout << "D::foobar()" << endl; }
};
```

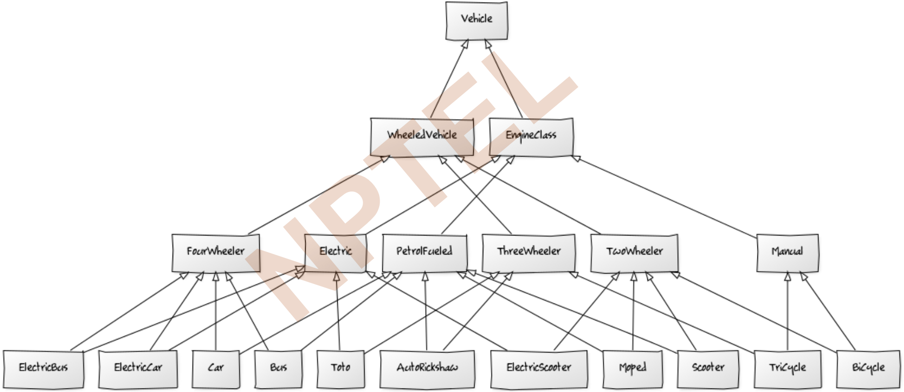
- **Consider the effect of calling `foo` and `foobar` for various objects and various pointers**

Design Choice

Design Choice

Design Choice: Inheritance or Composition

- Vehicle Hierarchy



Vehicle

WheeledVehicle

EngineClass

FourWheeler

Electric

PetrolFueled

ThreeWheeler

TwoWheeler

Manual

ElectricBus

ElectricCar

Car

Bus

Toto

AutoRickshaw

ElectricScooter

Moped

Scooter

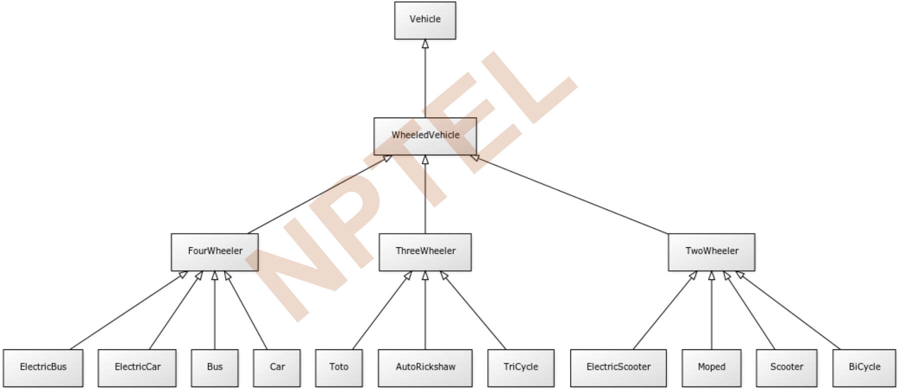
TriCycle

BiCycle

- Wheeled Hierarchy and Engine Hierarchy interact
- Large number of cross links!
- Multiplicative options make modeling difficult

Design Choice: Inheritance or Composition

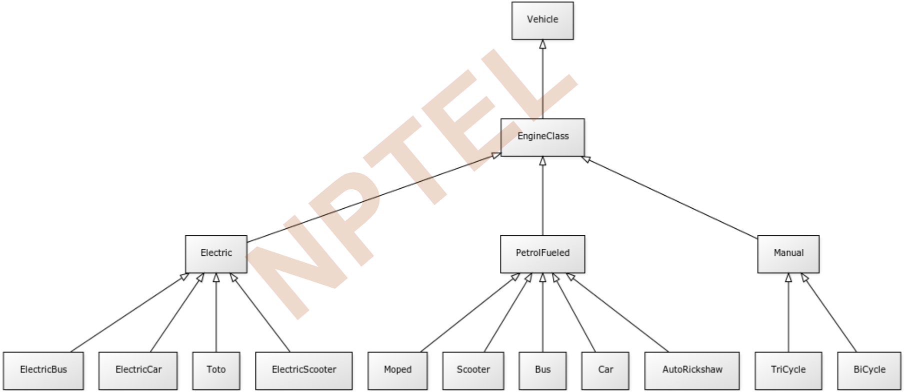
- Vehicle Hierarchy



- Wheeled Hierarchy use Engine as Component
- Linear options to simplify models
- Is this dominant?

Design Choice: Inheritance or Composition

- Vehicle Hierarchy



- Engine Hierarchy use Wheeled as Component
- Linear options to simplify models
- Is this dominant?

Module Summary

- Introduced the Semantics of Multiple Inheritance in C++
- Discussed the Diamond Problem and solution approaches
- Illustrated the design choice between inheritance and composition