Multiple & Virtual Inheritance in C++

Technion - Israel Institute of Technology

Last Update : Guy Suday, August 2018
Today's Main Topics

- Initialization of virtual bases
- Final classes in C++
- Multiple inheritance, virtual members and virtual inheritance.
- Construction order with multiple inheritance
Reminder: Multiple Inheritance

Car

Self-Driving Car

Robot
Reminder : Diamond Inheritance

Mammal

Animal

Winged Animal

Bat
Multiple Inheritance can cause *ambiguities* in methods binding.
- If a method is “inherited twice”.

An *ambiguate method call* can be either:

- Coincidental
- Inherent
Ambiguities

- Given a class $D$ that inherits from bases $B_1, B_2$. If both $B_1$ and $B_2$ declare a method $f()$, and a $D$ object calls $f()$, the call is *coincidental ambiguate*.

- If we specify **which method to bind** this ambiguity is solved.

- If we **rename** $B_1::f()$ to $B_1::g()$, this ambiguity is solved.
Ambiguities

- Given a class $D$ that inherits from bases $B_1, B_2$. If both $B_1$ and $B_2$ declares a method $f()$, and a $D$ object calls $f()$, the call is *coincidental ambiguity*.

  ```
  D* dObj = new D();
  ```

- If we override $f()$ on D (“*final Overrider*”) this ambiguity is solved.

  ```
  struct D : B1, B2 { virtual void f() { ... } };
  ```

**Diagram:**

```
     f()
    /   \
   /     \
B1    B2
     \
    / \
   /   \
   D  
     \
     f()
```

$f()$
Ambiguities

Given a class $D$ that inherits from bases $B_1, B_2$ that inherits from $A$. If $A$ declares a method $f()$, and a $D$ object calls $f()$, the call is inherent ambiguous.

- If we specify **which method to bind** this ambiguity is solved.

- If we override $f()$ on $D$ (**final Overrider**) this ambiguity is solved.
Ambiguities

- Given a class $D$ that inherits from bases $B_1$, $B_2$ that inherits from $A$. If $A$ declares a method $f()$, and a $D$ object calls $f()$, the call is inherent ambiguity.

- This ambiguity is caused due to ambiguous memory layout.
• Given a class $D$ that inherits from bases $B_1$, $B_2$ that inherits from $A$. If $A$ declares a method $f()$, and a $D$ object calls $f()$, the call is inherent ambiguity.

• If we use virtual inheritance this ambiguity is solved.

```cpp
D* dObj = new D();
```

```cpp
dObj -> f();
```
Reminder: Virtual Inheritance

- Mammal
- Animal
- Bat
- Winged Animal

Diagram showing inheritance relationships.

11
First let’s draw a D object, What will be printed?
Construction of a virtual base class

• **Answer:** nothing will be printed. The compiler will issue an error message.

• **Work-around 1:** define a **parameter-less constructor** in V.

• **Work-around 2:** call V’s constructor **directly** from the constructor of D.
  - Virtual bases are always initialized by most derived class – other initializations are ignored. This also applies if the most derived class is **not an immediate derived class** of the virtual base.
  - Work-around 1 is actually the same solution – by giving V a default constructor, it will always be implicitly called by the most derived class.

• **Comments:**
  - All **virtual** inheritances of the **same** object are **unified**.
  - All **non-virtual** inheritances of the **same** object are **distinct**.
Virtual base initialization example

```cpp
struct V {
    V();
    V(int);
    ...
};
struct B1 : virtual V {
    B1();
    B1(int i): V(i) { /*...*/ }
    ...
};
struct B2 : virtual V {
    B2();
    B2(int i) { /*...*/ }
    ...
};
struct D : B1, B2 {
    D(int i): V(i) { /*...*/ }
};
```
Reminder: Dispatch Algorithm

Given the invocation p->f():

- **Upcast** p to B_f, the uppermost class **that defines** f –
  \[ p' = \text{static
cast}<B_f*>(p); \]

- **Find the address** of f in B_f’s vtbl
  entry contains most derived implementation of f or thunk

- **this adjustment**: downcast p’ to the most-derived class that overrides f – **Done by the thunk**

- **Invoke** f

```
p' = static_cast<B_f*>(p);
p = p';
B1
B2
D

&D::f
&B2::g
this = \Delta(B1);
goto D::f;
```
C++11 Explicit final

```cpp
struct Basel1 final {
    //...
};
// ill-formed because the class Basel1 has been
// marked final
struct Derived1 : Basel1 {
    //...
};

struct Base2 {
    virtual void f() final;
};

struct Derived2 : Base2 {
    // ill-formed because the virtual function
    // Base2::f has been marked final
    void f();
};
```

Final is only for virtual functions.
Prevents overriding, but not hiding!
struct A{
    virtual void Drive() {}}
};
struct B : A{
    void Drive() {
        this->WearSeatBelt();
    }
    virtual void WearSeatBelt() {
        cout << "Now we can drive!" << endl;
    }
};
struct C : B{
    void Drive() { cout << "Who needs safety?" << endl; }
    void WearSeatBelt() { cout << "OK fine..." << endl; }
};

A* b = new C();
b->Drive();
// Who needs safety?
struct A {
    virtual void Drive() {}
};
struct B : A {
    void Drive() final{
        this->WearSeatBelt();
    }
    virtual void WearSeatBelt() {
        cout << "Now we can drive!" << endl;
    }
};
struct C : B {
    void Drive() { cout << "Who needs safety?" << endl; }
    void WearSeatBelt() { cout << "OK fine..." << endl; }
};

A* b = new C();
b->Drive();
// OK fine...

Will not compile otherwise
Java 8 introduced us with default methods that provides default implementation within the interface.

```java
interface I1 { default void f() { ... } }
interface I2 { default void f() { ... } }
class X implements I1, I2 { ... }
```
This new ability can cause an **ambiguity**.

```java
interface I1 { default void f() { ... } }
interface I2 { default void f() { ... } }
class X implements I1, I2 { ... }
```

**Solutions:**
- A final Overrider
- Re-declaring abstract (by using the normal interface-method-declaration).
Construction and multiple inheritance

- **Elements to initialize**: sub-objects and fields.
- **Where to initialize?** Best is in the initialization list (after the constructor signature). Sub-objects can only be initialized in the initialization list.
  - Order of elements in the initialization list is unrelated to the order in which they will actually be invoked. This makes it possible to guarantee that the construction order is the exact opposite of destruction order.
- **Construction order** is a recursive algorithm:
  1. Virtual base classes, in the order they occur in depth-first, left-to-right (by definition order) traversal of the hierarchy graph.
     - If a virtual base class is derived from a non-virtual base, then this non-virtual base will be constructed before the virtual base.
  2. Remaining base classes, in the order they occur in the hierarchy graph.
  3. Fields (data members).
  4. Constructor body.
- **Order of destruction** is the same in reverse.
Apply topological sort ranking inheritance

- DAG in a depth-first, left-right scan
  - Virtual and non-virtual inheritance are treated alike.
- Construct all virtual base classes (immediate and non-immediate)
  - Use ranking order.
  - Do not construct twice.
  - Apply recursively to construct their non-virtual bases.
  - Construct non-virtual immediate base classes:
    - Use ranking order (same as definition order).
    - Apply recursively to construct their non-virtual bases.
- Construction order in example:
  - U1 U2 Y X V2 V1 V3 V4 B1 B2 D
- Destruction order in example:
  - D B2 B1 V4 V3 V1 V2 X Y U2 U1