Programming Languages

Multiple Inheritance

Dr. Michael Petter
Winter term 2018
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?”
The classic motivation for inheritance is implementation inheritance

- **Code reusage**
  - Child specializes parents, replacing particular methods with custom ones
  - Parent acts as library of common behaviours
  - Implemented in languages like C++ or Lisp

Code sharing in interface inheritance inverts this relation

- **Behaviour contract**
  - Child provides methods, with signatures predetermined by the parent
  - Parent acts as generic code frame with room for customization
  - Implemented in languages like Java or C#
Interface Inheritance

List
...

Queue
enqueue(x)
dequeue()  

Stack
push(x)
pop()

CircularGraph
insertNodeAt(x,i)
removeNodeAt(x,i)
“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 -o IR.llvm
class A {
    int a; int f(int);
};
class B : public A {
    int b; int g(int);
};
class C : public B {
    int c; int h(int);
};

C c;
c.g(42);

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

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

2. Caching the dispatch result (\(\leadsto\) 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
   - Full 2-dim matrix
   - 1-dim Row Displacement Dispatch Tables
   - Virtual Tables (\(\leadsto\) LLVM/GNU C++, this talk)

   ![Virtual Table Example]

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

```
Object layout – virtual methods

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

A
int f(int)
int a

B
int g(int)
int b

C
int h(int)
int 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();
```

%! = 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!
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 ; select B-offset in C
%3 = call i32 @_g(%class.B* %2, i32 42) ; g is statically known
Ambiguities

```cpp
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**

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

**Solution II: Automagical resolution**

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

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.

**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: $C(A, B) \implies A \rightarrow B$
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

\[ L[A] = ABCW \]

✓ 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 \( B \rightarrow C \implies W \rightarrow V \)??
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!

CLOS: uses Refined RPDFS [3]

Refined RPRDFS

\[ L[A] = ABCDEFG \]

✓ Refine graph with conflict edge & rerun RPRDFS!
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.

\[
L[A] = A \; B \; C \; D \; E \; F \; G \quad \implies \quad F \rightarrow G
\]

\[
L[C] = C \; D \; G \; E \; F \quad \implies \quad G \rightarrow F
\]
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 \textit{local precedence order} $C(B_1 \ldots B_n)$. Then

$$L[C] = C \cdot \bigsqcup(L[B_1], \ldots, L[B_n], B_1 \cdots B_n) \mid C(B_1, \ldots, B_n)$$

$$L[Object] = Object$$

with

$$\bigsqcup_i(L_i) = \begin{cases} c \cdot (\bigsqcup_i(L_i \setminus c)) & \text{if } \exists_{\min k} \forall_j \ c = head(L_k) \not\in tail(L_j) \\ \triangle left \fail & \text{else} \end{cases}$$
MRO via C3 Linearization

\[
\begin{align*}
L[G] & \quad G \\
L[F] & \quad F \\
L[E] & \quad E \cdot F \\
L[D] & \quad D \cdot G \\
L[B] & \quad B \cdot F \cdot G \\
L[C] & \quad C \cdot D \cdot G \cdot E \cdot F \\
L[A] & \quad \text{fail}
\end{align*}
\]

C3 detects and reports a violation of \textit{monotonicity} with the addition of \(A(B,C)\) to the class set.

\textbf{C3 linearization} [1]: is used in \textit{Python 3, Perl 6, 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 *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);

; 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);

; pb->f(42);
%0 = load %class.B** %pb
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***
%2 = load i32(%class.B*, i32)*** %1
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0
%4 = load i32(%class.B*, i32)** %3
%5 = call i32 %4(%class.B* %0, i32 42)

;load the b-pointer
;cast to vtable
;load vptr
;select f() entry
;load function pointer

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

- **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 `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

```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);
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.

```llvm
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 \( \Delta B \) 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:

A
int f(int)
int c
B
int f(int)
int b
C
int f(int)
int l
L
int f(int)
int l
L
int f(int)
int b
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 = (B*)&c;
pl->f(42); // where to dispatch?
C* pc = (C*)(B*)pl;

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

```
W
 int f(int)
 int g(int)
 int h(int)
 int w

virtual

A
 int f(int)
 int a

virtual

B
 int g(int)
 int b

virtual

C
 int h(int)
 int c
```
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->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 { 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

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

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

define void @__g(%class.B* %this, i32 %i) {
    ; virtual thunk to B::g
%1 = bitcast %class.B* %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 vcalls
%5 = bitcast i8* %4 to i64*
%6 = load i64* %5 ; load g's vcall offset
%7 = getelementptr i8* %1, i64 %6 ; navigate to vcalloffset+ Wtop
%8 = bitcast i8* %7 to %class.B*
call void @_g(%class.B* %8, i32 %i)
ret void
}
A Virtual Table for a Virtual Subclass

gets a \textit{virtual base pointer}

A Virtual Table for a Virtual Base

consists of different parts:

1. \textit{virtual call offsets} per virtual function for adjusting \texttt{this} dynamically

2. \textit{offset to top} of an enclosing objects heap representation

3. \textit{typeid pointer} to an RTTI object (not relevant for us)

4. \textit{virtual function pointers} for resolving virtual methods

Virtual Base classes have \textit{virtual thunks} which look up the offset to adjust the \texttt{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
- 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
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...

A monotonic superclass linearization for dylan.

CodeSourcery, Compaq, EDG, HP, IBM, Intel, R. Hat, and SGI.
Itanium C++ ABI.

R. Ducournau and M. Habib.
On some algorithms for multiple inheritance in object-oriented programming.

R. Kleckner.
Bringing clang and llvm to visual c++ users.

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.

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

R. C. Martin.
The liskov substitution principle.

P. Sabanal and M. Yason.
Reversing c++.
In *Black Hat DC*, 2007.

B. Stroustrup.
Multiple inheritance for C++.