"So how do we include several parent objects?"

```java
class A {
    int a; int f(int);
};
class B {
    int b; int g(int);
};
class C : public A, public B {
    int c; int h(int);
};
...
B* b = new C();

%1 = call i8* @new(i64 12)
call void @memset.p0i8.i64(%i8* %1, i8 0, i64 12, i32 4, i1 false)
%2 = getelementptr i8* %1, i64 4
%b = bitcast i8* %2 to %class.B
```
**Static Type Casts**

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

B* b = new C();
```

%1 = call i8* @new(i64 12)  
call void @_memset.p0i8.i64(i8* %1, i8 0, i64 12, i32 4, i1 false)  
%2 = getelementptr i8* %1, i64 4  ; select B-offset in C  
%b = bitcast i8* %2 to %class.B*  

⚠️ implicit casts potentially add a constant to the object pointer.  
⚠️ getelementptr implements ΔB as 4 - i8!

**Keeping Calling Conventions**

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

C c;  
c.g(42);
```

%c = alloca %class.C  
%1 = bitcast %class.C* %c to i8*  
%2 = getelementptr i8* %1, i64 4  
%3 = call i32 @i32 %class.B* %2  

**Ambiguities**

```c
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**

```
    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

MRO via DFS

Leftmost Preorder Depth-First Search

Principle 1: *inheritance* is violated

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

LPDFS with Duplicate Cancellation

A \{B, C\} A B C W

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

LPDFS with Duplicate Cancellation

A \{B, C\} B(V, W) C(W, V)

△ Principle 2: *multiplicity* not fulfillable

However, B → C → W → V??
MRO via Refined Postorder DFS

Reverse Postorder Rightmost DFS

L[A] = A B F D C E G H W

✓ Linear extension of inheritance relation

RPRDFS

L[A] = A B C D G E F

⚠️ But principle 2 *multiplicity* is violated!

Refined RPRDFS
**MRO via Refined Postorder DFS**

**Reverse Postorder Rightmost DFS**

$L[A] = A \ B \ F \ D \ C \ E \ G \ H \ W$

- Linear extension of inheritance relation
- Topological sorting

**RPRDFS**

$L[A] = A \ B \ C \ D \ G \ E \ F$

- But principle 2 *multiplicity* is violated!

**CLOS** uses Refined RPDFS [3]

**Refined RPRDFS**

$L[A] = A \ B \ C \ D \ E \ F \ G$

- Refine graph with conflict edge & rerun RPRDFS!

**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.

---

**Generalization and Specialization (G/S)**

- Multiple Inheritance
- Implementation of Multiple Inheritance
- Method Resolution Order

---

**MRO via Refined Postorder DFS**

**Refined RPRDFS**

- *Monotonicity* is not guaranteed!

**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 Refined Postorder DFS

**Refined RPRDFS**

⚠️ *Monotonicity* is not guaranteed!

**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.

\[
\begin{align*}
L[A] &= A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \\
L[C] &= D \rightarrow G \rightarrow E \\
F \rightarrow G & \Rightarrow L_\text{rdfs}(F) = \text{rdfs}(G) \\
G \rightarrow F & \Rightarrow L_\text{rdfs}(G) = \text{rdfs}(F)
\end{align*}
\]
C3 detects and reports a violation of *monotonicity* with the addition of \( A(B,C) \) to the class set.

C3 linearization [1]: is used in OpenDylan, Python, and Perl 6.
MRO via C3 Linearization

\[ L[G] \quad G \]
\[ L[F] \quad F \]
\[ L[E] \quad E \cdot F \]
\[ L[D] \quad D \cdot G \]
\[ L[B] \quad B \cdot F \cdot G \]
\[ L[C] \quad C \cdot D \cdot G \cdot E \cdot F \]
\[ L[A] \quad A \cdot B \cdot C \cdot D \cdot ((F \cdot G) \sqcup (G \cdot E \cdot F)) \]

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

\[ L[C] = C \cdot \bigcup \{L[B_1], \ldots, L[B_n], B_1 \cdots B_n\} \quad | \quad C(B_1, \ldots, B_n) \]
\[ L[Object] = Object \]

with

\[ \bigcup_i (L_i) = \begin{cases} c \cdot (\bigcup_i (L_i \setminus c)) & \text{if } \exists_{\min} k \forall j \ c = \text{head}(L_k) \neq \text{tail}(L_j) \\ \text{fail} & \text{else} \end{cases} \]
“And what about dynamic dispatching in Multiple Inheritance?”

Virtual Tables for Multiple Inheritance

```cpp
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);
```

Casting Issues

```cpp
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);
```
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();
b->f(42);

\[ \text{this-Pointer for } C::f \text{ is expected to point to } C \]

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; int f(int); };

B* b = &c;
b->f(42);

\[ \text{Virtual tables are composed when multiple inheritance is used} \]
\[ \text{The } \text{vptr fields in objects are pointers to their corresponding virtual-subtables} \]
\[ \text{Casting preserves the link between an object and its corresponding virtual-subtable} \]

Thunks

Solution: thunks

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

\[ \text{define } \text{id} \{ \text{func}(\text{this}, \text{vtable}, \text{args}) \}\]
\[ \text{\%1 = bitcast } \text{\%class.B to i32 } \text{\%this to i32}\]
\[ \text{\%2 = getelementptr } \text{\%1, \%1, i64 -16 ; size-of(A)=16}\]
\[ \text{\%3 = bitcast } \text{\%2 to } \text{\%class.C}\]
\[ \text{\%4 = call i32 } \text{\%3(\%class.C, \%3, \%3)}\]
\[ \text{return } \text{\%4}\]

\[ \Rightarrow \text{B-in-C-vtable entry for } f(\text{int}) \text{ is the thunk } \text{f(\text{int})} \]
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; int f(int);
};
...
C c;
B* pb = &c;
pb->f(42);

Basic Virtual Tables (\textarrow{\textarrow} C++-ABI)

A Basic Virtual Table
consists of different parts:
\begin{itemize}
  \item \textit{offset to top} of an enclosing objects heap representation
  \item \textit{typeid} pointer to an RTTI object (not relevant for us)
  \item \textit{virtual function pointers} for resolving virtual methods
\end{itemize}

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