Script generated by TTT

Title: Petter: Programmiersprachen (13.01.2016)

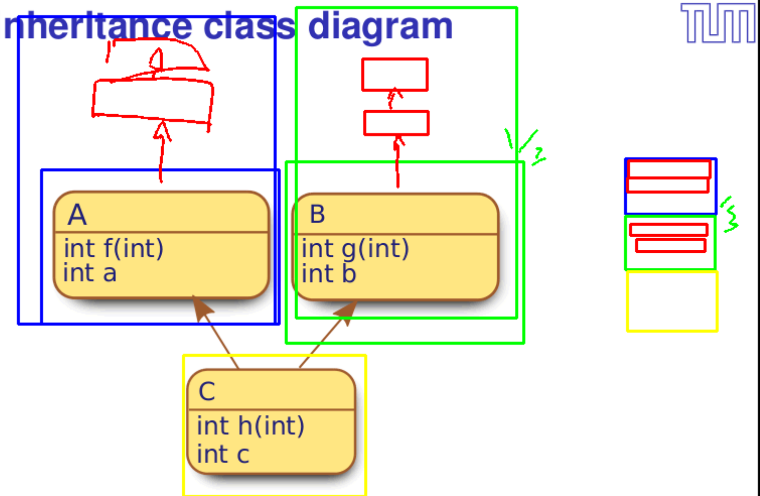
Date: Wed Jan 13 14:23:23 CET 2016

Duration: 101:38 min

Pages: 45

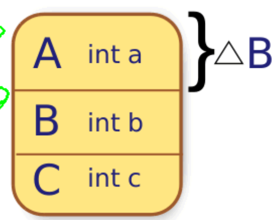
"So how do we include several parent objects?"

Multiple inheritance class diagram



Static Type Casts

```
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();
```

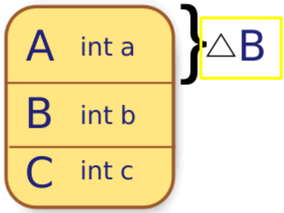



```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

```
%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*
```

Static Type Casts

```
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();
```




```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

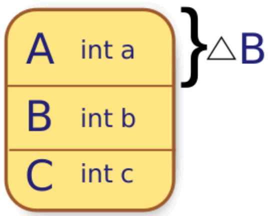
```
%1 = call i8* @_new(i64 12)
call void @_memset.p0i8.i64(i8* %1, i8 0, i64 12, i32 4, i1 false)
%2 = getelementptr i8* %1, i64 4          ; select B-offset in C
%b = bitcast i8* %2 to %class.B*
```

⚠ implicit casts potentially add a constant to the object pointer.

⚠ getelementptr implements ΔB as $4 \cdot i8$!

Keeping Calling Conventions

```
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);
```




```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

```
%c = alloca %class.C
%1 = bitcast %class.C* %c to i8*
%2 = getelementptr i8* %1, i64 4          ; select B-offset in C
%3 = call i32 @_g(%class.B* %2, i32 42)   ; g is statically known
```


Ambiguities

```
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:
$C(A,B) \implies C \to A \wedge C \to B$

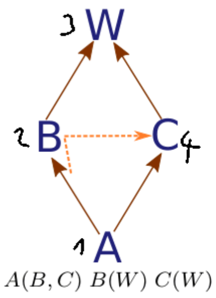
Principle 2: Multiplicity Relation
Defined by the succession of multiple parents. Example:
$C(A,B) \implies A \to B$

In General:

1. Inheritance is a uniform mechanism, and its searches ($\to$ 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

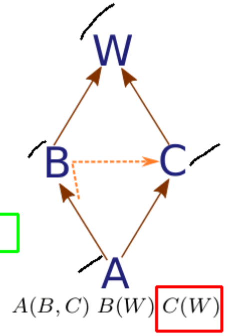
L[A] = A B W C

⚠ Principle 1 *inheritance* is violated

Python: classical python objects ($\leq$ 2.1) use LPDFS!

LPDFS with Duplicate Cancellation

L[A] = A B [illegible] C W



MRO via DFS

Leftmost Preorder Depth-First Search

L[A] = A B W C

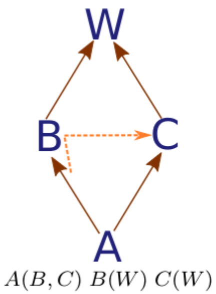
⚠ Principle 1 *inheritance* is violated

Python: classical python objects ($\leq$ 2.1) use LPDFS!

LPDFS with Duplicate Cancellation

L[A] = A B C W

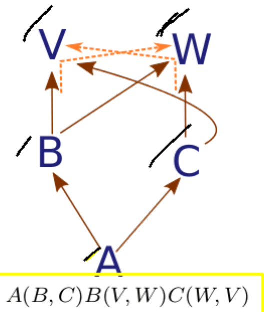
✓ Principle 1 *inheritance* is fixed



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

LPDFS with Duplicate Cancellation

A B [illegible] C W V



MRO via DFS

Leftmost Preorder Depth-First Search

L[A] = A B W C

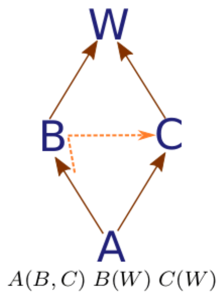
⚠ Principle 1 *inheritance* is violated

Python: classical python objects ($\leq$ 2.1) use LPDFS!

LPDFS with Duplicate Cancellation

L[A] = A B C W

✓ Principle 1 *inheritance* is fixed



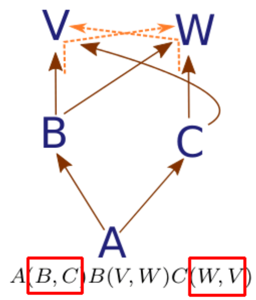
Python: new python objects (2.2) use LPDFS(DC)!

LPDFS with Duplicate Cancellation

L[A] = A B C W V

⚠ Principle 2 *multiplicity* not fulfillable

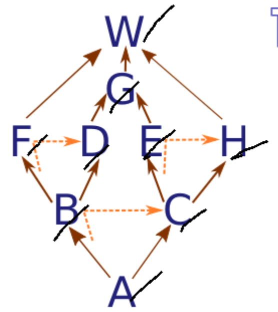
⚠ However $B \rightarrow C \implies W \rightarrow V$??



MRO via Refined Postorder DFS

Reverse Postorder Rightmost DFS

A B F D C E G H W



$A(B,C)$ $B(F,D)$ $C(E,H)$
$D(G)$ $E(G)$ $F(W)$ $G(W)$ $H(W)$

MRO via Refined Postorder DFS

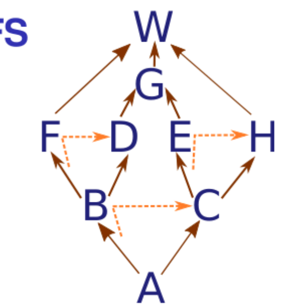
Reverse Postorder Rightmost DFS

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

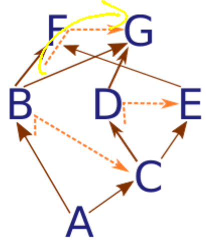
✓ Linear extension of inheritance relation

⇝ Topological sorting

RPRDFS



$A(B,C)$ $B(F,D)$ $C(E,H)$
$D(G)$ $E(G)$ $F(W)$ $G(W)$ $H(W)$



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

MRO via Refined Postorder DFS

Reverse Postorder Rightmost DFS

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

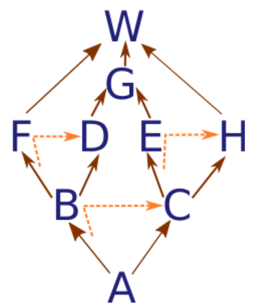
✓ Linear extension of inheritance relation

⇝ Topological sorting

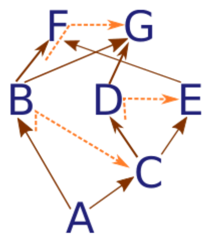
RPRDFS

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

⚠ But principle 2 *multiplicity* is violated!



$A(B,C)$ $B(F,D)$ $C(E,H)$
$D(G)$ $E(G)$ $F(W)$ $G(W)$ $H(W)$



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

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

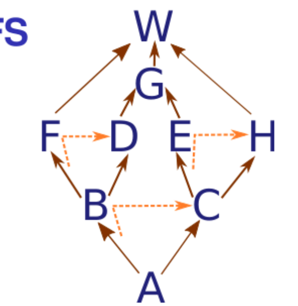
⇝ Topological sorting

RPRDFS

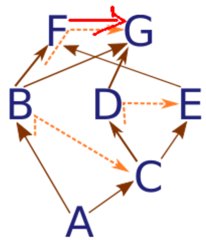
L[A] = A B C D G E F

⚠ But principle 2 *multiplicity* is violated!

Refined RPRDFS



$A(B,C)$ $B(F,D)$ $C(E,H)$
$D(G)$ $E(G)$ $F(W)$ $G(W)$ $H(W)$



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

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

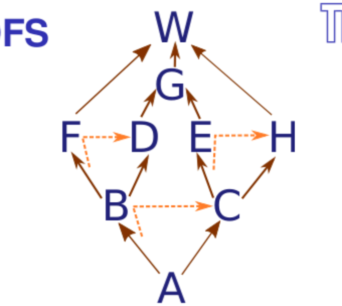
⚠ But principle 2 *multiplicity* is violated!

CLOS: uses Refined RPDFS [3]

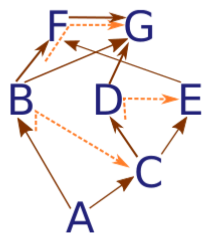
Refined RPRDFS

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

✓ Refine graph with conflict edge & rerun RPRDFS!



$A(B,C)$ $B(F,D)$ $C(E,H)$
$D(G)$ $E(G)$ $F(W)$ $G(W)$ $H(W)$

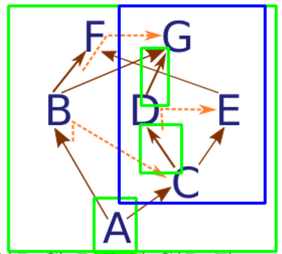


$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

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.



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

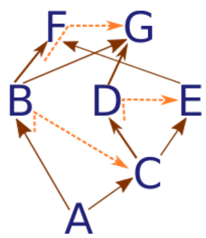
MRO via Refined Postorder DFS

Refined RPRDFS

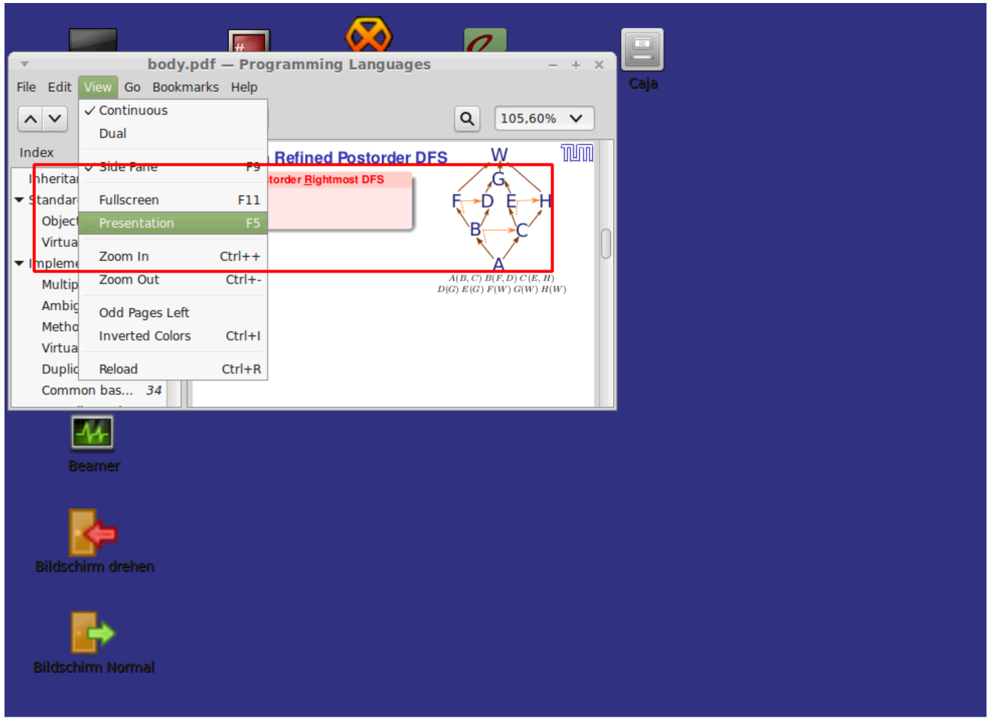
⚠ *Monotonicity* is not guaranteed!

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.



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$



MRO via Refined Postorder DFS

Refined RPRDFS

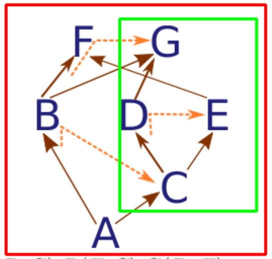
⚠ *Monotonicity* is not guaranteed!

Extension Principle: Monotonicity

If $C_1 \to C_2$ in C's linearization, then $C_1 \to C_2$ for every linearization of C's children.

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

$L[C] = D\ G\ E\ F \implies G \to F$



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

MRO via C3 Linearization

A linearization L is an attribute $L[C]$ 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

$$L[C] = C \cdot \bigsqcup(L[B_1], \ldots, L[B_n], B_1 \cdot \ldots \cdot B_n) \quad | \quad C(B_1, \ldots, B_n)$$

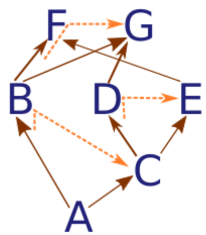
$$L[Object] = Object$$

with

$$\bigsqcup_i (L_i) = \begin{cases} c \cdot (\bigsqcup_i (L_i \setminus c)) & \text{if } \exists_{\text{min } k} \forall_j\ c = head(L_k) \notin tail(L_j) \\ ⚠\ \text{fail} & \text{else} \end{cases}$$

MRO via C3 Linearization

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

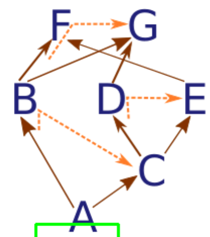


$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

MRO via C3 Linearization

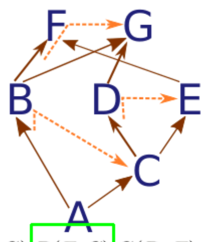
$L[G]\quad G$
$L[F]\quad F$
$L[E]\quad E \cdot F$
$L[D]\quad D \cdot G$
$L[B]$ $B \cdot \sqcup(L[F], L[G], F \cdot G) =$
$L[C]$
$L[A]$

$= B \cdot \sqcup(F, G, F \cdot G) =$
$= B \cdot (F \cdot \sqcup(G, G, G)) =$
$= B \cdot F \cdot G$



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

MRO via C3 Linearization

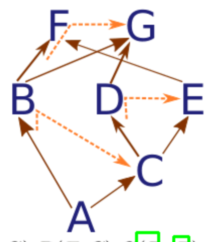


$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

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

$B \cdot \sqcup(L[F], L[G], F \cdot G) =$
$= B \cdot \sqcup(F, G, F \cdot G) =$
$= B \cdot (F \cdot \sqcup(G, G, G)) =$
$= B \cdot F \cdot G$

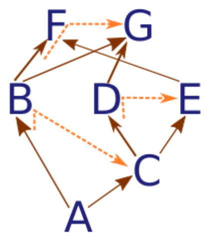
MRO via C3 Linearization



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

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

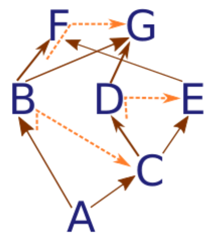
MRO via C3 Linearization



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

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

MRO via C3 Linearization



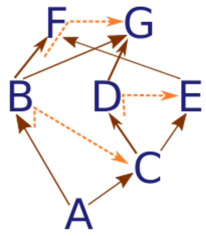
$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

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

$C \cdot D \cdot G \cdot E \cdot F$

MRO via C3 Linearization

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

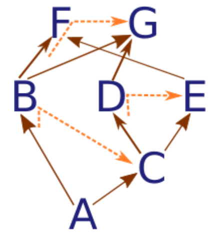


$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

MRO via C3 Linearization

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

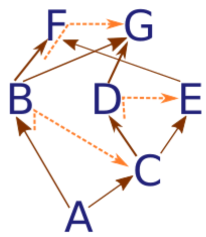
A · B · C · D



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

MRO via C3 Linearization

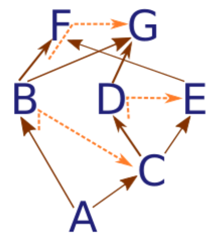
$L[G]$	G
$L[F]$	F
$L[E]$	$E \cdot F$
$L[D]$	$D \cdot G$
$L[B]$	$B \cdot F \cdot G$
$L[C]$	$C \cdot D \cdot G \cdot E \cdot F$
$L[A]$	⚠ fail



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

MRO via C3 Linearization

$L[G]$	G
$L[F]$	F
$L[E]$	$E \cdot F$
$L[D]$	$D \cdot G$
$L[B]$	$B \cdot F \cdot G$
$L[C]$	$C \cdot D \cdot G \cdot E \cdot F$
$L[A]$	⚠ fail

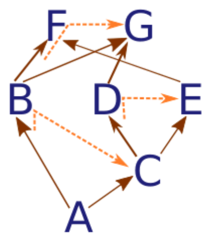


$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

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

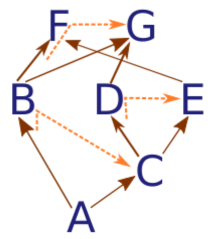
MRO via C3 Linearization



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

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

MRO via C3 Linearization



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

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

MRO via C3 Linearization

A linearization L is an attribute $L[C]$ 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

$$L[C] = C \cdot \bigsqcup(L[B_1], \ldots, L[B_n], B_1 \cdot \ldots \cdot B_n) \quad | \quad C(B_1, \ldots, B_n)$$

$$L[Object] = Object$$

with

$$\bigsqcup_i (L_i) = \begin{cases} c \cdot (\bigsqcup_i (L_i \setminus c)) & \text{if } \exists_{\text{min } k} \forall_j\ c = head(L_k) \notin tail(L_j) \\ ⚠ \text{ fail} & \text{else} \end{cases}$$

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

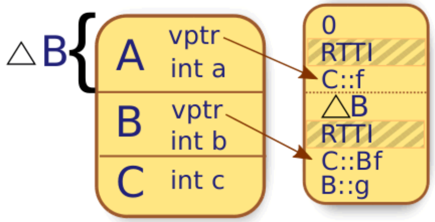
Languages with automatic linearization exist

- *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

```
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);
```

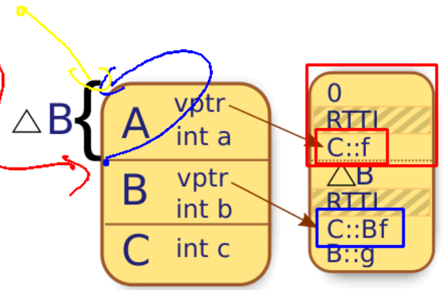



```
%class.C = type { %class.A, [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
%class.B = type { i32 (...)**, i32 }
```

```
; B* pb = &c;
%0 = bitcast %class.C* %c to i8*          ; type fumbling
%1 = getelementptr i8* %0, i64 16         ; offset of B in C
%2 = bitcast i8* %1 to %class.B*          ; get typing right
store %class.B* %2, %class.B** %pb        ; store to pb
```


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);
```



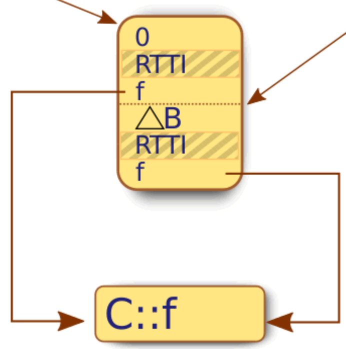

```
%class.C = type { %class.A, [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
%class.B = type { i32 (...)**, i32 }
```

```
; pb->f(42);
%0 = load %class.B** %pb                              ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***  ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                   ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0   ;select f() entry
%4 = load i32(%class.B*, i32)** %3                    ;load f()-thunk
%5 = call i32 %4(%class.B* %0, i32 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();
c->f(42);
```

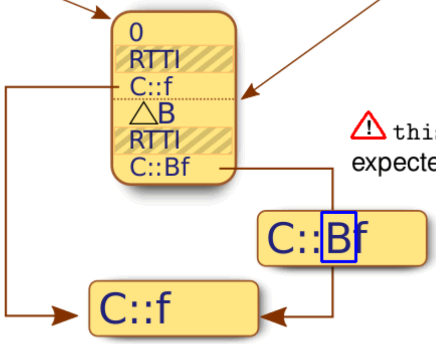
```
B* b = new C();
b->f(42);
```



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

```
B* b = new C();
b->f(42);
```



⚠ `this`-Pointer for `C::f` is expected to point to `C`

Thunks

Solution: *thunks*

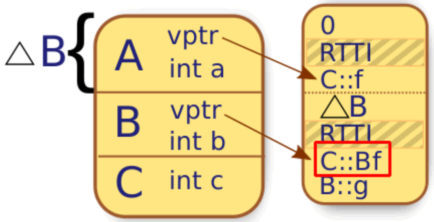
. . . are trampoline methods, delegating the virtual method to its original implementation with an adapted `this`-reference

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

⇝ `B-in-C-vtable` entry for `f(int)` is the thunk `_f(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);
```




```
%class.C = type { %class.A, [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
%class.B = type { i32 (...)**, i32 }
```

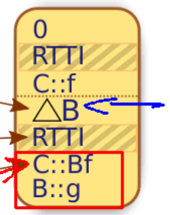
```
; pb->f(42);
%0 = load %class.B** %pb                                  ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***     ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                       ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0       ;select f() entry
%4 = load i32(%class.B*, i32)** %3                        ;load f()-thunk
%5 = call i32 %4(%class.B* %0, i32 42)
```


Basic Virtual Tables (⇝ C++-ABI)

A Basic Virtual Table

consists of different parts:

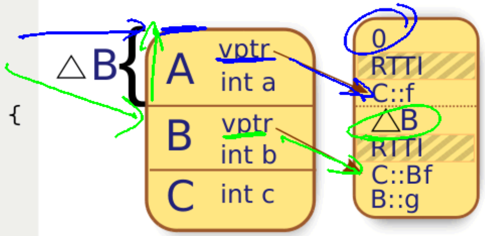
1. *offset to top* of an enclosing objects heap representation
2. *typeinfo pointer* to an RTTI object (not relevant for us)
3. *virtual function pointers* for resolving virtual methods



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

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);
```




```
%class.C = type { %class.A, [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
%class.B = type { i32 (...)**, i32 }
```

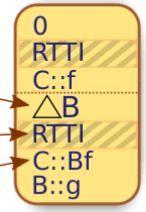
```
; pb->f(42);
%0 = load %class.B** %pb                                  ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***     ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                      ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0      ;select f() entry
%4 = load i32(%class.B*, i32)** %3                       ;load f()-thunk
%5 = call i32 %4(%class.B* %0, i32 42)
```


Basic Virtual Tables (⇝ C++-ABI)

A Basic Virtual Table

consists of different parts:

1. *offset to top* of an enclosing objects heap representation
2. *typeinfo pointer* to an RTTI object (not relevant for us)
3. *virtual function pointers* for resolving virtual methods



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