Basic Virtual Tables (\textarrow C++-ABI)

A Basic Virtual Table consists of different parts:

- \textit{offset to top} of an enclosing objects heap representation
- \textit{typeinfo pointer} to an RTTI object (not relevant for us)
- \textit{virtual function pointers} for resolving virtual methods

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

\texttt{clang -cc1 -fdump-vtable-layouts -emit-llvm code.cpp} yields the vtables of a compilation unit

Casting Issues

\begin{verbatim}
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);
\end{verbatim}

\begin{verbatim}
B* b = new C();
b->f(42);
\end{verbatim}

Thunks

Solution: \textit{thunks}

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

\begin{verbatim}
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
}
\end{verbatim}

\textarrow B-in-C-vtable entry for f(int) is the thunk \_f(int)
**Solution: thunks**

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

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

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

→ _f(int) adds the statically constant ΔB to this before the call to f(int)

**Common Bases – Duplicated Bases**

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

```
A
  int f(int) int a

B
  int f(int) int b

C
  int c

L
  int f(int) int l
```
Common Bases – Duplicated Bases

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

\[
\begin{align*}
L & : \text{int } f(\text{int}) \text{ int } l \\
A & : \text{int } f(\text{int}) \text{ int } a \\
B & : \text{int } f(\text{int}) \text{ int } b \\
C & : \text{int } c
\end{align*}
\]

Duplicated Base Classes

\[
\text{class } L \\
\{ \\
\text{int } l; \text{virtual void } f(\text{int}); \\
\}; \\
\text{class } A : \text{public } L \\
\{ \\
\text{int } a; \text{void } f(\text{int}); \\
\}; \\
\text{class } B : \text{public } L \\
\{ \\
\text{int } b; \text{void } f(\text{int}); \\
\}; \\
\text{class } C : \text{public } A, \text{public } B \\
\{ \\
\text{int } c; \\
\}; \\
\ldots \\
\text{C } c; \\
L* \text{ pl } = \&c; \\
\text{pl-}f(42); \\
C* \text{ pc } = (C*)\text{pl};
\]

Common Bases – Shared Base Class

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

\[
\begin{align*}
W & : \text{int } f(\text{int}) \text{ int } g(\text{int}) \text{ int } h(\text{int}) \text{ int } w \\
A & : \text{virtual int } f(\text{int}) \\
B & : \text{virtual int } g(\text{int}) \\
C & : \text{virtual int } h(\text{int}) \\
\text{virtual int } h(\text{int}) \\
\text{virtual int } w
\end{align*}
\]

Shared Base Class

\[
\text{class } W \\
\{ \\
\text{int } w; \text{virtual void } f(\text{int}); \\
\text{virtual void } g(\text{int}); \\
\text{virtual void } h(\text{int}); \\
\}; \\
\text{class } A : \text{public virtual } W \\
\{ \\
\text{int } a; \text{void } f(\text{int}); \\
\}; \\
\text{class } B : \text{public virtual } W \\
\{ \\
\text{int } b; \text{void } g(\text{int}); \\
\}; \\
\text{class } C : \text{public } A, \text{public } B \\
\{ \\
\text{int } c; \text{void } h(\text{int}); \\
\}; \\
\ldots \\
C* \text{ pc}; \\
\text{pc-}f(42); \\
((W*)\text{pc})\rightarrow h(42); \\
((A*)\text{pc})\rightarrow f(42);
\]

\[\text{Offsets to virtual base} \]
\[\text{Ambiguities} \]
\[\text{e.g. overwriting } f \text{ in } A \text{ and } B\]
Dynamic Type Casts

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

Shared Base Class

```cpp
class W {
  int w;
  virtual void f(int); virtual void g(int); virtual void h(int);
};
class A : public virtual W {
  int a; virtual 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)->g(42);
((A*)pc)->f(42);
```

Dynamic Type Casts

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

Again: Casting Issues

```cpp
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);
```
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(w){class E* this, 132 Xi) {
    virtual thunk to B: g
    %1 = bitcast class.B* to i8*
    %2 = bitcast i8* Xi to i8*
    %3 = load i8** %2 ; load V-table ptr
    %4 = getelementptr i8* %3, 164  %8 ; -32 bytes is g-entry in vcalls
    %5 = bitcast i8* %4 to i16*
    %6 = load i16* %5 ; load g's vcall offset
    %7 = getelementptr i8* %6, 164  %8 ; navigate to vcalloffset* Vtop
    %8 = bitcast i8* %7 to class.E*
    call void@g(class.E* %8, 132 Xi)
    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. typeid 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
- Occurrence 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

- 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

Further reading...


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.