Script generated by TTT

Title: Seidl: Virtual Machines (17.06.2014)

Date: Tue Jun 17 10:21:52 CEST 2014

Duration: 53:50 min

Pages: 23

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)
```

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:

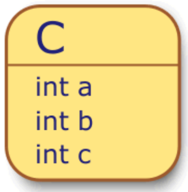
```
clang -O1 -S -emit-llvm source.cpp -o IR.llvm
```


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);
```

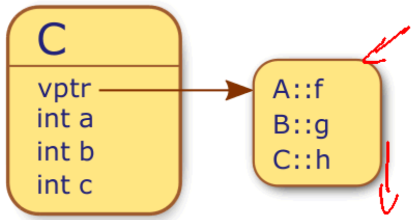



```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

```
%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
```


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);
```



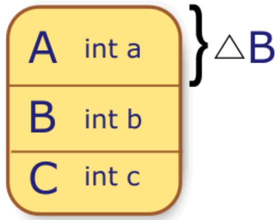

```
%class.C = type { %class.B, i32, [4 x i8] }
%class.B = type { [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
```

```
%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);
```



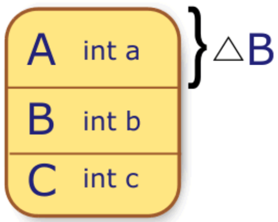

```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

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

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

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 · i8!

Ambiguities

```
class A { void f(int); };
class B { void f(int); };
class C : public A, public B {};

C* pc = new C();
pc->f(42);
```

⚠ Which method is called?

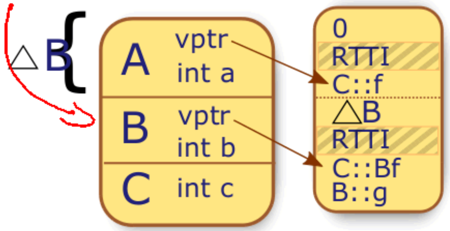
Solution I: Explicit qualification

```
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);
```

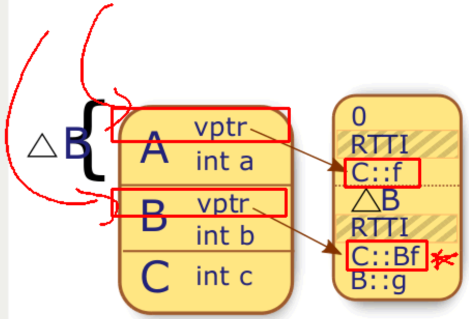



```
%class.C = type { %class.A, [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
%class.B = type { i32 (...)**, i32 }
```

```
; 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);
```




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

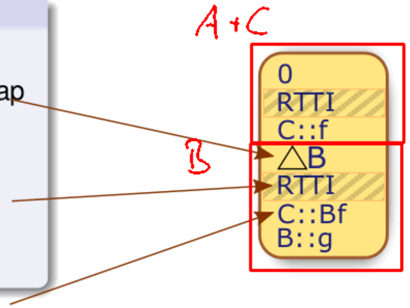
```
; pb->f(42);
%0 = load %class.B** %pb                                  ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***      ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                       ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0       ;select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


Basic Virtual tables (⇝ C++-ABI)

A Basic Virtual Table

consists of different parts:

1. *offset to top* of an enclosing objects heap representation
2. *typeinfo pointer* to an RTTI object (not relevant for us)
3. *virtual function pointers* 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

Thunks

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 */
```

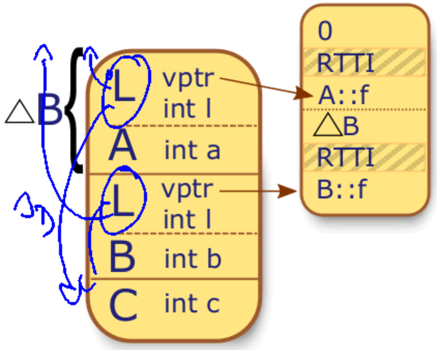
⇝ `B`-in-`C`-vtable entry for `f(int)` is the thunk `_f(int)`, adding ΔB to `this`:

```
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?"

Distinguished 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 = &c;
pl->f(42);
C* pc = (C*)pl;
```



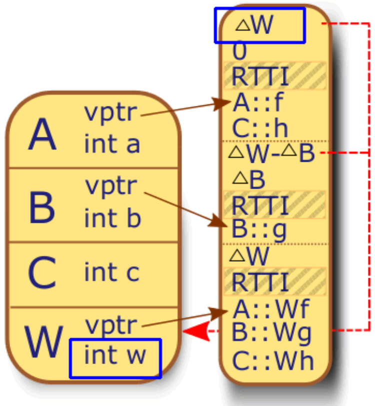

```
%class.C = type { %class.A, %class.B,
                  i32, [4 x i8] }
%class.A = type { [12 x i8], i32 }
%class.B = type { [12 x i8], i32 }
%class.L = type { i32 (...)**, i32 }
```

⚠ Ambiguity!

```
L* pl = (A*)&c;
C* pc = (C*)(A*)pl;
```


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



⚠ Offsets to virtual base

⚠ Ambiguities

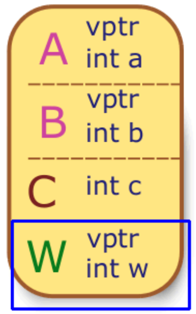
⇝ e.g. overwriting f in A *and* B

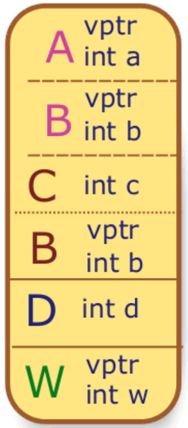
Dynamic vs. Static Casting

```
class D : public C,
          public B {
...
};
class A : public virtual W {
...
};
class B : public virtual W {
...
};
class C : public A , public B {
...
};
...
C c;
W* pw = &c;
C* pc = (C*)pw; // Compile error
```

vs.

```
C* pc = dynamic_cast<C*>(pw);
```



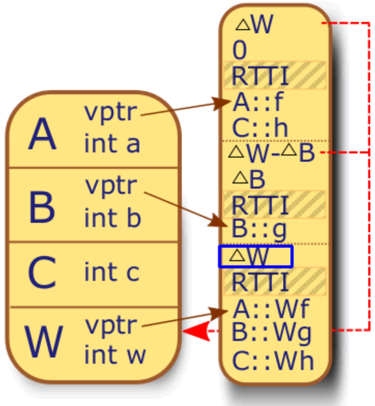


⚠ No guaranteed *constant* offsets between virtual bases and subclasses ⇝ No static casting!

⚠ *Dynamic casting* makes use of *offset-to-top*

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



⚠ Offsets to virtual base

⚠ Ambiguities

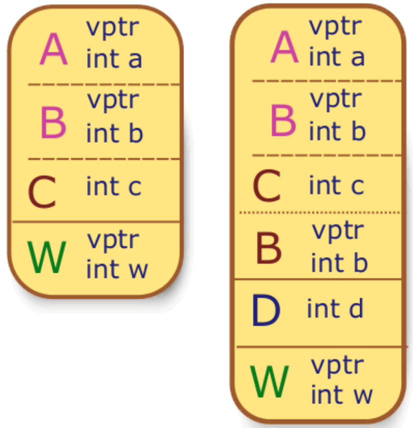
⇝ e.g. overwriting f in A *and* B

Dynamic vs. Static Casting

```
class D : public C,
          public B {
...
};
class A : public virtual W {
...
};
class B : public virtual W {
...
};
class C : public A , public B {
...
};
...
C c;
W* pw = &c;
C* pc = (C*)pw; // Compile error
```

vs.

```
C* pc = dynamic_cast<C*>(pw);
```

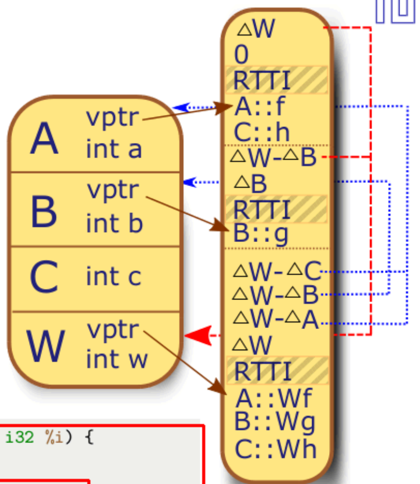


⚠ No guaranteed *constant* offsets between virtual bases and subclasses ⇝ No static casting!

⚠ *Dynamic casting* makes use of *offset-to-top*

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 @virtualThunkTo_B::g(%class.B* %this, i32 %i) {
  %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 @B::g(%class.B* %8, i32 %i)
  ret void
}
```


Virtual Tables for Virtual Bases (⇝ C++-ABI)

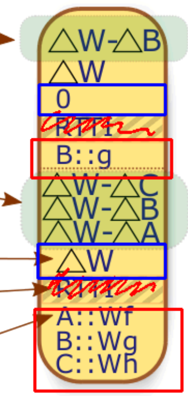
A Virtual Table for a Virtual Subclass

gets a *virtual base pointer*

A Virtual Table for a Virtual Base

consists of different parts:

1. *virtual call offsets* per virtual function for adjusting `this` dynamically
2. *offset to top* of an enclosing objects heap representation
3. *typeinfo pointer* to an RTTI object (not relevant for us)
4. *virtual function pointers* for resolving virtual methods



Virtual Base classes have *virtual thunks* which look up the offset to adjust the `this` pointer to the correct value in the virtual table!

Sidenote

Microsoft's MSVC++ implements a different memory model for the OO-features. Their compiler splits the virtual table into several smaller tables. It also keeps a vbptr (virtual base pointer) in the object representation, redirecting to the virtual base of a subclass.

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

Sidenote

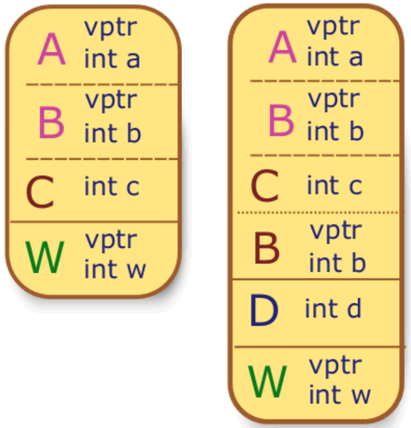
Microsoft's MSVC++ implements a different memory model for the OO-features. Their compiler splits the virtual table into several smaller tables. It also keeps a vbptr (virtual base pointer) in the object representation, redirecting to the virtual base of a subclass.

Dynamic vs. Static Casting

```
class D : public C,
          public B {
...
};
class A : public virtual W {
...
};
class B : public virtual W {
...
};
class C : public A , public B {
...
};
...
C c;
W* pw = &c;
C* pc = (C*)pw; // Compile error
```

vs.

```
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*