Ambiguities

```java
class A { void f(int); }
class B { void f(int); }
class C : public A, public B {}

c= pc;
pc->f(42);
```

⚠️ Which method is called?

**Solution I: Explicit qualification**

```java
pc->A::f(42);
pc->B::f(42);
```

**Solution II: Automagical resolution**

Idea: The Compiler introduces a linear order on the nodes of the inheritance graph.

Linearization

**Principle 1: Inheritance Relation**

Defined by parent-child. Example:

\[ C(A, B) \implies C \rightarrow A \land C \rightarrow B \]

**Principle 2: Multiplicity Relation**

Defined by the succession of multiple parents. Example:

\[ C(A, B) \implies A \rightarrow B \]

In General:

1. Inheritance is a uniform mechanism, and its searches (→ total order) apply identically for all object fields or methods.
2. In the literature, we also find the set of constraints to create a linearization as Method Resolution Order.
3. Linearization is a best-effort approach at best.

MRO via DFS

**Leftmost Preorder**

Depth-First Search

```
A B C W
```
MRO via DFS

**Leftmost Preorder Depth-First Search**

\[ L[A] = ABWC \]

⚠️ Principle 1 *inheritance* is violated

Python: classical python objects (≤ 2.1) use LPDFS!

LPDFS with Duplicate Cancellation

\[ L[A] = ABWC \]

⚠️ Principle 1 *inheritance* is violated

Python: classical python objects (≤ 2.1) use LPDFS!

LPDFS with Duplicate Cancellation

\[ L[A] = ABCW \]

✅ Principle 1 *inheritance* is fixed

Python: new python objects (2.2) use LPDFS(DC)!

LPDFS with Duplicate Cancellation

\[ L[A] = ABCW \]

⚠️ Principle 2 *multiplicity* not fulfillable

However \( B \rightarrow C \implies W \rightarrow V \)?
MRO via Refined Postorder DFS

**Reverse Postorder Rightmost DFS**

\[ L[A] = ABFDCEGHW \]

✓ Linear extension of inheritance relation

**RPRDFS**

\[ L[A] = ABCDGEF \]

⚠️ But principle 2 *multiplicity* is violated!

**Refined RPRDFS**

\[ L[A] = ABCDEFG \]

✓ Refine graph with conflict edge & rerun RPRDFS!
MRO via Refined Postorder DFS

Extension Principle: Monotonicity
If $C_1 \rightarrow C_2$ in $C$'s linearization, then $C_1 \rightarrow C_2$ for every linearization of $C$'s children.

MRO via C3 Linearization

A linearization $L[C]$ of a class $C$. Classes $B_1, \ldots, B_n$ are superclasses to child class $C$, defined in the local precedence order $C(B_1 \ldots B_n)$. Then

$$L[C] = \begin{cases} L[B_1], \ldots, L[B_n], B_1 \cdots B_n & \text{if } C(B_1, \ldots, B_n) \\ L[Object] = Object & \text{else} \end{cases}$$

MRO via Refined Postorder DFS

Extension Principle: Monotonicity
If $C_1 \rightarrow C_2$ in $C$'s linearization, then $C_1 \rightarrow C_2$ for every linearization of $C$'s children.

MRO via C3 Linearization

$$\begin{align*}
L[G] &\quad C \\
L[F] &\quad L \quad E \\
L[E] &\quad L \quad D \\
L[D] &\quad L \quad B \\
L[B] &\quad L \quad C \\
L[C] &\quad L \quad A \\
L[A] &\quad L \quad F \\
\end{align*}$$
Linearization vs. explicit qualification

**Linearization**
- No switch/duplexer code necessary
- No explicit naming of qualifiers
- Unique super reference
- Reduces number of multi-dispatching conflicts

**Qualification**
- More flexible, fine-grained
- Linearization choices may be awkward or unexpected

Languages with automatic linearization exist
- **CLOS** Common Lisp Object System
- **Dylan**, **Python** and **Perl 6** with C3
- Prerequisite for: Mixins

“And what about dynamic dispatching in Multiple Inheritance?”

Virtual Tables for Multiple Inheritance

```java
class A {
    int a; virtual int f(int);
};
class B {
    int b; virtual int f(int);
    virtual int g(int);
};
class C : public A, public B {
    int c; int f(int);
};
... C c;
B* pb = &c;
pb->f(42);
```

Virtual Tables for Multiple Inheritance

```java
%class.C = type %class.A { [12 x 18], 132 }
%class.A = type { 132 (...)**, 132 }
%class.B = type { 132 (...)**, 132 }

\triangle B

\[\begin{array}{l}
\text{vpotr \ int a} \\
\text{vpotr \ int b} \\
\text{int c}
\end{array}\]

\[\begin{array}{l}
\text{C::f} \\
\text{B::g}
\end{array}\]

\text{B* pb = &c;}
pb->f(42);
```
Virtual Tables for Multiple Inheritance

class A {
  int a; virtual int f(int);
};
class B {
  int b; virtual int f(int);
  virtual int g(int);
};
class C : public A, public B {
  int c; virtual int f(int);
};
...
C = new C();
c->f(42);

Basic Virtual Tables (\(\sim\) C++-ABI)

A Basic Virtual Table
consists of different parts:
- offset to top of an enclosing object's memory representation
- typeinfo pointer to an RTTI object (not relevant for us)
- virtual function pointers for resolving virtual methods

- Virtual tables are composed when multiple inheritance is used
- The \(\text{vptr}\) fields in objects are pointers to their corresponding virtual-subtables
- Casting preserves the link between an object and its corresponding virtual-subtable
- clang -cc1 -fdump-vtable-layouts -emit-llvm code.cpp yields the vtables of a compilation unit

Casting Issues

class A { int a; };
class B { virtual int f(int); };
class C : public A, public B {
  int c; virtual int f(int);
};
B* b = new C();
b->f(42);

Thunks

Solution: thunks
...are trampoline methods, delegating the virtual method to its original implementation with an adapted this-reference

define i32 @f_i32_i32(%class.B* %this@i32 %i32 %i1) {
  %1 = bitcast %class.B* %this to i8 *
  %2 = getelementptr i8* %1, i64 -16
  ; sizeof(A)=16
  %3 = bitcast i8* %2 to %class*
  %4 = call i32 @f_i32_i32(%class.C* %3, i32 %i1)
  ret i32 %4
}

\(\sim\) B-in-C-vtable entry for f(int) is the thunk \(\_f\)
Casting Issues

class A { int a; };
class B { virtual int f(int);};
class C : public A, public B { 
    int c; int f(int);
};
C* c = new C();
c->f(42);

C::f

△ this-Pointer for C::f is expected to point to C

C::Bf

Casting Issues

class A { int a; };
class B { virtual int f(int);};
class C : public A, public B { 
    int c; int f(int);
};
B* b = new C();
b->f(42);

C::f

△ this-Pointer for C::f is expected to point to C

C::Bf

Casting Issues

class A { int a; };
class B { virtual int f(int);};
class C : public A, public B { 
    int c; int f(int);
};
C* c = new C();
c->f(42);

C::f

△ this-Pointer for C::f is expected to point to C

C::Bf

Casting Issues

class A { int a; };
class B { virtual int f(int);};
class C : public A, public B { 
    int c; int f(int);
};
B* b = new C();
b->f(42);

C::f

△ this-Pointer for C::f is expected to point to C

C::Bf

C::f
Common Bases – Duplicated Bases

Standard C++ multiple inheritance conceptually duplicates representations for common ancestors:

```
class L {
  int i; virtual void f(int);
};
class A : public L {
  int a; void f(int);
};
class B : public L {
  int b; void f(int);
};
class C : public A, public B {
  int c;
};
...
C c;
L* pl = &c;
pl->f(42);
C* pc = (C*)pl;
```

Common Bases – Shared Base Class

Optionally, C++ multiple inheritance enables a shared representation for common ancestors, creating the diamond pattern:

```
class W {
  int w; virtual void f(int);
  virtual void g(int);
  virtual void h(int);
};
class A : public virtual W {
  int a; void f(int);
};
class B : public virtual W {
  int b; void g(int);
};
class C : public A, public B {
  int c; void h(int);
};
...
C* pc;
pc->f(42);
(W*)pc->g(42);
((A*)pc)->h(42);
```

Duplicated Base Classes

```
class L {
  int i; virtual void f(int);
};
class A : public L {
  int a; void f(int);
};
class B : public L {
  int b; void f(int);
};
class C : public A, public B {
  int c;
};
```

Shared Base Class

```
class W {
  int w; virtual void f(int);
  virtual void g(int);
  virtual void h(int);
};
class A : public virtual W {
  int a; void f(int);
};
class B : public virtual W {
  int b; void g(int);
};
class C : public A, public B {
  int c; void h(int);
};
```
Dynamic Type Casts

```cpp
class A : public virtual W {
    ...
};
class B : public virtual W {
    ...
};
class C : public A, public B {
    ...
};
class D : public C, public B {
    ...
};
C* pc = (C*)pw; // Compile error

vs.
C* pc = dynamic_cast<C*>(pw);
```

Dynamic Type Casts

```cpp
class A : public virtual W {
    ...
};
class B : public virtual W {
    ...
};
class C : public A, public B {
    ...
};
class D : public C, public B {
    ...
};
C* pc = static_cast<C*>(pw); // Compile error

vs.
C* pc = dynamic_cast<C*>(pw);
```

Shared Base Class

```cpp
class W {
    int w; virtual void f(int);
    virtual void g(int);
    virtual void h(int);
};
class A : public virtual W {
    int a; void f(int);
};
class B : public virtual W {
    int b; void g(int);
};
class C : public A, public B {
    int c; void h(int);
};
C* pc;
pc->f(42);
((W*)pc)->g(42);
((A*)pc)->f(42);
```

Shared Base Class

```cpp
class W {
    int w; virtual void f(int);
    virtual void g(int);
    virtual void h(int);
};
class A : public virtual W {
    int a; void f(int);
};
class B : public virtual W {
    int b; void g(int);
};
class C : public A, public B {
    int c; void h(int);
};
C* pc;
pc->f(42);
((W*)pc)->g(42);
((A*)pc)->f(42);
```

Offsets to virtual base
Ambiguities
~ e.g. overwriting f in A and B
### Shared Base Class

```cpp
class W {
    int w; virtual void f(int);
    virtual void g(int);
    virtual void h(int);
};
class A : public virtual W {
    int a; void f(int);
};
class B : public virtual W {
    int b; void g(int);
};
class C : public A, public B {
    int c; int f(int);
};
```

### Again: Casting Issues

```cpp
class W { virtual int f(int); };
class A : virtual W { int a; };
class B : virtual W { int b; };
class C : public A, public B {
    int c; int f(int);
};
```

### Virtual Tables for Virtual Bases (≈ C++-ABI)

A Virtual Table for a Virtual Subclass

- Gets a virtual base pointer

A Virtual Table for a Virtual Base

- Virtual call offsets per virtual function for adjusting this dynamically
- Offset to top of an enclosing object's heap representation
- Typeinfo pointer to an RTTI object (not relevant for us)
- Virtual function pointers for resolving virtual methods

Virtual Base classes have virtual thunks which look up the offset to adjust the this pointer to the correct value in the virtual table!

### Compiler and Runtime Collaboration

#### Compiler generates:

- One code block for each method
- One virtual table for each class-composition, with
  - References to the most recent implementations of methods of a unique common signature (non-dispatching)
  - Sub-tables for the composed subclasses
  - Static top-of-object and virtual bases offsets per sub-table
  - (Virtual) thunks as this-adapters per method and subclass if needed

#### Runtime:

- At program startup virtual tables are globally created
- Allocation of memory space for each object followed by constructor calls
- Constructor stores pointers to virtual table (or fragments) in the objects
- Method calls transparently call methods statically or from virtual tables, unaware of real class identity
- Dynamic casts may use offset-to-top field in objects
Full Multiple Inheritance (FMI)
- Removes constraints on parents in inheritance
- More convenient and simple in the common cases
- Occurrence of diamond pattern not as frequent as discussions indicate

Multiple Interface Inheritance (MII)
- Simpler implementation
- Interfaces and aggregation already quite expressive
- Too frequent use of FMI considered as flaw in the class hierarchy design

Further reading...