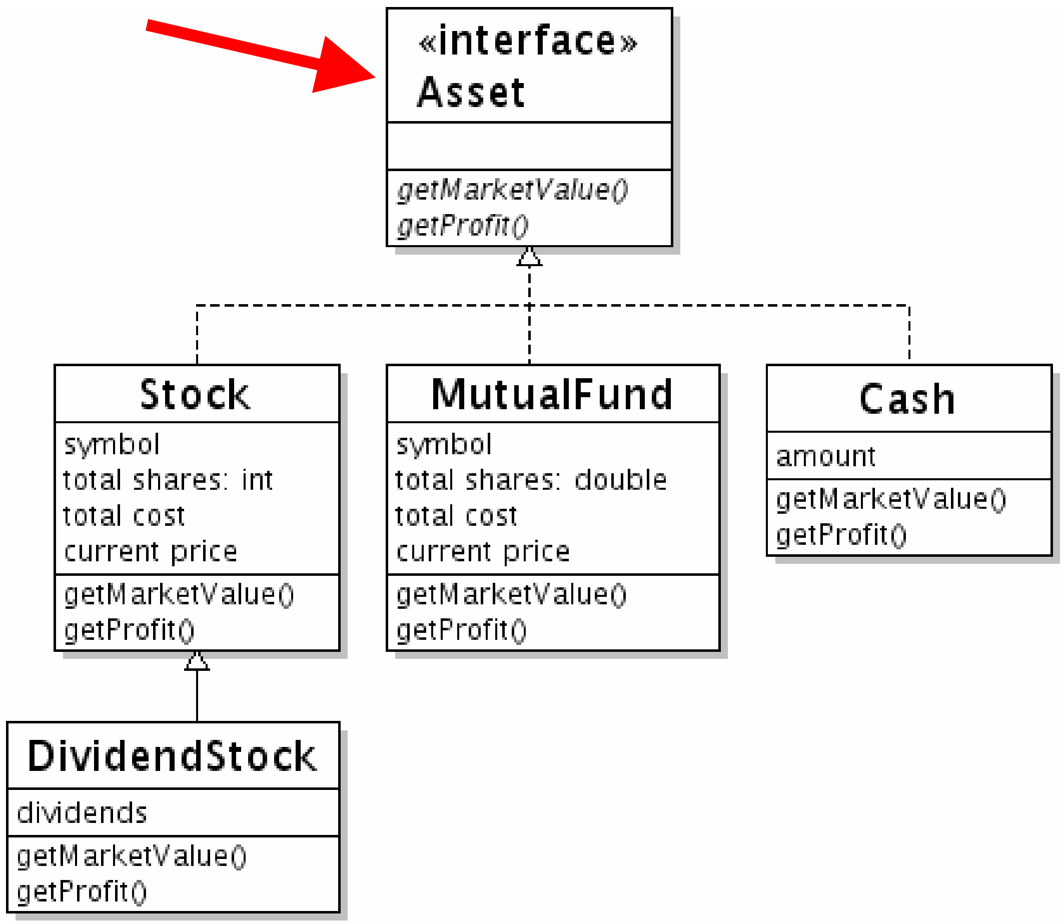
CSE 303
Lecture 24

Inheritance in C++, continued

slides created by Marty Stepp
http://www.cs.washington.edu/303/

Recall: Investments design



- we implemented the inheritance between Stock and DividendStock
- now we'd like an interface for the top-level supertype

Interfaces, abstract classes

- Java provides two special features for creating type hierarchies:
  - **interfaces**: Sets of method declarations with no bodies.
    Classes can promise to implement an interface.
    Provides a supertype without any code sharing.
    - key benefit: **polymorphism**. Can treat multiple types the same way.
  - **abstract classes**: Partially implemented classes that can have a mixture of declarations (without bodies) and definitions (with bodies).
    - a hybrid between a class and an interface
- C++ does not have interfaces, but it (sort of) has abstract classes.

Pure virtual methods

```
class Name {
    public:
        virtual returntype name(parameters) = 0;
        ...
};
```

- **pure virtual method**: One that is declared but not implemented.
  - If a class has any pure virtual methods, no objects of it can be made.
    - We call this an **abstract class**.
  - declared by setting the method equal to 0
  - must be implemented by subclasses (else they will be abstract)

An "interface"

```
#ifndef _ASSET_H
#define _ASSET_H

// Represents assets held in an investor's portfolio.
class Asset {
    public:
        virtual double cost() const = 0;
        virtual double marketValue() const = 0;
        virtual double profit() const = 0;
};

#endif
```

- Simulate an interface using a class with all pure virtual methods
  - we don't need `Asset.cpp`, because no method bodies are written
  - other classes can extend `Asset` and implement the methods

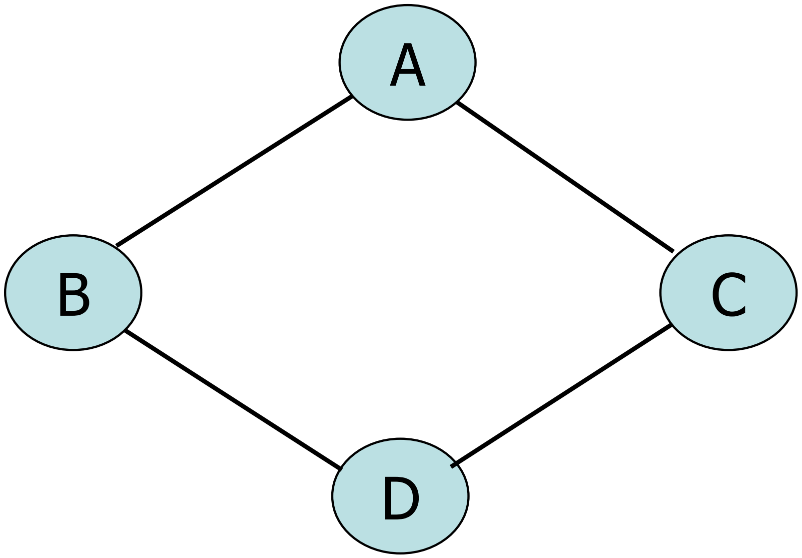
Multiple inheritance

```
class Name : public BaseClass1, public BaseClass2, ...,
             public BaseClassN {
    ...
};
```

- **single inheritance**: A class has exactly one superclass (Java)
- **multiple inheritance**: A class may have >= 1 superclass (C++)
  - powerful
  - helps us get around C++'s lack of interfaces
    - (can extend many abstract classes if necessary)
  - can be confusing
  - often leads to conflicts or strange bugs

Potential problems

- common dangerous pattern: "The Diamond"
  - classes B and C extend A
  - class D extends A and B



- problems:
  - D inherits two copies of A's members
  - If B and C both define a member with the same name, they will conflict in D

- How can we solve these problems and disambiguate?

Disambiguating

```
class B {              // B.h
    public:
        virtual void method1();
};

class C {              // C.h
    public:
        virtual void method1();
};

                       // D.cpp
void D::foo() {
    method1();         // error - ambiguous reference to method1
    B::method1();      // calls B's version
}
```

- *Explicit resolution* is required to disambiguate the methods

Virtual base classes

```
class Name : public virtual BaseClass1, ...,
             public virtual BaseClassN {
    ...
};
```

- declaring base classes as `virtual` eliminates the chance that a base class's members will be included twice

Friends (with benefits?)

```
class Name {
    friend class Name;
    ...
};
```

- a C++ class can specify another class or function as its *friend*
  - the friend is allowed full access to the class's private members!
  - a selective puncture in the encapsulation of the objects
  - (should not be used often)
    - common usage: on overloaded operators outside a class (e.g. <<)

Private inheritance

```
class Name : private BaseClass {
    ...
};
```

- **private inheritance**: inherits behavior but doesn't tell anybody
  - internally in your class, you can use the inherited behavior
  - but client code cannot treat an object of your derived class as though it were an object of the base class (no polymorphism/subtype)

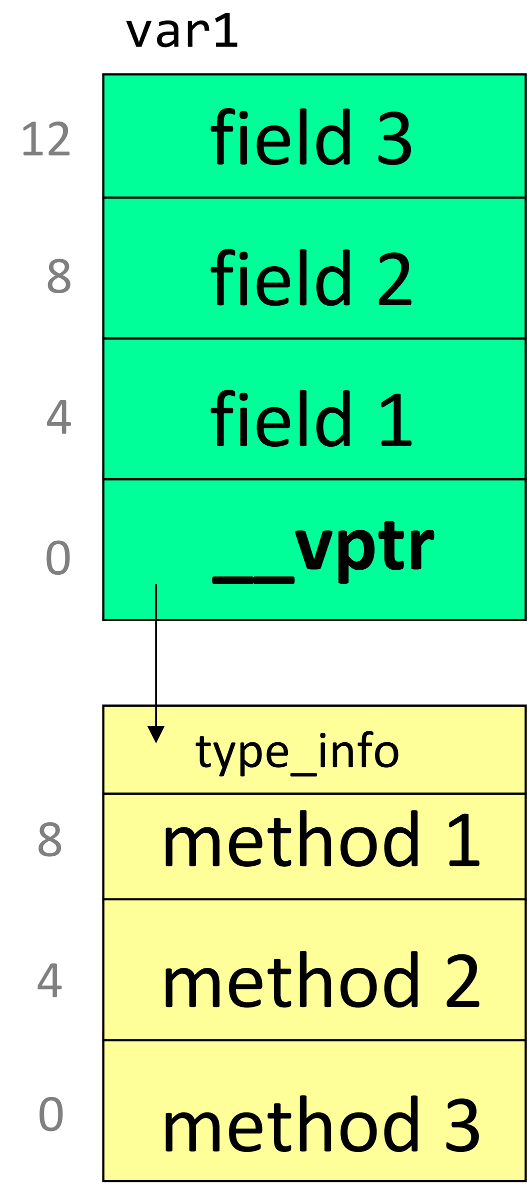
  - a way of getting code reuse without subtyping/polymorphism

Objects in memory

```
A* var1 = new B();
```

- each object in memory consists of:
  - its fields, in declaration order
  - a *pointer* to a structure full of information about the object's methods
    (a **virtual method table** or **vtable**)

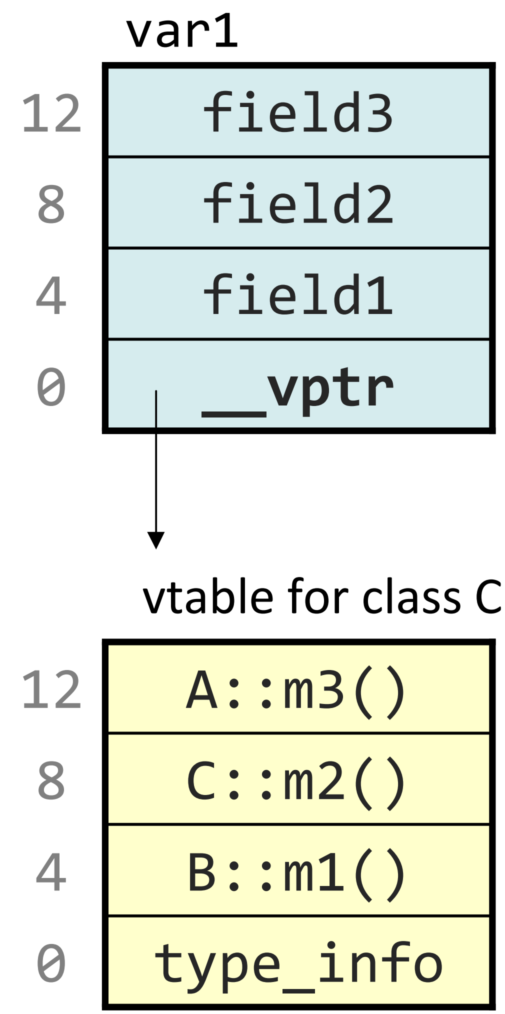
  - one vtable is shared by all objects of a class
  - the vtable also contains information about the type of the object
- use `g++ -fdump-class-hierarchy` to see memory layout



Object memory layout

```
class A {
    int field1;
    virtual void m1(int x);
    virtual void m2(int x);
    virtual void m3(int x);
};

class B : public A {
    float field2;
    virtual void m1(int x);
};

class C : public B {
    int field3;
    virtual void m2(int x);
};
```

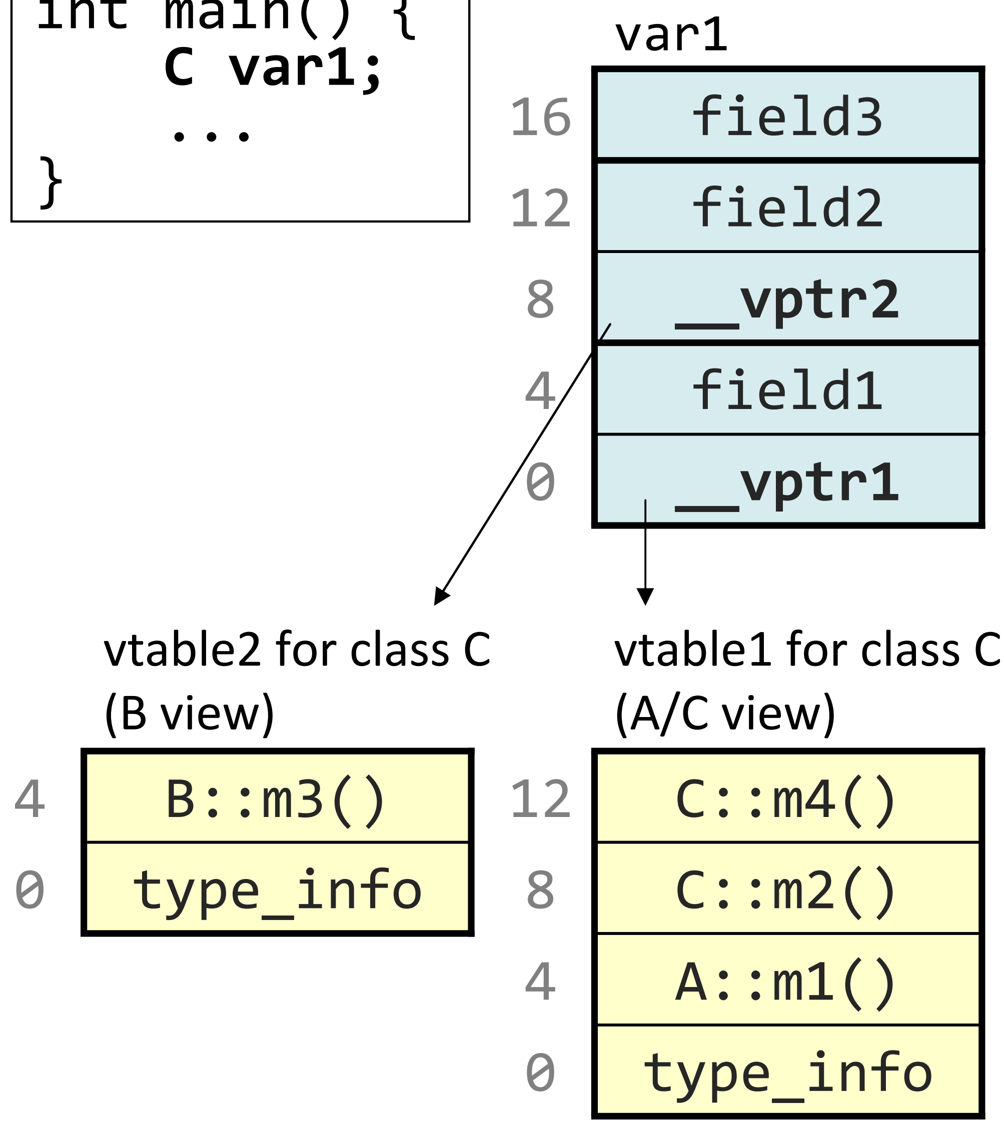
```
int main() {
    C var1;
    ...
}
```



Multiple inheritance layout

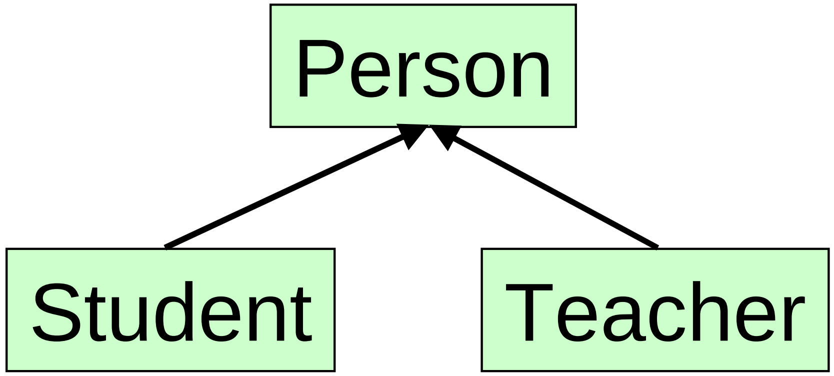
```
class A {
    int field1;
    virtual void m1(int x);
    virtual void m2(int x);
};

class B {
    float field2;
    virtual void m3(int x);
};

class C : public A,
          public B {
    int field3;
    virtual void m2(int x);
    virtual void m4(int x);
};
```

```
int main() {
    C var1;
    ...
}
```



Type-casting pointers

```
Person* p1 = new Student();
Person* p2 = new Teacher();
```

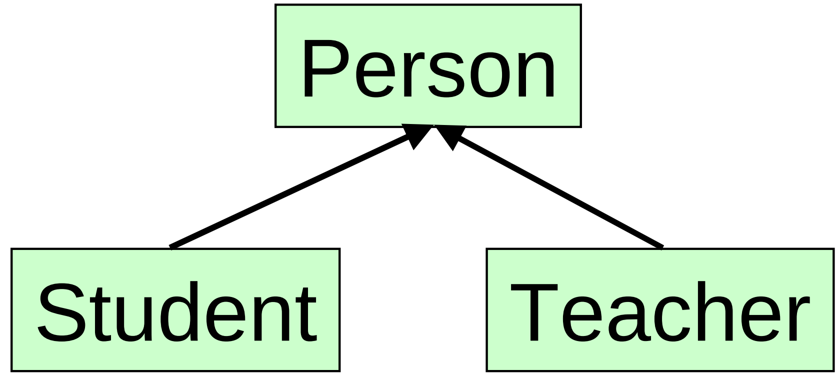



```
Student* s1 = (Student*) p1;   // ok
Student* s2 = (Student*) p2;   // subtle bugs!
```

- casting up the inheritance tree works
- but if the cast fails, can introduce subtle bugs
- why is the above code a problem?
  - p2's vtable is the `Teacher` vtable; using it as a `Student` will cause the wrong methods to be called, or the wrong addresses to be mapped on lookups

Dynamic (checked) casts

```
Person* p1 = new Student();
Person* p2 = new Teacher();
```

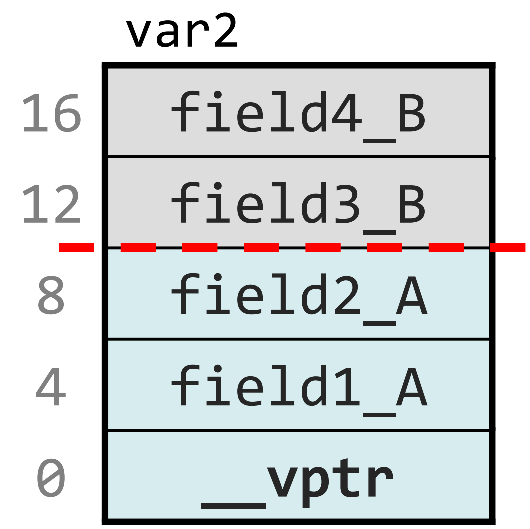



```
Student* s1 = dynamic_cast<Student*>(p1);   // ok
Student* s2 = dynamic_cast<Student*>(p2);   // s4 == NULL
```

- `dynamic_cast` returns NULL if the cast fails
- code still crashes, but at least it doesn't behave in unexpected ways

Slicing

```
class A { ... };
class B : public A { ... };

...

B var1;
A var2 = var1;    // sliced!
```



- **slicing**: When a derived object is converted into a base object.
  - extra info from B class is lost in `var2`
  - often, this is okay and doesn't cause any problems
  - but can lead to problems if data from the "A part" of `var1` depends on data from the "B part"