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
;address 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 -01 -S -emit-llvm source.cpp -o IR.llvm`

Object layout – virtual methods

```
class A {
  int a; virtual 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.A = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.C = type { i32 }

% = alloca %class.C
%1 = bitcast %class.C* %c to %class.B*
%2 = call i32 i32(%%%class.B%1, i32 42) % g is statically known
```

```
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.vptr = bitcast %class.C* %c to i32 (%class.B*, i32)**; vtbl
%1 = load (%class.B*, i32)** %C.vptr; dereference vptr
%2 = getelementptr i1, i64 1
%3 = load (%class.B*, i32)** %2; dereference g()-entry
%4 = call i32 i32(%%%class.B%3, 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
%l = bitcast %class.C* %c to i8*
%l2 = getelementptr i8* %l, i64 4
%l3 = call i32 @__cgsym(%class.B* %l2, i32 42) ; g is statically known

△ getelementptr implements ∆B as 4 · i8!

Multiple Inheritance
Implementation of Multiple Inheritance

Multiple Base Classes in layout 7/21

Ambiguities

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);

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; virtual int f(int);
    ...
    C c;
    B* pb = &c;
pb->f(42);

Basic Virtual tables (∼ C++-ABI)

A Basic Virtual Table

consists of different parts:

- offset to top of an enclosing objects heap representation
- typeinfo pointer to an RTTI object (not relevant for us)
- 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

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:

```cpp
define i32 @_f(%class.B* %this, i32 %1) {
    %1 = bitcast %class.B* to i8* %this to i8*;
    %2 = getelementptr i8*, i16, 164 0 ; sizeof(B)=16
    %3 = bitcast i8* %2 to %class.C*;
    %4 = call i32 @_f(%class.C* %3, i32 %1)
    ret i32 %4
}
```
“But what if there are common ancestors?”

**Common base classes**

```cpp
class W {
    int w; virtual void f(int);
    virtual void g(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);
```

**Distinguished base classes**

```cpp
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;
```

**Dynamic vs. Static Casting**

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

**Offsets to virtual base**

Ambiguities
~ e.g. overwriting f in A and B

**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);

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);

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);

// Virtual Table for a Virtual Subclass

define void @virtualThunk_W_RTTI(%.class.RT, %this, i132 %i) {
    %1 = bitcast %.class.RT to i132
    %2 = bitcast %1 to i16
    %3 = load i16* %2 ; load W-rtti ptr
    %4 = getelementptr i16, %1, 164 -32 ; -32 bytes is g-entry in vcall
    %5 = bitcast %4 to i16
    %6 = load 164* %5 ; load g's vcall offset
    %7 = getelementptr i16 , %1, 164 %6 ; navigate to vcalloffset Wtop
    %8 = bitcast %7 to %.base.B *
    call void @virtualThunk_W_RTTI(%.base.B *, %132 %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:

- virtual call offsets per virtual function for adjusting this dynamically
- offset to top of an enclosing objects heap representation
- typeinfo pointer to an RTTI object (not relevant for us)
- 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

- Different purposes of inheritance
- Heap Layouts of hierarchically constructed objects in C++
- Virtual Table layout
- LLVM IR representation of object access code
- Linearization as alternative to explicit disambiguation
- Pitfalls of Multiple Inheritance

Dynamic vs. Static Casting

```c
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