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
%3 = call i32 @g(%class.B* %2, i32 42)

g@t1 implements @t2 as 4:i8!

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);

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

%class.C = type { %class.A, [i2 x i8], i32 }
%class.A = type { i32 (...)*, i32 }
%class.B = type { i32 (...)*, i32 }
%class.B* = type { i32 }
%class.B->* = type { i32 }
%call = call i32 i32*%class.B* , i32 42

; load the b-pointer
X0 = load %class.B* %pb
X1 = bitcast %class.B* %0 to i32 (%class.B* , i32)***
X2 = load i32(%class.B* , i32)*** %1
X3 = getelementptr i32 (%class.B* , i32)*** %X2 , i64 0
X4 = load i32(%class.B* , i32)*** %X3
X5 = call i32 i32*%class.B* , i32 42

; cast to vtable
; load vptr

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?
Virtual Table

A Virtual Table consists of different parts:
- the constant offset of an object's heap representation to its parent's heap representation
- a pointer to a runtime type information object (not relevant for us)
- 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);
};

Virtual Table 2

Remarks:
- The virtual table is created at compile time and filled with offsets, virtual method pointers and thunks
- △B 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 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*p 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 ∆B 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
}

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);

"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);
};

A

\textit{Offsets to virtual base}

\textit{Ambiguities}

\textit{\textasciitilde{} e.g. overwriting f in A and B}

\textit{Casting!}

\[ A\cdot pc = (A\cdot w) \]

\[ A\cdot pc = (A\cdot w) \]

Compiler and Runtime Collaboration

\textbf{Compile time:}

\begin{itemize}
    \item Compiler generates one code block for each method per class
    \item Compiler generates one virtual table for each class, with
        \begin{itemize}
            \item references to the most recent implementations of methods of a \textit{unique common signature}
            \item static offsets of top and virtual bases
        \end{itemize}
    \item Each virtual table may be \textit{composed from customized virtual tables} of parents (\textasciitilde{} thunks)
    \item If needed, compiler generates thunks to adjust the this parameter of methods
\end{itemize}

\textbf{Runtime:}

\begin{itemize}
    \item Calls to constructors allocate memory space
    \item Constructor stores pointers to virtual table (or fragments) respectively
    \item Method calls transparently call methods statically or from virtual tables, unaware of real class identity
    \item Dynamic casts may use top pointer
\end{itemize}

Polemics of Multiple Inheritance

\textbf{Full Multiple Inheritance (FMI)}

\begin{itemize}
    \item Most powerful inheritance principle known
    \item More convenient and simple in the common cases
    \item Occurrence of diamond problem not as frequent as discussions indicate
\end{itemize}

\textbf{Multiple Interface Inheritance (MII)}

\begin{itemize}
    \item MII not as complex as FMI
    \item MII together with aggregation expresses most practical problems
    \item Killer example for FMI yet to be presented
    \item Too frequent use of FMI considered as flaw in the class hierarchy design
\end{itemize}

Lessons Learned

\begin{itemize}
    \item Different purposes of inheritance
    \item Heap Layouts of hierarchically constructed objects in C++
    \item Virtual Table layout
    \item LLVM IR representation of object access code
    \item Linearization as alternative to explicit disambiguation
    \item Pitfalls of Multiple Inheritance
\end{itemize}
Further reading...

CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI.
Itanium C++ 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)*.
1997.

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 project.
URL: http://llvm.org/docs/LangRef.html.

Robert C. Martin.
The liskov substitution principle.

Bjarne Stroustrup.
Multiple inheritance for C++.