Script generated by TTT

Title: Petter: Programmiersprachen (18.12.2019)

Date: Wed Dec 18 12:20:14 CET 2019

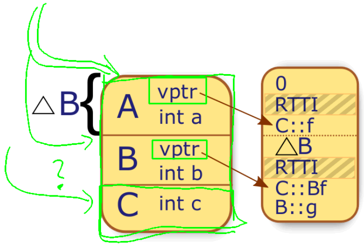
Duration: 76:16 min

Pages: 28

"And what about dynamic dispatching in Multiple Inheritance?"

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 }
```

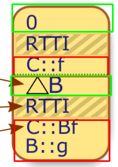
```
; pb->f(42);
%0 = load %class.B** %pb                                  ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***      ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                       ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0       ;select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

Basic Virtual Tables (⇝ C++-ABI)

A Basic Virtual Table

consists of different parts:

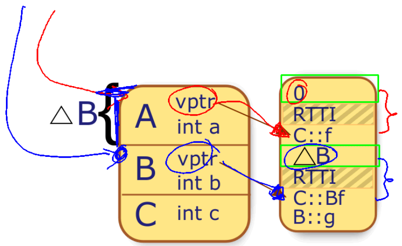
1. *offset to top* of an enclosing objects memory representation
2. *typeinfo pointer* to an RTTI object (not relevant for us)
3. *virtual function pointers* for resolving virtual methods



- Virtual tables are composed when multiple inheritance is used
- The `vptr` fields in objects are pointers to their corresponding virtual-subtables
- Casting preserves the link between an object and its corresponding virtual-subtable
- `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 }
```

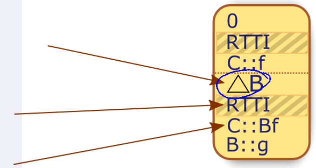
```
; pb->f(42);
%0 = load %class.B** %pb                                ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***    ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                     ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0     ;select f() entry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

Basic Virtual Tables (⇝ C++-ABI)

A Basic Virtual Table

consists of different parts:

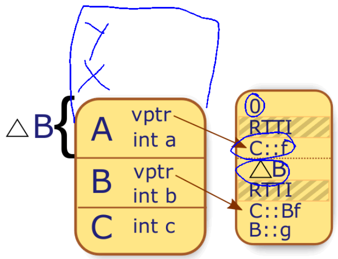
1. *offset to top* of an enclosing objects memory representation
2. *typeinfo pointer* to an RTTI object (not relevant for us)
3. *virtual function pointers* for resolving virtual methods



- Virtual tables are composed when multiple inheritance is used
- The `vptr` fields in objects are pointers to their corresponding virtual-subtables
- Casting preserves the link between an object and its corresponding virtual-subtable
- `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 }
```

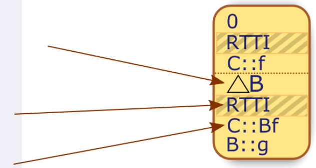
```
; pb->f(42);
%0 = load %class.B** %pb                                ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***    ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                     ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0     ;select f() entry
%4 = load i32(%class.B*, i32)** %3                      ;load function pointer
%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 memory representation
2. *typeinfo pointer* to an RTTI object (not relevant for us)
3. *virtual function pointers* for resolving virtual methods

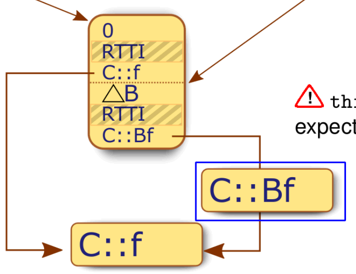


- Virtual tables are composed when multiple inheritance is used
- The `vptr` fields in objects are pointers to their corresponding virtual-subtables
- Casting preserves the link between an object and its corresponding virtual-subtable
- `clang -cc1 -fdump-vtable-layouts -emit-llvm code.cpp` yields the vtables of a compilation unit

Casting Issues

```
class A { int a; };
class B { virtual int f(int); };
class C : public A , public B {
  int c; int f(int);
};
C* c = new C();
c->f(42);
```

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



⚠ `this`-Pointer for `C::f` is expected to point to `C`

Thunks

Solution: thunks

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

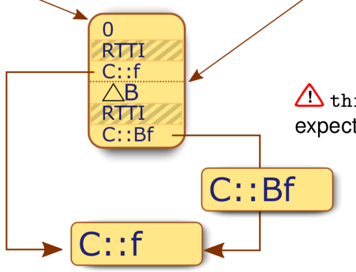
```
define i32 @__f(%class.B* %this, i32 %i) {
  %1 = bitcast %class.B* %this to i8*
  %2 = getelementptr i8* %1, i64 -16        ; sizeof(A)=16
  %3 = bitcast i8* %2 to %class.C*
  %4 = call i32 @_f(%class.C* %3, i32 %i)
  ret i32 %4
}
```

⇝ `B`-in-`C`-vtable entry for `f(int)` is the thunk `_f(int)`

Casting Issues

```
class A { int a; };
class B { virtual int f(int); };
class C : public A , public B {
  int c; int f(int);
};
C* c = new C();
c->f(42);
```

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

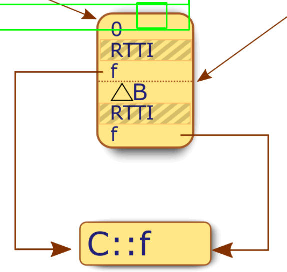


⚠ `this`-Pointer for `C::f` is expected to point to `C`

Casting Issues

```
class A { int a; };
class B { virtual int f(int); };
class C : public A , public B {
  int c; int f(int);
};
C* c = new C();
c->f(42);
```

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

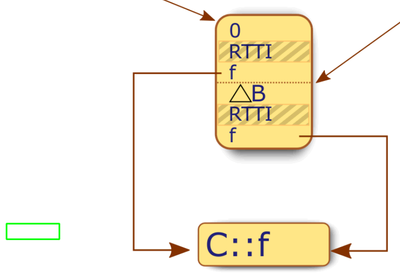


⚠ `this`-Pointer for `C::f` is expected to point to `C`

Casting Issues

```
class A { int a; };
class B { virtual int f(int); };
class C : public A , public B {
  int c; int f(int);
};
C* c = new C();
c->f(42);
```

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



⚠ `this`-Pointer for `C::f` is expected to point to `C`

Thunks

Solution: thunks

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

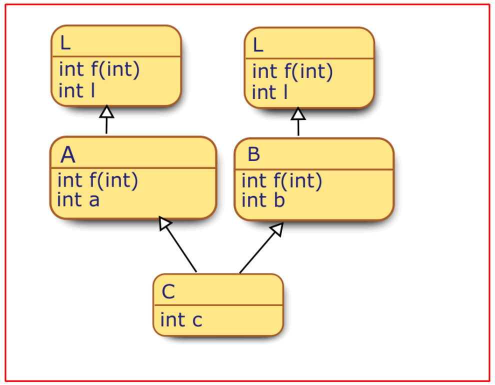
```
define i32 @__f(%class.B* %this, i32 %i) {
  %1 = bitcast %class.B* %this to i8*
  %2 = getelementptr i8* %1, i64 -16        ; sizeof(A)=16
  %3 = bitcast i8* %2 to %class.C*
  %4 = call i32 @_f(%class.C* %3, i32 %i)
  ret i32 %4
}
```

- ⇝ `B`-in-`C`-vtable entry for `f(int)` is the thunk `_f(int)`
- ⇝ `_f(int)` adds a compiletime constant ΔB to `this` before calling `f(int)`
- ⇝ `f(int)` addresses its locals relative to what it assumes to be a `C` pointer

"But what if there are common ancestors?"

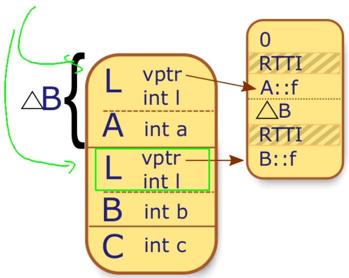
Common Bases – Duplicated Bases

Standard C++ multiple inheritance conceptually duplicates representations for common ancestors:



Duplicated Base Classes

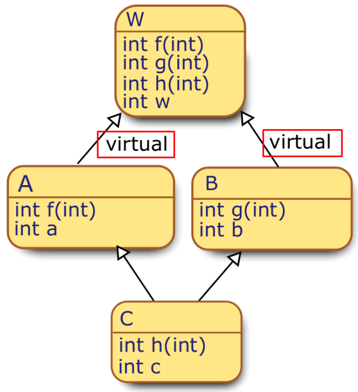
```
class L {
  int l; virtual void f(int);
};
class A : public L {
  int a; void f(int);
};
class B : public L {
  int b; void f(int);
};
class C : public A , public B {
  int c;
};
...
C c;
L* pl = (B*)&c;
pl->f(42); // where to dispatch?
C* pc = (C*)(B*)pl;
```




```
%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 }
```

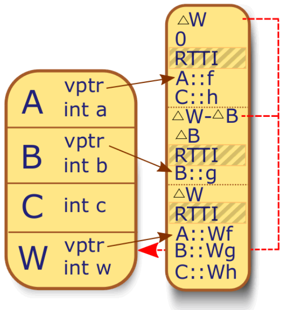
Common Bases – Shared Base Class

Optionally, C++ multiple inheritance enables a shared representation for common ancestors, creating the *diamond pattern*:



Shared Base Class

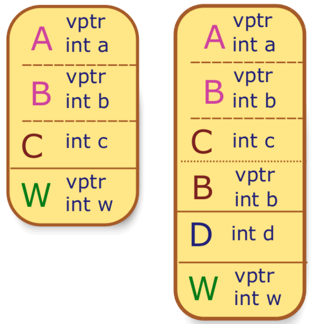
```
class W {
  int w; virtual void f(int);
  virtual void g(int);
  virtual void h(int);
};
class A : public virtual W {
  int a; void f(int);
};
class B : public virtual W {
  int b; void g(int);
};
class C : public A, public B {
  int c; void h(int);
};
...
C* pc;
pc->f(42);
```



⚠ Ambiguities
⇝ e.g. overriding f in A *and* B

Dynamic Type Casts

```
class A : public virtual W {
...
};
class B : public virtual W {
...
};
class C : public A , public B {
...
};
class D : public C,
          public B {
...
};
...
C c;
W* pw = &c;
C* pc = dynamic_cast<C*>(pw);
```

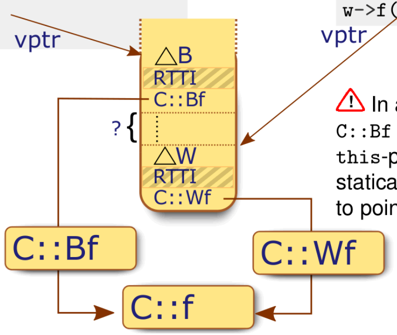


⚠ No guaranteed *constant* offsets between virtual bases and subclasses ⇝ No static casting!
⚠ *Dynamic casting* makes use of *offset-to-top*

Again: Casting Issues

```
class W { virtual int f(int); };
class A : virtual W { int a; };
class B : virtual W { int b; };
class C : public A , public B {
  int c; int f(int);
};
B* b = new C();
b->f(42);
```

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

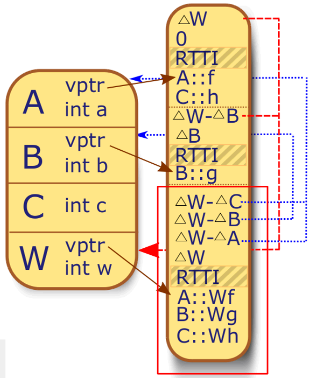


⚠ In a conventional thunk `C::Bf` adjusts the `this`-pointer with a statically known constant to point to `C`

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);
```

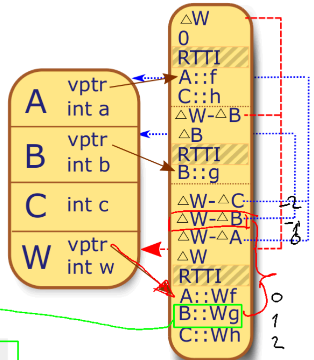
```
define void @__g(%class.B* %this, i32 %i) { ; virtual thunk to B::g
  %1 = bitcast %class.B* %this to i8*
  %2 = bitcast i8* %1 to i8**
  %3 = load i8** %2                          ; load W-vtable ptr
  %4 = getelementptr i8* %3, i64 -32 ; -32 bytes is g-entry in vcalls
  %5 = bitcast i8* %4 to i64*
  %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 @_g(%class.B* %8, i32 %i)
  ret void
}
```



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 @__g(%class.B* %this, i32 %i) { ; virtual thunk to B::g
  %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 @_g(%class.B* %8, i32 %i)
  ret void
}
```

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

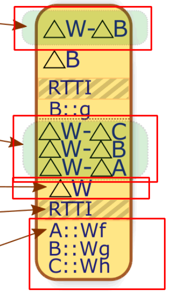
A Virtual Table for a Virtual Subclass

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!

Compiler and Runtime Collaboration

Compiler generates:

1. ... one code block for each method
2. ... one virtual table for each class-composition, with
   - references to the most recent implementations of methods of a *unique common signature* (⇝ single dispatching)
   - sub-tables for the composed subclasses
   - static top-of-object and virtual bases offsets per sub-table
   - (virtual) thunks as `this`-adapters per method and subclass if needed

Runtime:

1. At program startup virtual tables are globally created
2. Allocation of memory space for each object followed by constructor calls
3. Constructor stores pointers to virtual table (or fragments) in the objects
4. Method calls transparently call methods statically or from virtual tables, *unaware of real class identity*
5. Dynamic casts may use *offset-to-top* field in objects

Polemics of Multiple Inheritance

Full Multiple Inheritance (FMI)

- Removes constraints on parents in inheritance
- More convenient and simple in the common cases
- Occurance of diamond pattern not as frequent as discussions indicate

Multiple Interface Inheritance (MII)

- simpler implementation
- Interfaces and aggregation already quite expressive
- 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

Sidenote for MS VC++

- the presented approach is implemented in GNU C++ and LLVM
- Microsoft's MS VC++ approaches multiple inheritance differently
  - splits the virtual table into several smaller tables
  - keeps a vbptr (virtual base pointer) in the object representation, pointing to the virtual base of a subclass.

Further reading...

K. Barrett, B. Cassels, P. Haahr, D. Moon, K. Playford, and T. Withington.
A monotonic superclass linearization for dylan.
In *Object Oriented Programming Systems, Languages, and Applications*, 1996.

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

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

R. Kleckner.
Bringing clang and llvm to visual c++ users.
URL: `http://llvm.org/devmtg/2013-11/#talk11`.

B. 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.

L. L. R. Manual.
Llvm project.
URL: `http://llvm.org/docs/LangRef.html`.

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

P. Sabanal and M. Yason.
Reversing c++.
In *Black Hat DC*, 2007.
URL: `https://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf`.

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