Script generated by TTT

Title: Simon: Programmiersprachen (13.12.2013)

Date: Fri Dec 13 14:17:06 CET 2013

Duration: 83:10 min

Pages: 28

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Multiple Inheritance

Dr. Axel Simon and Dr. Michael Petter
Winter term 2013

Outline

Inheritance Principles
1. Interface Inheritance
2. Implementation Inheritance
3. Liskov Substition Principle and Shapes

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

Discussion & Learning Outcomes

Outline

Inheritance Principles
1. Interface Inheritance
2. Implementation Inheritance
3. Liskov Substition Principle and Shapes

C++ Object Heap Layout
1. Basics
2. Single-Inheritance
3. Virtual Methods

Excursion: Linearization
1. Ambiguous common parents
2. Principles of Linearization
3. Linearization algorithm

C++ Multiple Parents Heap Layout
1. Multiple-Inheritance
2. Virtual Methods
3. Common Parents

Discussion & Learning Outcomes

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

Interface vs. Implementation inheritance

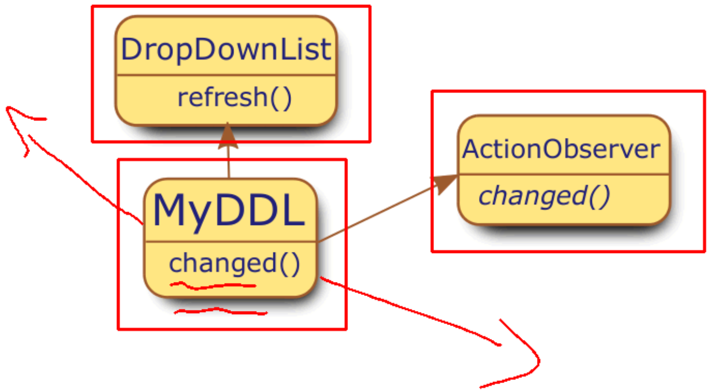
The classic motivation for inheritance is implementation inheritance

- *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

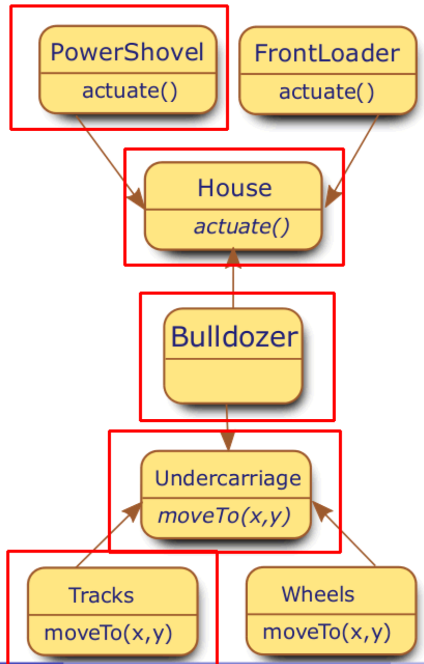
Code sharing in interface inheritance inverts this relation

- *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: LSP and Square-Rect-Problem

The Liskov Substitution Principle

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

```
class Rectangle {
  void setWidth (int w){ this.w=w; }
  void setHeight(int h){ this.h=h; }
  void getWidth ()     { return w; }
  void getHeight()     { return h; }
}
class Square extends Rectangle {
  void setWidth (int w){ this.w=w;h=w; }
  void setHeight(int h){ this.h=h;w=h; }
}
```

```
Rectangle r =
         new Square(2);
r.setWidth(3);
r.setHeight(4);
assert r.getHeight()*
         r.getWidth()==12;
```


Excursion: LSP and Square-Rect-Problem

The Liskov Substitution Principle

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

```
class Rectangle {
  void setWidth (int w){ this.w=w; }
  void setHeight(int h){ this.h=h; }
  void getWidth ()     { return w; }
  void getHeight()     { return h; }
}
class Square extends Rectangle {
  void setWidth (int w){ this.w=w;h=w; }
  void setHeight(int h){ this.h=h;w=h; }
}
```

```
Rectangle r =
         new Square(2);
r.setWidth(3);
r.setHeight(4);
assert r.getHeight()*
         r.getWidth()==12;
```


⚠ Behavioural assumptions

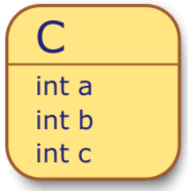
Excursion: Brief introduction to LLVM IR

Low Level Virtual Machine as reference semantics:

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

;allocation of objects
%a = alloca %struct.A

;adress adjustments for selection in structures:
%1 = getelementptr %struct.A* %a, i64 2

;load from memory
%2 = load i32(i32)* %1

;indirect call
%retval = call i32 (i32)* %2(i32 42)
```


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);
```

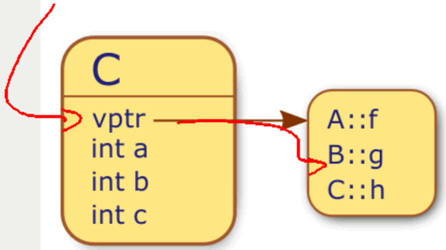



```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

```
%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
```

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);
```



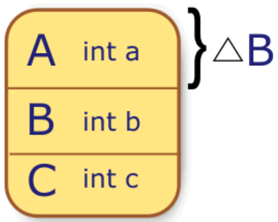

```
%class.C = type { %class.B, i32, [4 x i8] }
%class.B = type { [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
```

```
%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 Base Classes

```
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);
```



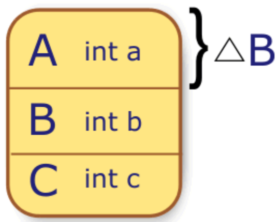

```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

```
%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* %4, i32 42)  ; g is statically known
```

%2

Multiple Base Classes

```
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* %4, i32 42)  ; g is statically known
```

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

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

Inheritance Relation H
Defined by ancestors.

Multiplicity M
Defined by the order of multiple ancestors.

Principles

1. An inheritance mechanism (maps Object to sequence of ancestors) must follow the inheritance partial order H
2. The inheritance is a uniform mechanism, and its searches ($\rightarrow$ total order) apply identically for all object properties ($\rightarrow$fields/methods)
3. In any case the inheritance relation H overrides the multiplicity M
4. When there is no contradiction between multiplicity M and inheritance H, the inheritance search must follow the partial order $H \cup M$.

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

Inheritance Relation H
Defined by ancestors.

Multiplicity M
Defined by the order of multiple ancestors.

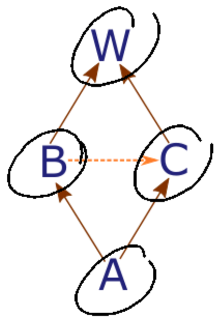
Principles

1. An inheritance mechanism (maps Object to sequence of ancestors) must follow the inheritance partial order H
2. The inheritance is a uniform mechanism, and its searches ($\rightarrow$ total order) apply identically for all object properties ($\rightarrow$fields/methods)
3. In any case the inheritance relation H overrides the multiplicity M
4. When there is no contradiction between multiplicity M and inheritance H, the inheritance search must follow the partial order $H \cup M$.

Linearization algorithm candidates

Depth-First Search

A B W C

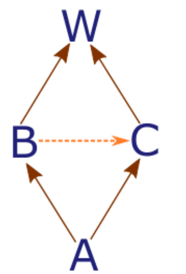


Linearization algorithm candidates

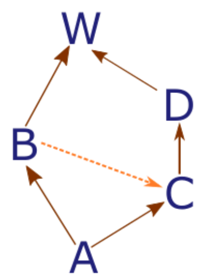
Depth-First Search

A B W C

⚠ Principle 1 *inheritance* is violated



Breadth-First Search

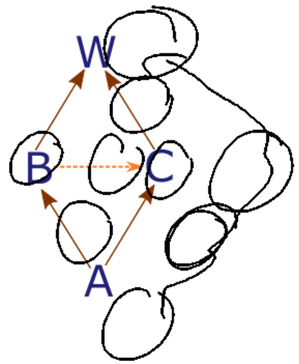


Linearization algorithm candidates

Depth-First Search

A B W C

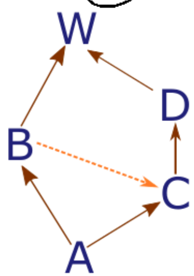
⚠ Principle 1 *inheritance* is violated



Breadth-First Search

A B C W D

⚠ Principle 1 *inheritance* is violated

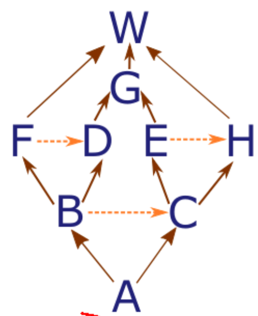


Linearization algorithm candidates

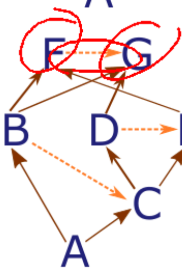
Reverse Postorder Rightmost DFS

A B F D C E G H W

✓ Linear extension of inheritance relation



Reverse Postorder Rightmost DFS

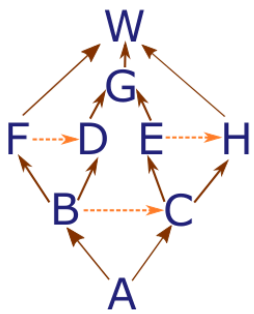


Linearization algorithm candidates

Reverse Postorder Rightmost DFS

A B F D C E G H W

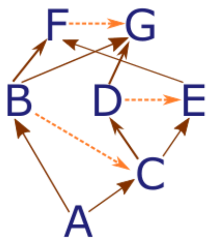
✓ Linear extension of inheritance relation



Reverse Postorder Rightmost DFS

A B C D G E F

⚠ But principle 4 *multiplicity* is violated!



Linearization Algorithm

Idea [Ducournau and Habib(1987)]

Successively perform Reverse Postorder Rightmost DFS and refine inheritance graph G with *contradiction arcs*.

The reservoir set of potential *contradiction arcs* CA is initially M, while the inheritance graph G starts from H.

do

1. $search \leftarrow$ RPDFS$_G$
2. $CA \leftarrow$ {contradiction arcs of upper search} $\cap M$
3. $G \leftarrow G \cup CA;$

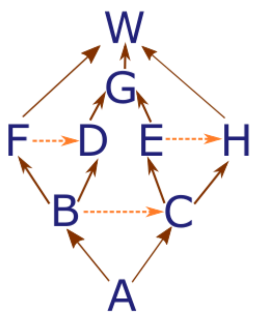
while $(CA \neq \emptyset) \wedge (search$ violates $H \cup M)$

Linearization algorithm candidates

Reverse Postorder Rightmost DFS

A B F D C E G H W

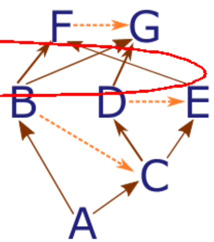
✓ Linear extension of inheritance relation



Reverse Postorder Rightmost DFS

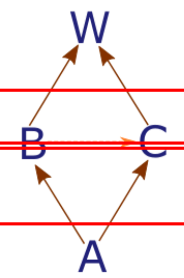
A B C D G E F

⚠ But principle 4 *multiplicity* is violated!



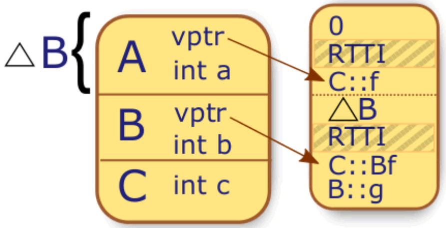
Linearization algorithm candidates

Depth-First Search



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
```