Multiple Inheritance

Yet another abstraction mechanism:
ReadWriteFile = ReadFile + WriteFile
- Pros: model reality, code reuse (is that a good reason?!) 
- Cons: error prone, challenging implementation (is that a good reason?!) 

Alternatives:
- Composition – no polymorphism
- Interfaces – no code reuse
- Mixins – not common
- Traits – not common
Memory Layout

D* d = new D();
B2* b2 = d;

b2->m_b2  \rightarrow *(b2 + offsetof(B2, m_b2))
d->m_b2  \rightarrow *(d + \Delta(B1) + offsetof(B2, m_b2))

Casting is no longer just a semantic (compile-time) operation

Layout is compiler-dependent (but deterministic)
Objects are constructed according to the order of the inheritance list(s)

- In the common case: **DFS** on the inheritance tree
- class D : public B1, public B2 \(\rightarrow\) B1, B2, D
  - Even if *initialization list* differs!
- Well specified (as opposed to layout – why?)
**Reminder – Vtable and Inheritance**

```cpp
struct B {
    virtual void f() { ... }
};

struct D : B {
    virtual void h() { ... }
};
```

Can we do this in multiple inheritance?
struct B1 {
    virtual void f() { ... }
};
struct B2 {
    virtual void g() { ... }
};
struct D : B1, B2 {
    virtual void h() { ... }
};
D* d = new D();
B1* b1 = d; b1->f();  // ok?
B2* b2 = d; b2->g();  // ok??
struct B1 {
    virtual void f() { ... }
};
struct B2 {
    virtual void f() { ... }
};
struct D : B1, B2 {
    int x;
    void f() override { x = ... }
};
B2* b2 = new D();
b2->f();  // ok?

Where is this?

Why is &D::f written on both vtables?
Vtable and MI – Take 3

```cpp
struct B1 {
    virtual void f() { ... }
};
struct B2 {
    virtual void f() { ... }
    virtual void g() { ... }
};
struct D : B1, B2 {
    void f() override { ... }
    virtual void h() { ... }
};
```

Diagram:

- **Δ(B1)**:
  - `&D::f`: `-Δ(B1)`
  - `&B2::g`: `0`

- **func. fix**:
  - `&D::f`: `0`
  - `&D::h`: `0`
Extended Vtable

- Each Vtable entry includes a function pointer and this fix
- C code for implementing the virtual method call $p->f()$:
  
  ```c
  vtbl_entry* vp = *(vtbl_entry**)p;
  vp += index_of(f);
  void* new_this = (char*)p + vp->delta;
  (vp->func)(new_this);
  ```

- Relatively simple, portable and quite efficient
- Contradicts the C++ Spirit:
  - You pay the price of multiple inheritance even if it is not used
This idea allowed the introduction of multiple inheritance into C++!
Dispatch Algorithm

Given the invocation \( p \rightarrow f() \):

1. Upcast \( p \) to \( B_f \), the uppermost class that defines \( f \) –
   \[ p' = \text{static_cast}<B_f*>(p); \]

2. Find the address of \( f \) in \( B_f \)'s vtbl
   ▶ entry contains most derived implementation of \( f \) or thunk

3. this adjustment: downcast \( p' \) to the most-derived class that
   overrides \( f \) – Done by the thunk

4. Invoke \( f \)

Note: if MI is not used, #1 and #3 are eliminated. Zero overhead!
Why Upcast Before Downcast?

- At the call site, the receiver’s pointer can point to more than one legal place
- The thunk has a hard-coded adjustment

⇒ For the result to be correct, adjustment must always start from the same sub-object:

The one in which the function was first defined
1. \( p = \text{static
cast}<E*>(d) \)
   - Where \( f \) is first defined
2. The index of \( f \): 0
3. Downcast \( p \) to \( G \)
   - Where \( f \) is last overridden
   - Using thunk (not dynamic_cast)
4. Invoke \( p->f() \)

```cpp
D* d = new Z();
d->f();
```
Dealing With Covariant Return

- C++ (and Java) support covariant return:
  - virtual B* f() → D* f() override
- Given multiple inheritance, casting B to D and vice versa may be nontrivial
- Given dynamic dispatch, the compiler can’t do all the work
- Given zero overhead principle, the amount of work must be limited
Try 1: Return as Declared

```cpp
struct D dObj;       // address: 0x1000
struct B1 { ... };  // sizeof(B1) = 8 bytes
struct B2 {
    virtual B2* f() {
        B2* ret = &dObj; // 0x1008
        return ret;
    }
};
struct D : B1, B2 {
    D* f() override {
        return &dObj; // 0x1000
    }
};
B2* b2_b2 = new B2();
B2* b2_d = new D();
B2* r_b2 = b2_b2->f();
B2* r_d = b2_d->f();
r_b2 and r_d have the same static type, but point to different parts of dObj (on this wrong solution). The compiler can’t upcast the second call, because it doesn’t know D::f will be invoked.
```
Try 2: Always Return Base Type

struct D dObj;      // address: 0x1000
struct B1 { ... };  // sizeof(B1) = 8 bytes
struct B2 {
    virtual B2* f() { 
        B2* ret = &dObj;  // 0x1008
        return ret;
    }
};
struct D : B1, B2 {
    D* f() override {
        return /*(B2*)*/&dObj;  // 0x1008
    }
};

D d_d;
B2* r_b2 = d_d.f();
D* r_d = d_d.f();

The compiler casts D::f’s return to a B2*
D::f is expected to return a pointer to a D, which can be assigned to a D*. But the inserted upcast returns a pointer to a B2*.
Try 3: Upcast & Downcast

```cpp
struct D dObj; // address: 0x1000
struct B1 { ... }; // sizeof(B1) = 8 bytes
struct B2 {
    virtual B2* f() {
        B2* ret = &dObj; // 0x1008
        return ret;
    }
};
struct D : B1, B2 {
    D* f() override {
        return *(B2*)*/&dObj; // 0x1008
    }
};
```

The compiler casts `D::f`'s return to a `B2*`

The compiler knows about the upcast in `D::f`, so it downcasts the return if needed. But static cast isn't always possible (later), and dynamic cast is wasteful.
Try 4: Upcast In A Thunk

```c
struct D dObj;       // address: 0x1000
struct B1 { ... };  // sizeof(B1) = 8 bytes
struct B2 {
    virtual B2* f() {
        B2* ret = &dObj; // 0x1008
        return ret;
    }
};

struct D : B1, B2 {
    D* f() override {
        return &dObj; // 0x1000
    }
};
```

In D’s vtable, the returned D is upcast to a B2

```
B2* b2_b2 = new B2();
B2* b2_d = new D();
D d_d;
B2* r_b2 = b2_b2->f();
B2* r_d = b2_d->f();
D* r_d = d_d.f();
```

Now, all the cases above work:
1. Calling B2::f
2. Calling D::f via a pointer to B2
3. Calling D::f directly
Calling D::f Via Vtable

struct D dObj;  // address: 0x1000
struct B1 { ... };  
struct B2 {
    virtual B2* f() {
        B2* ret = &dObj;  // 0x1008
        return ret;
    }
};
struct D : B1, B2 {
    D* f() override {
        return &dObj;  // 0x1000
    }
};

Have 2 entries for f in D’s vtable:
1. Returns a B2
2. Returns a D

D* d_d = new D();
B2* b2_d = d_d;
D* r_d = d_d->f();
B2* r_b2 = b2_d->f();

Now, on the first call to f (via a pointer to D), vtable entry #2 is used, and on the second call vtable #1 is used.
Diamond Inheritance
A.K.A The Dreaded Diamond of Death!

- Diamond: same base class occurs in more than one ancestor
  - Occurs in every large system
  - Must occur if the inheritance hierarchy has a common root

- Common use case:

![Diagram of Diamond Inheritance]

- File
- Input File
- Output File
- Input Output File
Approaches to Repeated Inheritance

- Forbid the situation
  - A diamond occurs naturally in many cases
  - Demand complicates the design
- Single copy of common base class
  - Not always what you want
- Multiple copies of common base class
  - Not always what you want
- Either single or multiple copies of common base class
  - Eiffel’s approach: decision made by designer of D
  - C++’s approach: decision made by designer of B1 and B2
Multiple Occurrences of a Base

```cpp
struct ClickableNode : Node {
    // A list node that points to a clickable object
};

struct DrawableNode : Node {
    // A list node that points to a drawable object
};

struct ButtonNode : ClickableNode, DrawableNode {
    // A list node that points to a button,
    // which is both clickable and drawable
};
```
1 Common Base, 2 Sub-objects

Do we really need 2 node sub-objects?

Class Diagram

Object Diagram
The *virtual* keyword causes *InitCheckedArray* to contain a single sub-object of class *Array* instead of two!
Standard C++ Stream I/O Library

I/O Streams

- `istream`
- `osstream`
- `iostream`

File Streams

- `fstreambase`
- `ifstream`
- `ofstream`

Input stream stuff

Stream status flags

In-out stream

File stuff (sets status flags)

Output file stuff