Script generated by TTT

Title: Simon: Programmiersprachen (20.12.2013)

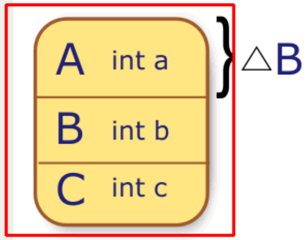
Date: Fri Dec 20 14:15:31 CET 2013

Duration: 66:53 min

Pages: 16

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* %2, 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

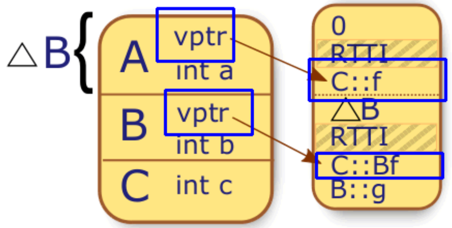
```
pc->A::f(42);
pc->B::f(42);
```

Solution II: Automagical resolution

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

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 }
```

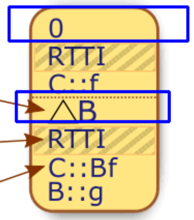
```
; pb->f(42);
%0 = load %class.B** %pb                              ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***  ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                   ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0   ;select f() entry
%4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

Virtual table

A Virtual Table

consists of different parts:

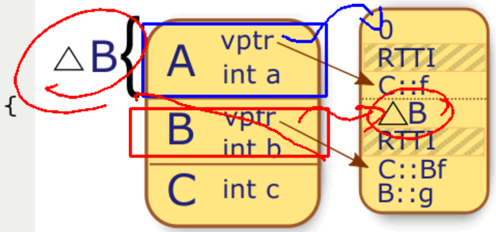
1. the constant offset of an objects heap representation to its parents heap representation
2. a pointer to a runtime type information object (not relevant for us)
3. method pointers of the overwritten methods for resolving virtual methods



- Several virtual tables are joined when multiple inheritance is used ⇝ Casts!
- The `vptr` field in each object points at the beginning of the first virtual method pointer
- `clang -cc1 -fdump-vtable-layouts -emit-llvm code.cpp` yields the vtables of a compilation unit

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 }
```

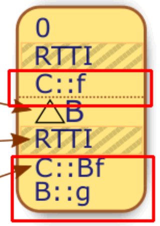
```
; pb->f(42);
%0 = load %class.B** %pb                                  ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***      ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                       ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0       ;select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


Virtual table

A Virtual Table

consists of different parts:

1. the constant offset of an objects heap representation to its parents heap representation
2. a pointer to a runtime type information object (not relevant for us)
3. method pointers of the overwritten methods for resolving virtual methods



- Several virtual tables are joined when multiple inheritance is used ⇝ Casts!
- The `vptr` field in each object points at the beginning of the first virtual method pointer
- `clang -cc1 -fdump-vtable-layouts -emit-llvm code.cpp` yields the vtables of a compilation unit

Virtual table 2

Remarks:

- The virtual table is created at compile time and filled with offsets, virtual method pointers and thunks
- ΔB is the relative position of the `B` part in `C`, and known at compile time. This entry is primarily used for dynamic casts:

```
C c;
B* b = &c;
void* v = dynamic_cast<void*>(b);
printf("%d, %d, %d",&c,b,v);
```

Virtual table 3

If a B-casted C-Object calls f(int), we have to dispatch to the overwritten method C::f(int). However, C::f(int) might access fields from A, but is provided with a pointer to the B-Object-Part of this.

Solution: *thunks*

. . . are trampoline methods, delegating the virtual method to its original implementation with an adapted this-reference

```
C c;
B* pb=&c;
pb->f(42); /* f(int) provided by C::f(int),
              addressing its variables relative to C */
```

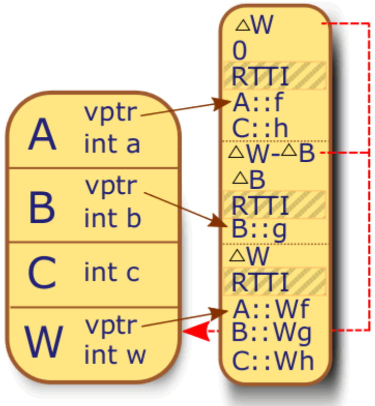
⇝ B-in-C-vtable entry for f(int) is the thunk _f(int), adding ΔB to this:

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


"But what if there are common ancestors?"

Common base classes

```
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->f(42);
((W*)pc)->h(42);
((A*)pc)->f(42);
```



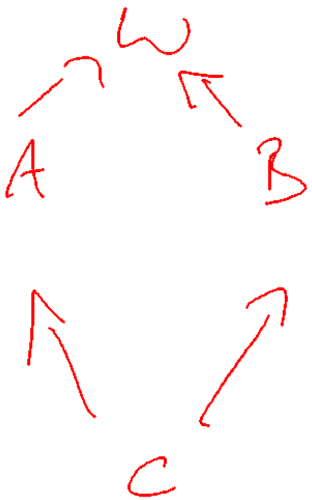
⚠ Offsets to virtual base

⚠ Ambiguities

⇝ e.g. overwriting f in A *and* B

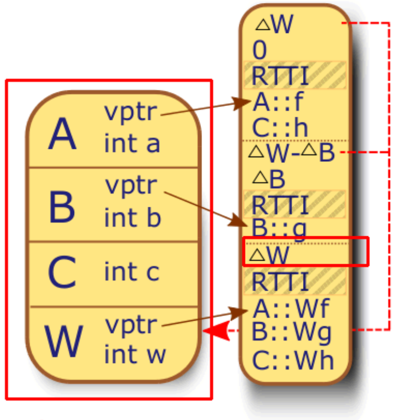
⚠ Casting!

```
W* pw = &c;
C* pc = (C*)pw;
```

Common base classes

```
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->f(42);
((W*)pc)->h(42);
((A*)pc)->f(42);
```



- ⚠ Offsets to virtual base
- ⚠ Ambiguities
  ⇝ e.g. overwriting f in A *and* B
- ⚠ Casting!

```
W* pw = &c;
C* pc = (C*)pw;
```


Compiler and Runtime Collaboration

Compile time:

1. Compiler generates one code block for each method per class
2. Compiler generates one virtual table for each class, with
   - references to the most recent implementations of methods of a *unique common signature*
   - static offsets of top and virtual bases
3. Each virtual table may be *composed from customized virtual tables* of parents (⇝ thunks)
4. If needed, compiler generates thunks to adjust the `this` parameter of methods

Runtime:

1. Calls to constructors allocate memory space
2. Constructor stores pointers to virtual table (or fragments) respectively
3. Method calls transparently call methods statically or from virtual tables, unaware of real class identity
4. Dynamic casts may use top pointer

Polemics of Multiple Inheritance

Full Multiple Inheritance (FMI)

- Most powerful inheritance principle known
- More convenient and simple in the common cases
- Occurance of diamond problem not as frequent as discussions indicate

Multiple Interface Inheritance (MII)

- MII not as complex as FMI
- MII together with aggregation expresses most practical problems
- Killer example for FMI yet to be presented
- 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

Further reading...

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

Roland Ducournau and Michel Habib.
On some algorithms for multiple inheritance in object-oriented programming.
In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1987.

Barbara 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 Language Reference Manual.
Llvm project.
URL: `http://llvm.org/docs/LangRef.html`.

Robert C. Martin.
The liskov substitution principle.
In *C++ Report*, 1996.

Bjarne Stroustrup.
Multiple inheritance for C++.
In *Computing Systems*, 1999.