Script generated by TTT

Title: Petter: Programmiersprachen (16.12.2015)

Date: Wed Dec 16 14:24:29 CET 2015

Duration: 88:22 min

Pages: 24

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Multiple Inheritance

Dr. Michael Petter
Winter term 2015

Outline

Outline

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

Interface vs. Implementation inheritance

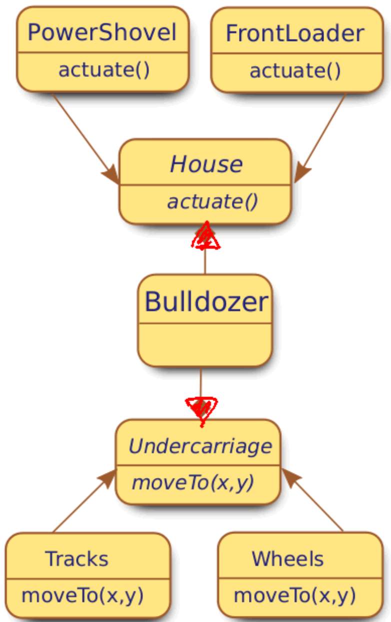
The classic motivation for inheritance is implementation inheritance

- *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

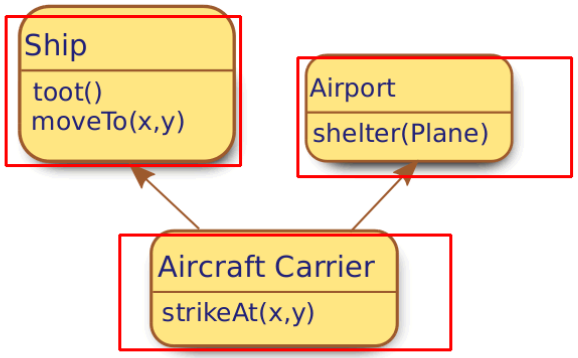
Code sharing in interface inheritance inverts this relation

- *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



"So how do we lay out objects in memory anyway?"

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 }

;(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:

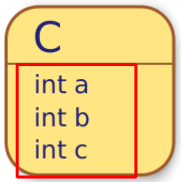
```
clang -O1 -S -emit-llvm source.cpp -o IR.llvm
```


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);
```

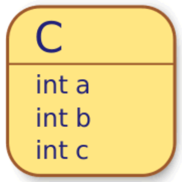



```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

```
%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.A* %this, i64 0, i32 1
  %2 = load i32* %1
  %3 = add i32 %2, %p
  ret i32 %3
}
```

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;
};
```

C
int a
int b
int c

```
%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);
```

C
int a
int b
int c

```
%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
```


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 4711)
```


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 4711)
```

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

```
%1 = getelementptr(%class.C* %c, i64 0, i32 0)
%2 = load i32* %1
assert (%2==%class.D)
call i32 @_f(%class.C* %c, i32 42)
```

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 4711)
```
2. Caching the dispatch result (⇝ Hotspot/JIT)
```
%1 = getelementptr(%class.C* %c, i64 0, i32 0)
%2 = load i32* %1
assert (%2==%class.D)
call i32 @_f(%class.C* %c, i32 42)
```
3. Precomputing the dispatching result in tables

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 4711)
```
2. Caching the dispatch result (⇝ Hotspot/JIT)
```
%1 = getelementptr(%class.C* %c, i64 0, i32 0)
%2 = load i32* %1
assert (%2==%class.D)
call i32 @_f(%class.C* %c, i32 42)
```
3. Precomputing the dispatching result in tables
   1. Full 2-dim matrix

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 4711)
```
2. Caching the dispatch result (⇝ Hotspot/JIT)
```
%1 = getelementptr(%class.C* %c, i64 0, i32 0)
%2 = load i32* %1
assert (%2==%class.D)
call i32 @_f(%class.C* %c, i32 42)
```
3. Precomputing the dispatching result in tables
   1. Full 2-dim matrix
   2. 1-dim Row Displacement Dispatch Tables

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

	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	

Single-Dispatching implementation choices

Single-Dispatching needs runtime action:

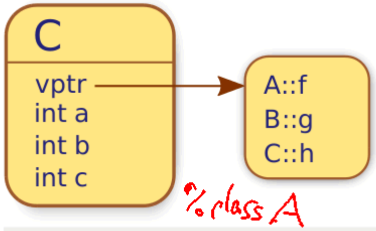
1. Manual search run through the super-chain (Java Interpreter ⇝ last talk)
```
call i32 @__dispatch(%class.C* %c,i32 4711)
```
2. Caching the dispatch result (⇝ Hotspot/JIT)
```
%1 = getelementptr(%class.C* %c, i64 0, i32 0)
%2 = load i32* %1
assert (%2==%class.D)
call i32 @_f(%class.C* %c, i32 42)
```
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)

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

	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	

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)
```


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 4711)
```

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

```
%1 = getelementptr(%class.C* %c, i64 0, i32 0)
%2 = load i32* %1
assert (%2==%class.D)
call i32 @_f(%class.C* %c, i32 42)
```

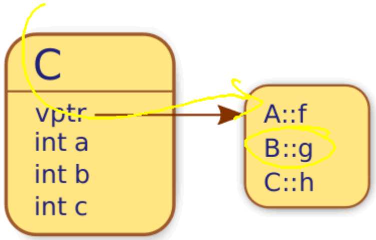
3. Precomputing the dispatching result in tables
   1. Full 2-dim matrix
   2. 1-dim Row Displacement Dispatch Tables
   3. Virtual Tables (⇝ LLVM/GNU C++,this talk)

	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 {
  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);
```



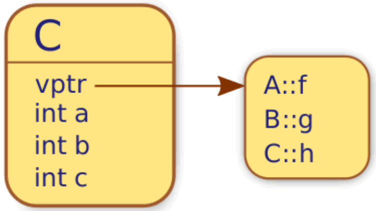

```
%class.C = type { %class.B, i32, [4 x i8] }
%class.B = type { [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
```

```
%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?"

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)
```