TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK

Programming Languages

Multiple Inheritance

Dr. Michael Petter

Winter Term 2021

Outline

Inheritance Principles

1. Interface Inheritance
2. Implementation Inheritance
3. Dispatching implementation choices

C++ Object Heap Layout

1. Basics
2. Single-Inheritance
3. Virtual Methods

C++ Multiple Parents Heap Layout

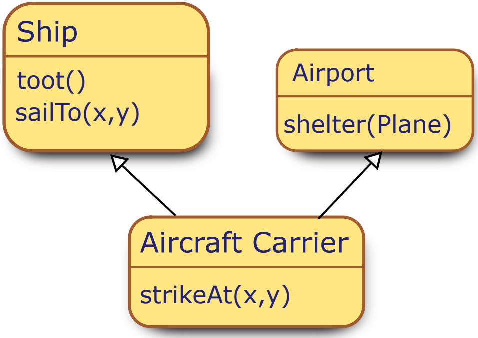
1. Multiple-Inheritance
2. Virtual Methods
3. Common Parents

Excursion: Linearization

1. Ambiguous common parents
2. Principles of Linearization
3. Linearization algorithms

“Wouldn’t it be nice to inherit from several parents?”

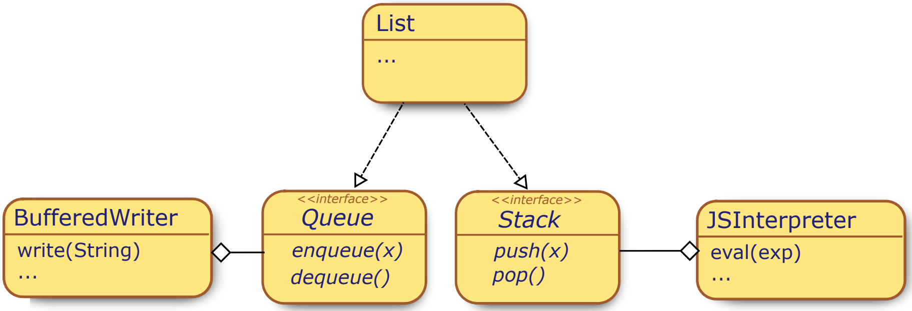
Implementation inheritance



The classic motivation for inheritance is implementation inheritance

- *Code reusage*
- Parent acts as library of common behaviours
- Child specializes parents, replacing particular methods with custom ones

Interface Inheritance



Code sharing in interface inheritance inverts this relation

- *Behaviour contract*
- Parent formalizes type requirements to participate in shared code frames
- Child provides functionality in methods whith signatures predefined by the parent

“So how do we lay out objects in memory anyway?”

Excursion: Brief introduction to LLVM IR

LLVM intermediate representation as reference semantics:

```
;(recursive) struct definitions
%struct.A = type { i32, %struct.B, i32(i32)* }
%struct.B = type { i64, [10 x [20 x i32]], i8 }

;(stack-) allocation of objects
%a = alloca %struct.A
;adress computation for selection in structure (pointers):
%1 = getelementptr %struct.A* %a, i64 0, i64 2
;load from memory
%2 = load i32(i32)* %1
;indirect call
%retval = call i32 (i32)* %2(i32 42)
```

Retrieve the memory layout of a compilation unit with:

```
clang -cc1 -x c++ -v -fdump-record-layouts -emit-llvm source.cpp
```

Retrieve the IR Code of a compilation unit with:

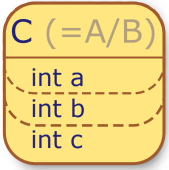
```
clang -O1 -S -emit-llvm source.cpp -fno-discard-value-names -o /dev/stdout | llvm-cxxfilt
```

Object layout

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

...

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

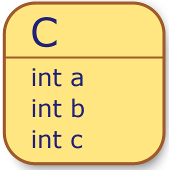



```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

```
%c = alloca %class.C
%1 = bitcast %class.C* %c to %class.B*
%2 = call i32 @_g(%class.B* %1, i32 42) ; g is statically known
```

Translation of a method body

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




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

```
define i32 @_g(%class.B* %this, i32 %p) {
  %1 = getelementptr %class.B* %this, i64 0, i32 1
  %2 = load i32* %1
  %3 = add i32 %2, %p
  ret i32 %3
}
```

“Now what about polymorphic calls?”

Single-Dispatching implementation choices

Single-Dispatching needs runtime action:

1. Manual search run through the super-chain (Java Interpreter ⇝ last talk)

```
call i32 @__dispatch(%class.C* %c,i32 42,i32* "f(int,void)")
```

2. Caching the dispatch result (⇝ Hotspot/JIT)

```
; caching the recent result value of the __dispatch function
; call i32 @__dispatch(%class.C* %c,i32 42)
assert (%c type %class.D) ; verify objects class presumption
call i32 @_f_from_D(%class.C* %c, i32 42) ; directly call f
```

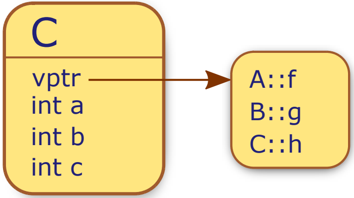
3. Precomputing the dispatching result in tables
   1. Full 2-dim matrix
   2. 1-dim Row Displacement Dispatch Tables
   3. Virtual Tables
      (⇝ LLVM/GNU C++,this talk)

	f()	g()	h()	i()	j()	k()	l()	m()	n()
A	1								
B	1	2							
C	3		4						
D	3	2	4	5					
E						6		7	
F					8	9		7	

A	B			F			
1	1	2	...	8	9		7

Object layout – virtual methods

```
class A {                public:
  int a; virtual int f(int);
         virtual int g(int);
         virtual int h(int);
};
class B : public A { public:
  int b; int g(int);
};
class C : public B { public:
  int c; int h(int);
};  ...
C c;
c.g(42);
```

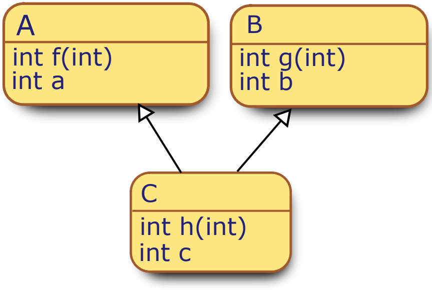



```
%class.C = type { %class.B, i32, [4 x i8] }
%class.B = type { [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
```

```
%c.vptr = bitcast %class.C* %c to i32 (%class.B*, i32)*** ; vtbl
%1 = load (%class.B*, i32)*** %c.vptr     ; dereference vptr
%2 = getelementptr %1, i64 1              ; select g()-entry
%3 = load (%class.B*, i32)** %2           ; dereference g()-entry
%4 = call i32 %3(%class.B* %c, i32 42)
```

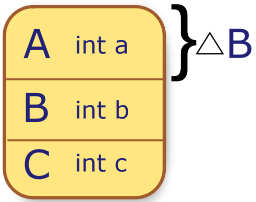
“So how do we include several parent objects?”

Multiple inheritance class diagram



Static Type Casts

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

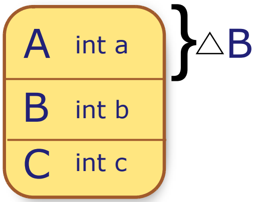
```
%1 = call i8* @_new(i64 12)
call void @_memset.p0i8.i64(i8* %1, i8 0, i64 12, i32 4, i1 false)
%2 = getelementptr i8* %1, i64 4            ; select B-offset in C
%b = bitcast i8* %2 to %class.B*
```

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

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

Keeping Calling Conventions

```
class A { public:
  int a; int f(int);
};
class B { public:
  int b; int g(int);
};
class C : public A , public B {
  public:
  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 { public: void f(int); };
class B { public: 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:
$C(A, B) \implies C \mathrel{-\!\!\triangleright} A \land C \mathrel{-\!\!\triangleright} B$

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

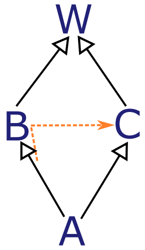
MRO via DFS

Leftmost Preorder Depth-First Search

$$L[A] = ABWC$$

⚠ Principle 1 *inheritance* is violated

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



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

LPDFS with Duplicate Cancellation

$$L[A] = ABCW$$

✓ Principle 1 *inheritance* is fixed

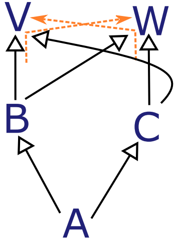
Python: new python objects (2.2) use LPDFS(DC)!

LPDFS with Duplicate Cancellation

$$L[A] = ABCWV$$

⚠ Principle 2 *multiplicity* not fulfillable

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



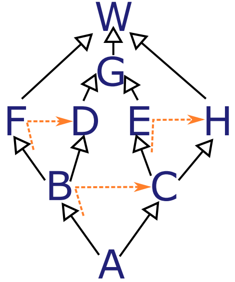
$A(B,C)B(V,W)C(W,V)$

MRO via Refined Postorder DFS

<u>R</u>everse <u>P</u>ostorder <u>R</u>ightmost DFS

$$L[A] = ABFDCEGHW$$

✓ Linear extension of inheritance relation



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

RPRDFS

$$L[A] = ABCDGEF$$

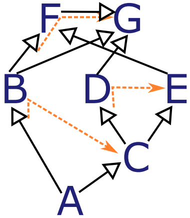
⚠ But principle 2 *multiplicity* is violated!

CLOS: uses Refined RPDFS [3]

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

Refined RPRDFS

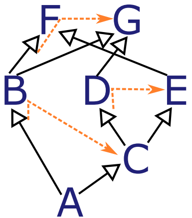
⚠ *Monotonicity* is not guaranteed!

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.

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

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



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

MRO via C3 Linearization

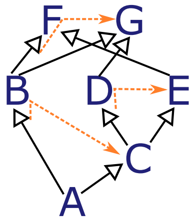
A linearization L is an attribute $L[C]$ of a class C. Classes $B_1, \ldots, B_n$ are superclasses to child class C, defined in the *local precedence order* $C(B_1 \ldots B_n)$. Then

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

with

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

MRO via C3 Linearization



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

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

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 *Python 3*, *Raku*, and *Solidity*

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

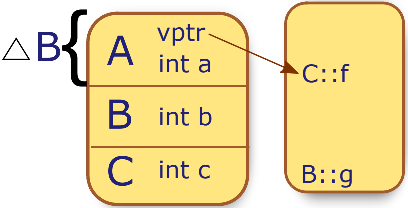
Languages with automatic linearization exist

- *CLOS* Common Lisp Object System
- *Solidity, Python 3* and *Raku* with C3
- Prerequisite for → Mixins

“And what about dynamic dispatching in Multiple Inheritance?”

Virtual Tables for Multiple Inheritance

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

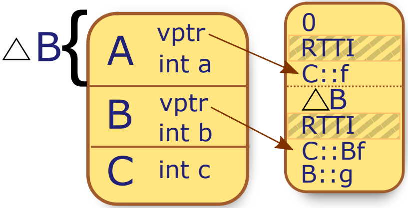



```
%class.C = type { %class.A, [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
%class.B = type { i32 (...)**, i32 }
```

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

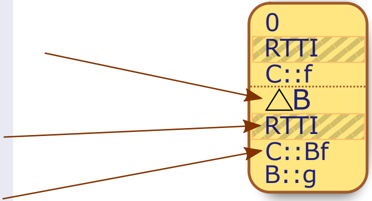
```
; pb->f(42);
%0 = load %class.B** %pb                               ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***   ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                    ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0    ;select f() entry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

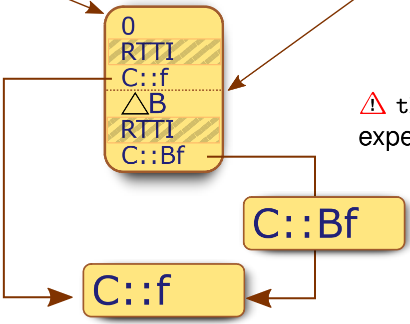


- Virtual tables are composed when multiple inheritance is used
- The `vptr` fields in objects are pointers to their corresponding virtual-subtables
- Casting preserves the link between an object and its corresponding virtual-subtable
- `clang -cc1 -fdump-vtable-layouts -emit-llvm source.cpp` yields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

Casting Issues

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

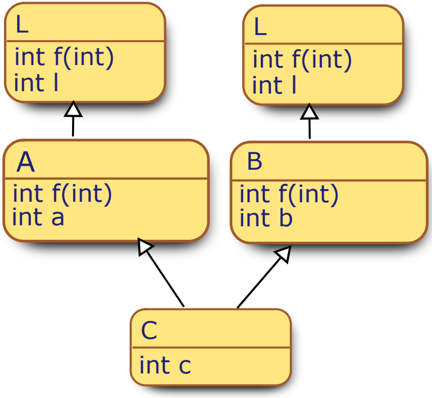
```
define i32 @__f(%class.B* %this, i32 %i) {
  %1 = bitcast %class.B* %this to i8*
  %2 = getelementptr i8* %1, i64 -16      ; sizeof(A)=16
  %3 = bitcast i8* %2 to %class.C*
  %4 = call i32 @_f(%class.C* %3, i32 %i)
  ret i32 %4
}
```

- ⇝ `B-in-C`-vtable entry for `f(int)` is the thunk `_f(int)`
- ⇝ `_f(int)` adds a compiletime constant ΔB to `this` before calling `f(int)`
- ⇝ `f(int)` addresses its locals relative to what it assumes to be a `C` pointer

“But what if there are common ancestors?”

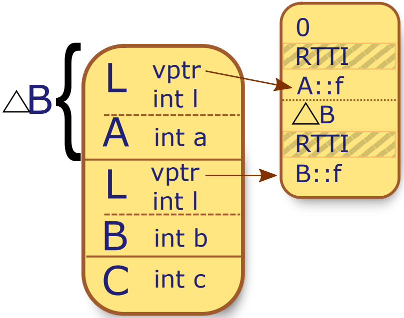
Common Bases – Duplicated Bases

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



Duplicated Base Classes

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

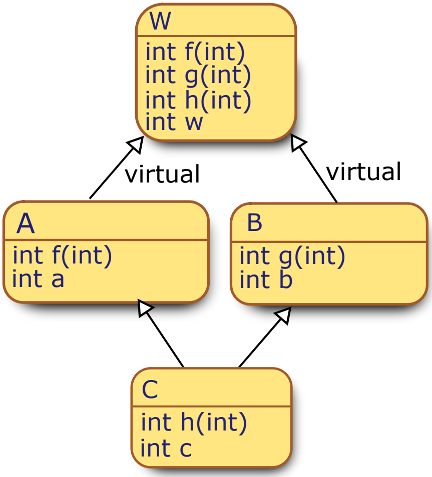



```
%class.C = type { %class.A, %class.B,
                  i32, [4 x i8] }
%class.A = type { [12 x i8], i32 }
%class.B = type { [12 x i8], i32 }
%class.L = type { i32 (...)**, i32 }
```

⚠ Ambiguity!

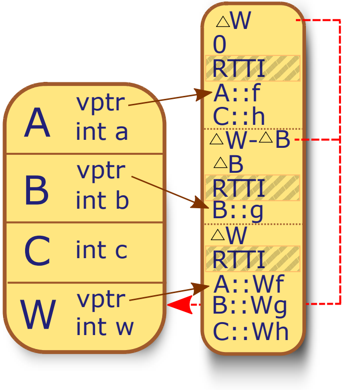
Common Bases – Shared Base Class

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



Shared Base Class

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



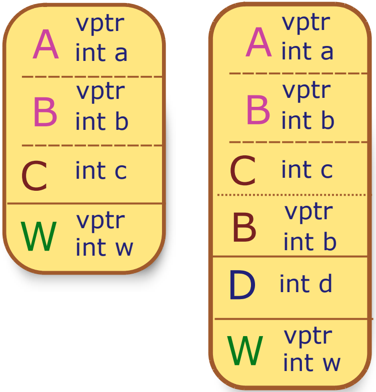
⚠ Ambiguities

⇝ e.g. overriding f in A *and* B

⚠ Offsets to virtual base

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

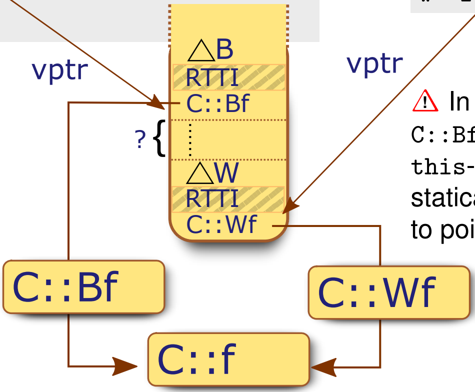


⚠ No guaranteed *constant* offsets between virtual bases and subclasses ⇝ No static casting!
⚠ *Dynamic casting* makes use of *offset-to-top*

Again: Casting Issues

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

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

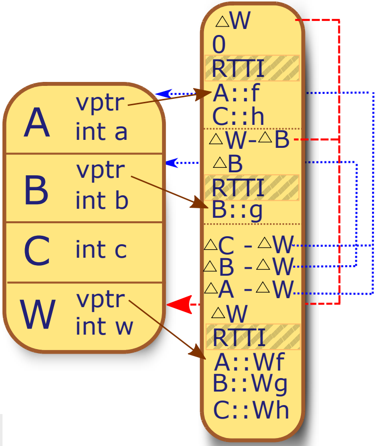


⚠ In a conventional thunk `C::Bf` adjusts the `this`-pointer with a statically known constant to point to `C`

Virtual Thunks

```
class W { ...
   public: virtual void g(int);
};
class A : public virtual W {...};
class B : public virtual W {
   public: virtual void g(int i){ }; int b;
};
class C : public A,public B{...};
C c;
W* pw = &c;
pw->g(42);
```

```
define void @__g(%class.W* %this, i32 %i) { ; virtual thunk to B::g
  %1 = bitcast %class.W* %this to i8*
  %2 = bitcast i8* %1 to i8**
  %3 = load i8** %2                      ; load W-vtable ptr
  %4 = getelementptr i8* %3, i64 -32 ; -32 bytes is g-entry in vcall offsets
  %5 = bitcast i8* %4 to i64*
  %6 = load i64* %5                      ; load g's vcall offset
  %7 = getelementptr i8* %1, i64 %6  ; navigate to Wtop+vcalloffset
  %8 = bitcast i8* %7 to %class.B*
  call void @_g(%class.B* %8, i32 %i)
  ret void
}
```



Virtual Tables for Virtual Bases ($\rightsquigarrow$ C++-ABI)

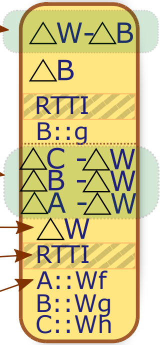
A Virtual Table for a Virtual Subclass

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* ($\leadsto$ 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

Lessons Learned

Lessons Learned

1. Different purposes of inheritance
2. Heap Layouts of hierarchically constructed objects in C++
3. Virtual Table layout
4. LLVM IR representation of object access code
5. Linearization as alternative to explicit disambiguation
6. Pitfalls of Multiple Inheritance

Sidenote for MS VC++

- the presented approach is implemented in GNU C++ and LLVM
- Microsoft's MS VC++ approaches multiple inheritance differently
  - splits the virtual table into several smaller tables
  - keeps a vbptr (virtual base pointer) in the object representation, pointing to the virtual base of a subclass.

Further reading...

[1] 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.

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

[3] 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.

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

[5] 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.

[6] L. L. R. Manual.
Llvm project.
URL: `http://llvm.org/docs/LangRef.html`.

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

[8] 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`.

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

Mini Seminars

1. SC=CC in Multicore Architectures with Cache (Meixner/Sorin 2006/2009)
2. Litmus Testing Memory Models: Herdtools 7
3. The Linux Kernel Memory Model
4. A Formal Analysis of the NVIDIA PTX Memory Consistency Model (2019)
5. GPU Concurrency: Weak Behaviours and Programming Assumptions (2015)
6. Transactional Memory Systems other than TSX: IBM Power 8 / BlueGene / zEnterprise
7. Lambda Calculus: Y Combinator and Recursion / SKI Combinator
8. Templates vs. Inheritance