**Script** generated by TTT

Title: Petter: Programmiersprachen (21.12.2016)

Date: Wed Dec 21 14:16:13 CET 2016

Duration: 94:27 min

Pages: 45

## 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:
$C(A,B) \implies C \multimap A \wedge C \multimap B$

**Principle 2: Multiplicity Relation**

Defined by the succession of multiple parents. Example:
$C(A,B) \implies A \rightarrow B$

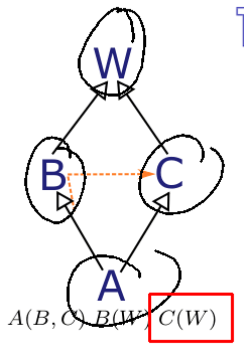
In General:

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

ABWC



## MRO via DFS

**Leftmost Preorder Depth-First Search**

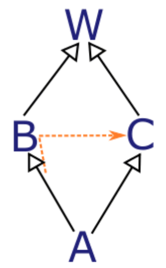
$$L[A] = ABWC$$

⚠ Principle 1 *inheritance* is violated

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

**LPDFS with Duplicate Cancellation**

AB~~W~~CW



$A(B,C)\ B(W)\ C(W)$

## MRO via DFS

**Leftmost Preorder Depth-First Search**

$$L[A] = ABWC$$

⚠ Principle 1 *inheritance* is violated

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

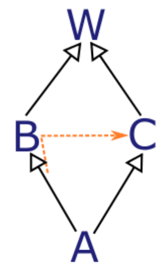
**LPDFS with Duplicate Cancellation**

$$L[A] = ABCW$$

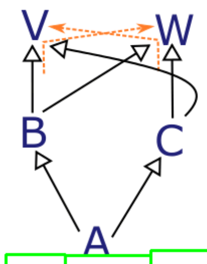
✓ Principle 1 *inheritance* is fixed

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

**LPDFS with Duplicate Cancellation**



$A(B,C)\ B(W)\ C(W)$



$A(B,C)B(V,W)C(W,V)$

## MRO via DFS

**Leftmost Preorder Depth-First Search**

$$L[A] = ABWC$$

⚠ Principle 1 *inheritance* is violated

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

**LPDFS with Duplicate Cancellation**

$$L[A] = ABCW$$

✓ Principle 1 *inheritance* is fixed

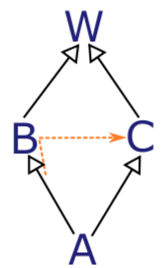
Python: new python objects (2.2) use LPDFS(DC)!

**LPDFS with Duplicate Cancellation**

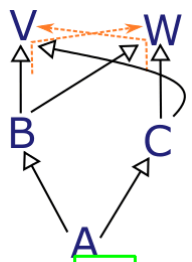
$$L[A] = \boxed{ABC\boxed{WV}}$$

⚠ Principle 2 *multiplicity* not fulfillable

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



$A(B,C)\ B(W)\ C(W)$

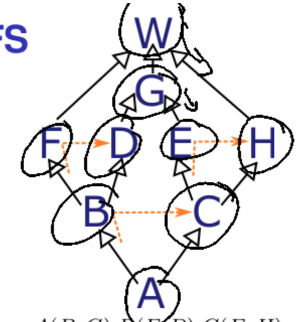


$A(B,C)B(V,W)C(W,V)$

## MRO via Refined Postorder DFS

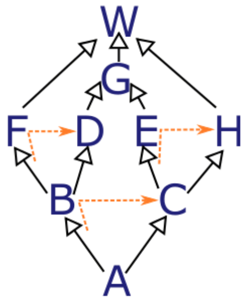
**Reverse ~~Postorder Rightmost DFS~~**

WHGECDFBA



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

## MRO via Refined Postorder DFS



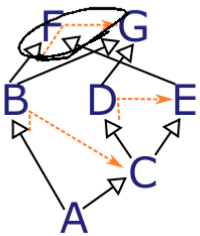

$A(B,C)\ B(F,D)\ C(E,H)$
$D(G)\ E(G)\ F(W)\ G(W)\ H(W)$

**Reverse Postorder Rightmost DFS**

$$L[A] = ABFDCEGHW$$

✓ Linear extension of inheritance relation

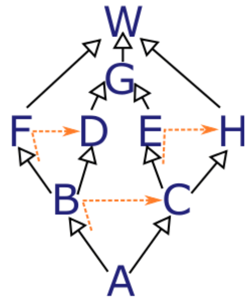
**RPRDFS**




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

---

## MRO via Refined Postorder DFS




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

**Reverse Postorder Rightmost DFS**

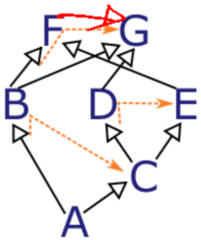
$$L[A] = ABFDCEGHW$$

✓ Linear extension of inheritance relation

**RPRDFS**

$$L[A] = ABCDGEF$$

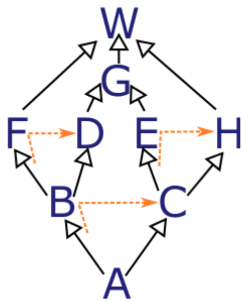
⚠ But principle 2 *multiplicity* is violated!




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

---

## MRO via Refined Postorder DFS




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

**Reverse Postorder Rightmost DFS**

$$L[A] = ABFDCEGHW$$

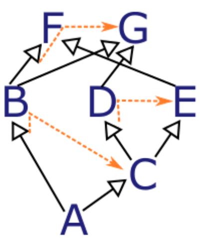
✓ Linear extension of inheritance relation

**RPRDFS**

$$L[A] = ABCDGEF$$

⚠ But principle 2 *multiplicity* is violated!

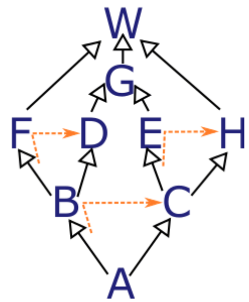
**Refined RPRDFS**




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

---

## MRO via Refined Postorder DFS




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

**Reverse Postorder Rightmost DFS**

$$L[A] = ABFDCEGHW$$

✓ Linear extension of inheritance relation

**RPRDFS**

$$L[A] = ABCDGEF$$

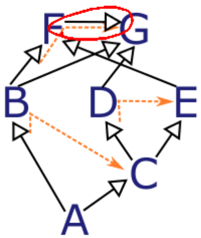
⚠ But principle 2 *multiplicity* is violated!

CLOS: uses Refined RPDFS [?]

**Refined RPRDFS**

$$L[A] = ABCDEFG$$

✓ Refine graph with conflict edge & rerun RPRDFS!

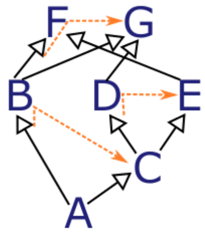



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

## 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.




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

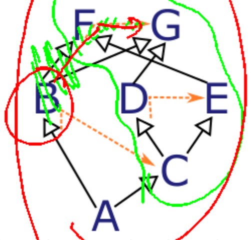
## MRO via Refined Postorder DFS

**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 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

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

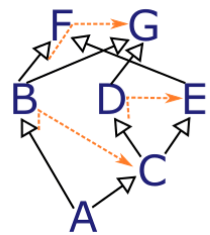
$$L[Object] = Object$$

with

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

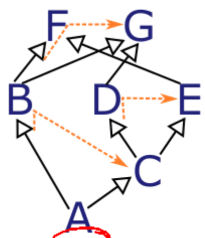
## MRO via C3 Linearization

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




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

# MRO via C3 Linearization



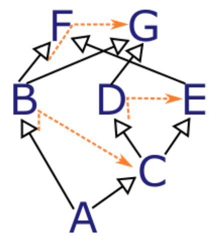

$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

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

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

$G$

# MRO via C3 Linearization



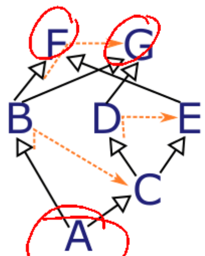

$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

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

$C \cdot \sqcup(D \cdot G, EF, DE)$

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

# MRO via C3 Linearization

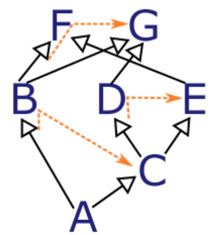



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

| | |
|---|---|
| $L[G]$ | $G$ |
| $L[F]$ | $F$ |
| $L[E]$ | $E \cdot F$ |
| $L[D]$ | $D \cdot G$ |
| $L[B]$ | $B \cdot F \cdot G$ |
| $L[C]$ | $C \cdot (L[D] \sqcup L[E] \sqcup (D \cdot E))$ |
| $L[A]$ | $O \; A \cdot \sqcup(BFG, CDGEF, BC)$ |

A B C D

# MRO via C3 Linearization




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

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

## 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

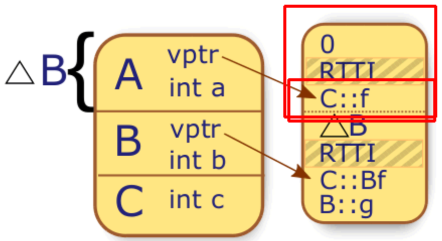
**Languages with automatic linearization exist**

- *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);
```

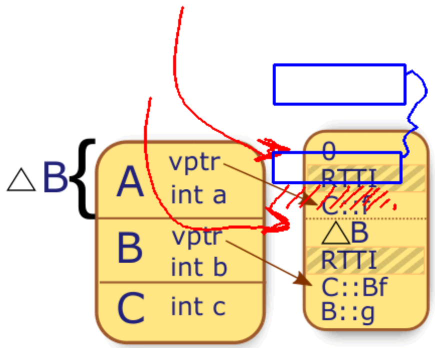



```
%class.C = type { %class.A, [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
%class.B = type { i32 (...)**, i32 }
```

```
; 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);
```

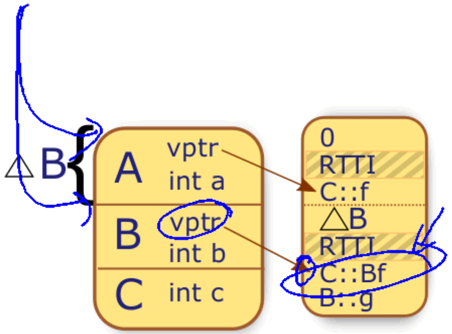



```
%class.C = type { %class.A, [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
%class.B = type { i32 (...)**, i32 }
```

```
; 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)
```

## 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 }
```

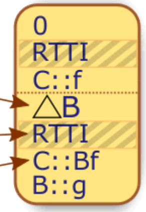
```
; pb->f(42);
%0 = load %class.B** %pb                                  ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***      ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                       ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0       ;select 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 memory representation
2. *typeinfo pointer* to an RTTI object (not relevant for us)
3. *virtual function pointers* for resolving virtual methods

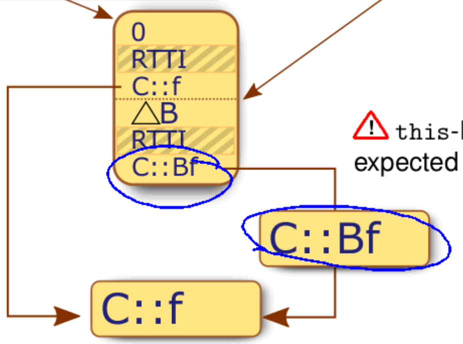



- Virtual tables are composed when multiple inheritance is used
- The `vptr` fields in objects are pointers to their corresponding virtual-subtables
- Casting preserves the link between an object and its corresponding virtual-subtable
- `clang -cc1 -fdump-vtable-layouts -emit-llvm code.cpp` yields the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

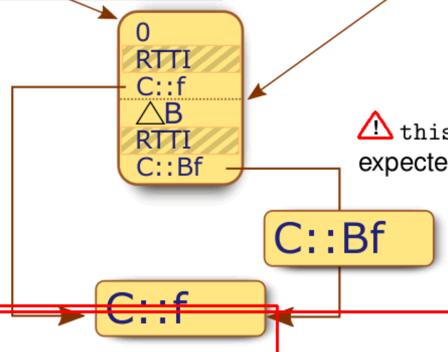
```
define i32 @__f(%class.B* %this, i32 %i) {
  %1 = bitcast %class.B* %this to i8*
  %2 = getelementptr i8* %1, i64 -16          ; sizeof(A)=16
  %3 = bitcast i8* %2 to %class.C*
  %4 = call i32 @_f(%class.C* %3, i32 %i)
  ret i32 %4
}
```

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

## 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);
```

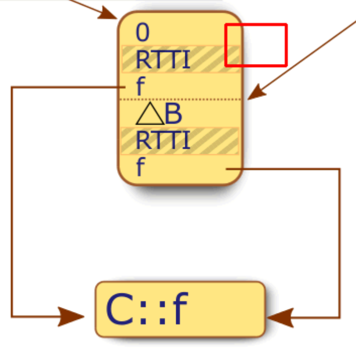



⚠ `this`-Pointer for `C::f` is expected to point to `C`

## 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);
```

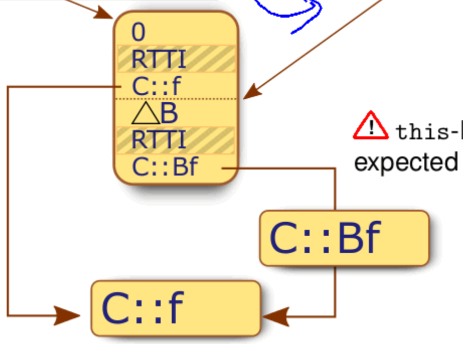



⚠ `this`-Pointer for `C::f` is expected to point to `C`

## 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);
```

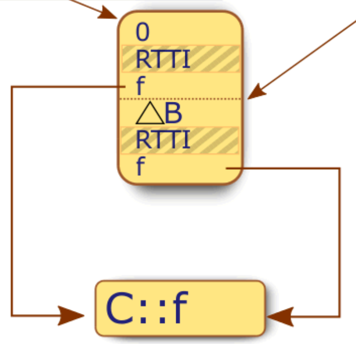



⚠ `this`-Pointer for `C::f` is expected to point to `C`

## Casting Issues

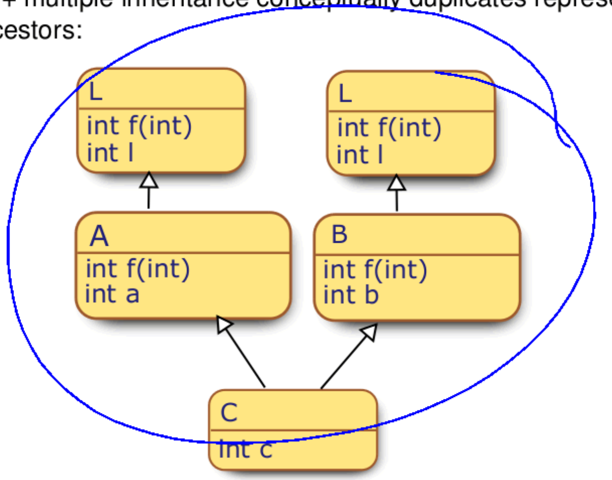
```
class A { int a; };
class B { virtual int f(int);};
class C : public A , public B {
  int c; int f(int);
};
C* c = new C();
c->f(42);
```

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



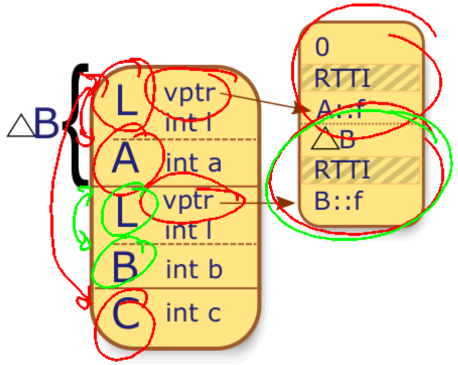
## Common Bases – Duplicated Bases

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




## Duplicated Base Classes

```
class L {
  int l; 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;
```



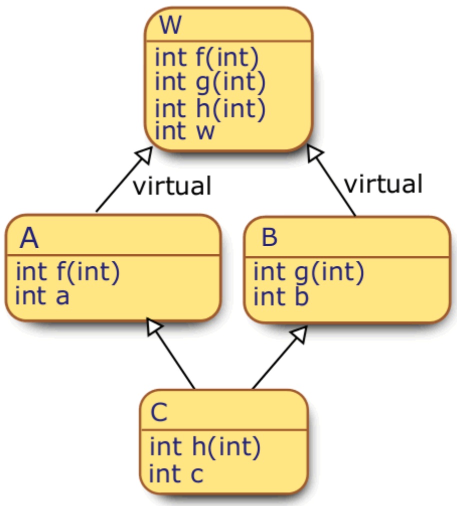

```
%class.C = type { %class.A, %class.B,
                  i32, [4 x i8] }
%class.A = type { [12 x i8], i32 }
%class.B = type { [12 x i8], i32 }
%class.L = type { i32 (...)**, i32 }
```

⚠ Ambiguity!

```
L* pl = (A*)&c;
C* pc = (C*)(A*)pl;
```

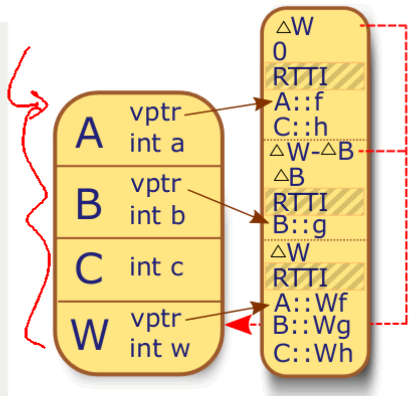
## Common Bases – Shared Base Class

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




## 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);
};
...
C* pc;
pc->f(42);
((W*)pc)->h(42);
((A*)pc)->f(42);
```




⚠ Offsets to virtual base

⚠ Ambiguities

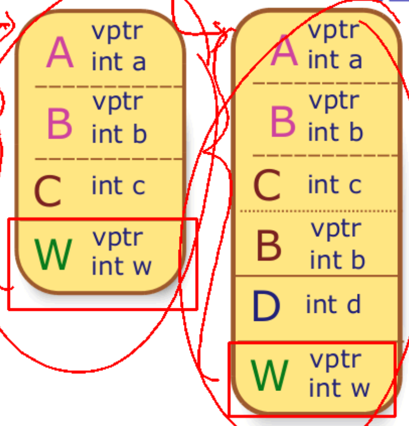
⇝ e.g. overwriting f in A *and* B

## Dynamic Type Casts

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

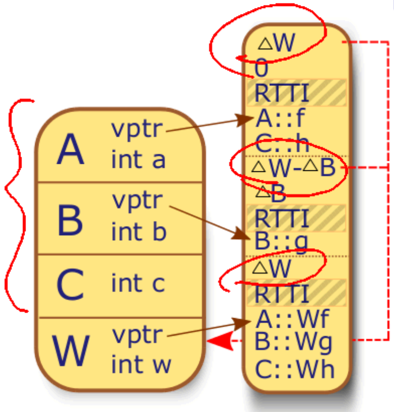
vs.

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




## 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);
};
...
C* pc;
pc->f(42);
((W*)pc)->h(42);
((A*)pc)->f(42);
```



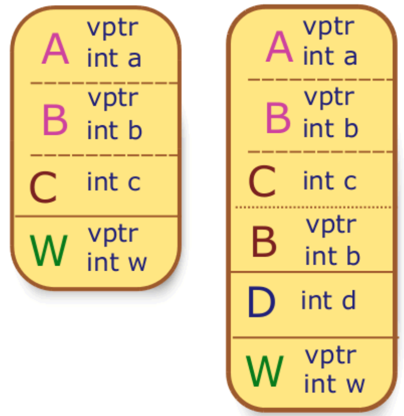

- ⚠ Offsets to virtual base
- ⚠ Ambiguities
- ⇝ e.g. overwriting f in A *and* B


## Dynamic Type Casts

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

vs.

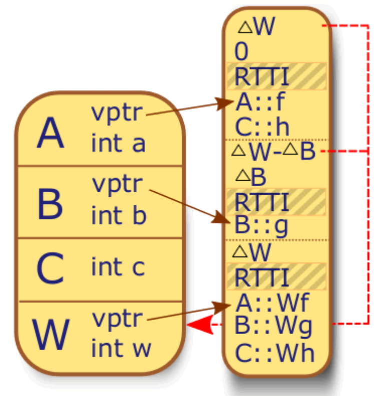
```
C* pc = dynamic_cast<C*>(pw);
```



## 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);
};
...
C* pc;
pc->f(42);
((W*)pc)->h(42);
((A*)pc)->f(42);
```

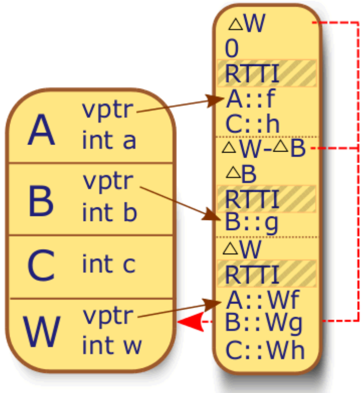





- ⚠ Offsets to virtual base
- ⚠ Ambiguities
- ⇝ e.g. overwriting f in A *and* B

## 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);
};
...
C* pc;
pc->f(42);
((W*)pc)->h(42);
((A*)pc)->f(42);
```




⚠ Offsets to virtual base

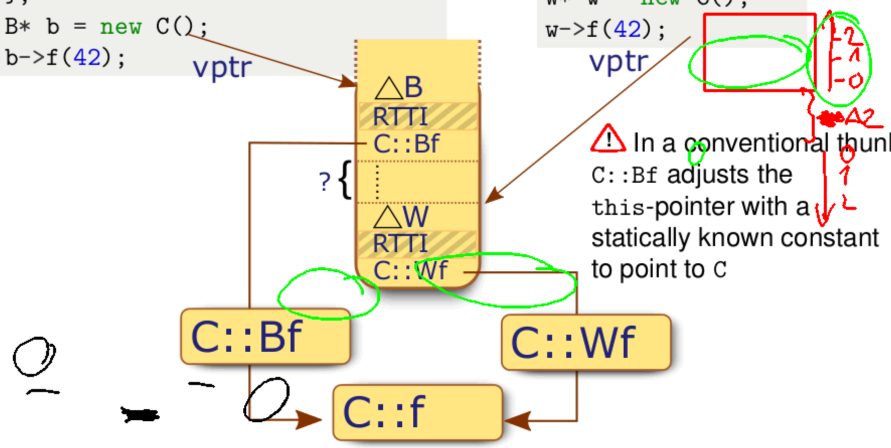
⚠ Ambiguities

⇝ e.g. overwriting f in A *and* B

## Again: Casting Issues

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

```
W* w = new C();
w->f(42);
```




⚠ In a conventional thunk `C::Bf` adjusts the `this`-pointer with a statically known constant to point to `C`

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

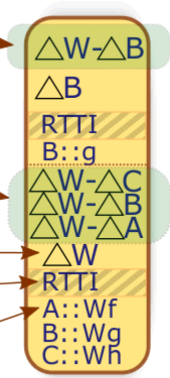
**A Virtual Table for a Virtual Subclass**

gets a *virtual base pointer*

**A Virtual Table for a Virtual Base**

consists of different parts:

1. *virtual call offsets* per virtual function for adjusting `this` dynamically
2. *offset to top* of an enclosing objects heap representation
3. *typeinfo pointer* to an RTTI object (not relevant for us)
4. *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:

1. ... one code block for each method
2. ... one virtual table for each class-composition, with
   - references to the most recent implementations of methods of a *unique common signature* (⇝ single 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:

1. At program startup virtual tables are globally created
2. Allocation of memory space for each object followed by constructor calls
3. Constructor stores pointers to virtual table (or fragments) in the objects
4. Method calls transparently call methods statically or from virtual tables, *unaware of real class identity*
5. Dynamic casts may use *offset-to-top* field in objects

# Polemics of Multiple Inheritance

**Full Multiple Inheritance (FMI)**

- Removes constraints on parents in inheritance
- More convenient and simple in the common cases
- Occurance 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...


K. Barrett, B. Cassels, P. Haahr, D. Moon, K. Playford, and T. Withington.
A monotonic superclass linearization for dylan.
In *Object Oriented Programming Systems, Languages, and Applications*, 1996.

CodeSourcery, Compaq, EDG, HP, IBM, Intel, R. Hat, and SGI.
Itanium C++ ABI.
URL: http://www.codesourcery.com/public/cxx-abi.

R. Ducournau and M. Habib.
On some algorithms for multiple inheritance in object-oriented programming.
In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1987.

R. Kleckner.
Bringing clang and llvm to visual c++ users.
URL: http://llvm.org/devmtg/2013-11/#talk11.

B. Liskov.
Keynote address – data abstraction and hierarchy.
In *Addendum to the proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 17–34, 1987.

L. L. R. Manual.
Llvm project.
URL: http://llvm.org/docs/LangRef.html.

R. C. Martin.
The liskov substitution principle.
In *C++ Report*, 1996.

P. Sabanal and M. Yason.
Reversing c++.
In *Black Hat DC*, 2007.
URL: https://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf.

B. Stroustrup.
Multiple inheritance for C++.
In *Computing Systems*, 1999.