Script generated by TTT

Title: Petter: Programmiersprachenh (05.12.2018)

Date: Wed Dec 05 14:11:14 CET 2018

Duration: 101:38 min

Pages: 28

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

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

"Wouldn't it be nice to inherit from several parents?"

Interface vs. Implementation inheritance

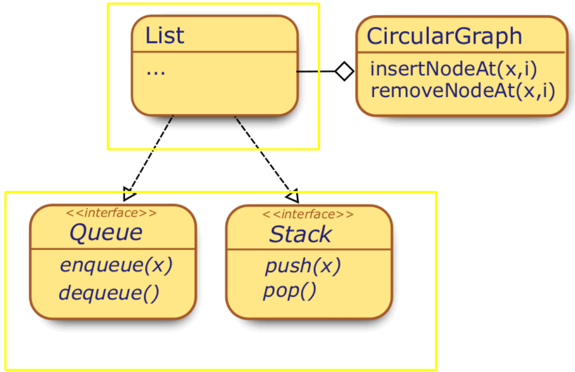
The classic motivation for inheritance is implementation inheritance

- *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

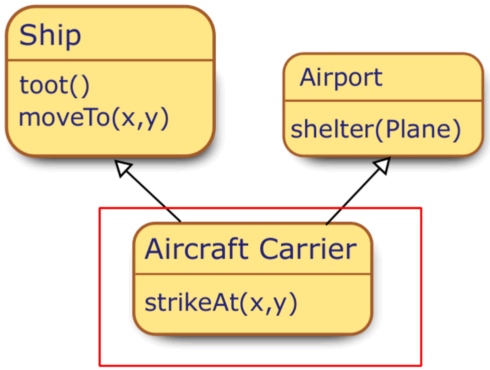
Code sharing in interface inheritance inverts this relation

- *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



Implementation inheritance



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:

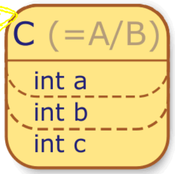
```
clang -O1 -S -emit-llvm source.cpp -o IR.llvm
```

Object layout

```
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);
```

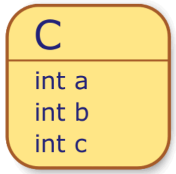



```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

```
%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 {
  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;
};
```




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

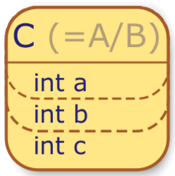
```
define i32 @_g(%class.B* %this, i32 %p) {
  %1 = getelementptr %class.B* %this, i64 0, i32 1
  %2 = load i32* %1
  %3 = add i32 %2, %p
  ret i32 %3
}
```

Object layout

```
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);
```




```
%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
```

"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)")
```


Single-Dispatching implementation choices

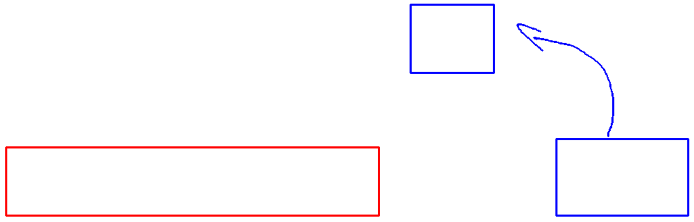
Single-Dispatching needs runtime action:

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

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

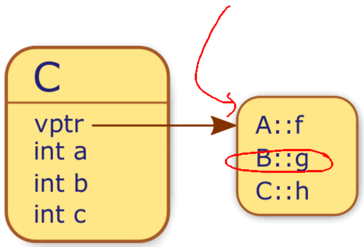

2. Caching the dispatch result (⇝ Hotspot/JIT)

```
; 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
```



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);
```

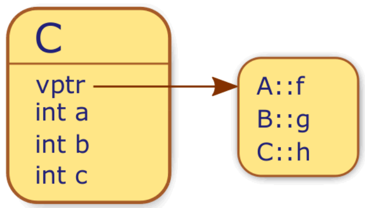



```
%class.C = type { %class.B, i32, [4 x i8] }
%class.B = type { [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
```

```
%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)
```


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);
```



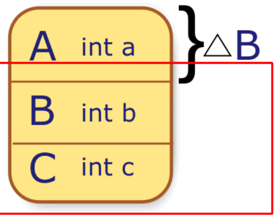

```
%class.C = type { %class.B, i32, [4 x i8] }
%class.B = type { [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
```

```
%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?"

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();
```

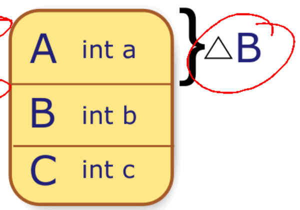



```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

```
%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();
```

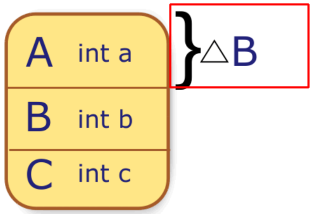



```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

```
%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*
```

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:

$C(A,B) \implies C \rightarrow\!\!\triangleright A \wedge C \rightarrow\!\!\triangleright B$

Principle 2: Multiplicity Relation

Defined by the succession of multiple parents. Example:

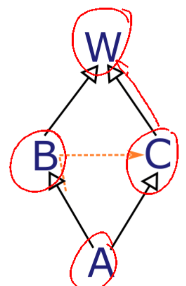
$C(A,B) \implies A \rightarrow B$

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



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

A B ~~W~~ C W

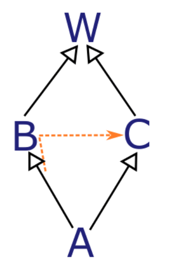
MRO via DFS

Leftmost Preorder Depth-First Search

$$L[A] = ABWC$$

⚠ Principle 1 *inheritance* is violated

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



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

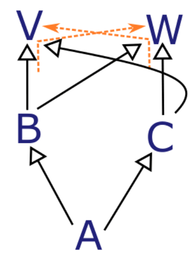
LPDFS with Duplicate Cancellation

$$L[A] = ABCW$$

✓ Principle 1 *inheritance* is fixed

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

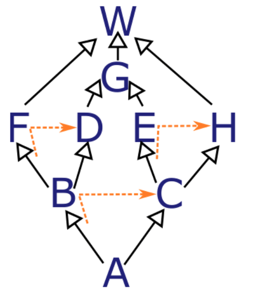
LPDFS with Duplicate Cancellation



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

MRO via Refined Postorder DFS

Reverse Postorder Rightmost DFS

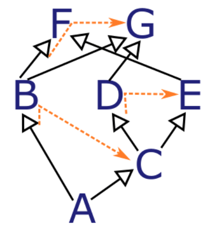


$A(B,C)\ B(F,D)\ C(E,H)$
$D(G)\ E(G)\ F(W)\ G(W)\ H(W)$

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 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
- *Solidity, Python 3* and *Perl 6* with C3
- Prerequisite for $\rightarrow$ Mixins

“And what about dynamic dispatching in Multiple Inheritance?”