Object-Oriented Principles and Practice / C++

Alice E. Fischer

June 17, 2013

Abstract Classes Revisited

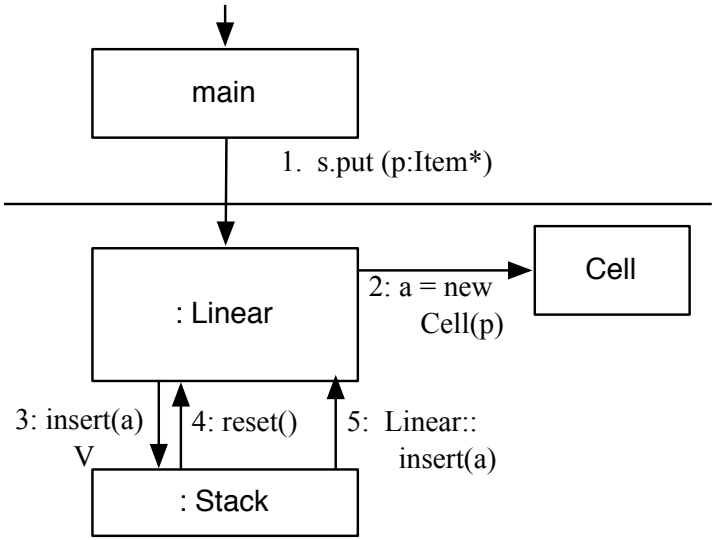
Collaboration Diagrams

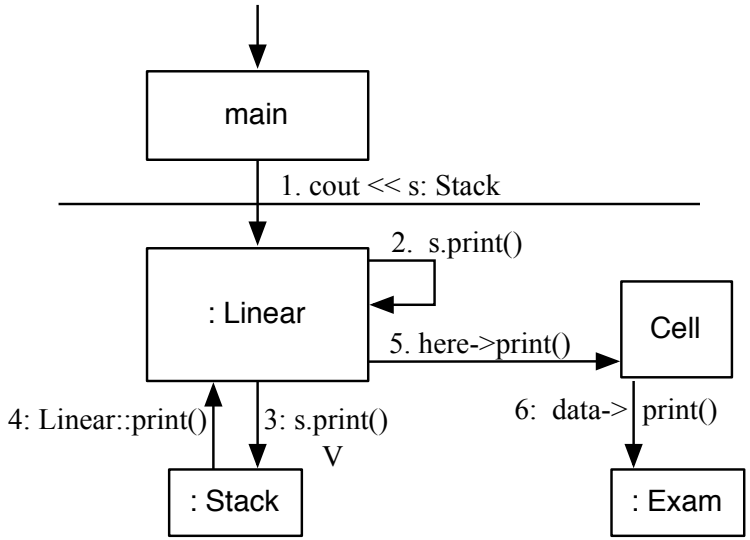
Collaboration Diagrams

A *collaboration diagram* shows how control moves from one class to another in the process of executing a single function call.

- We use collaboration diagrams to show the action of two virtual function calls.
- The numbering system used here is a simplification of the standard UML numbering.
- Stack inherits a put function from Linear.
- During execution of Stack::put(), control goes first to Linear, which allocates a new Dell.
- Then insert is called; it is virtual and defined in Stack, so control goes to Stack.
- Stack calls Linear::reset() to initialize the insertion pointers, then calls the inherited insert() to perform the insertion.

Collaboration Diagrams





Look, also, at the structure chart on page 193.

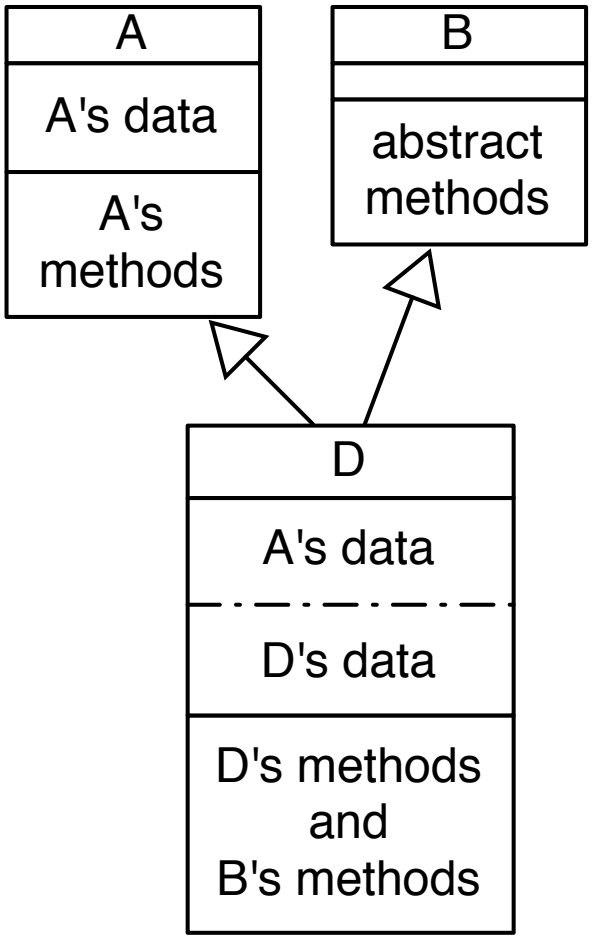
Multiple Inheritance

Concepts
Structure of the objects
Donut diagrams
Virtual inheritance

Multiple Inheritance

Multiple inheritance simply means deriving a class from two or more base classes.

Suppose class D is derived from both A and B, where A is a real class and B is a pure abstract class. B supplies a set of promises that class D must fulfill.



Abstract Class Inheritance Example

We derive the class Item
from Exam (a real class) and Ordered (an interface):

- `class Item : public Exam, public Ordered {...}`
- If a class is Ordered, we can sort it.
- The integers are a predefined ordered set, but Exams are not.
- We can make Exams into an ordered set by defining three functions.
- Functions defined in Item satisfy the promises in Ordered.
- Constants defined in Item could be needed by algorithms that operated on sorted domains.

Multiple Inheritance

Now suppose class D is derived from both A and B, and both A and B are real classes with data members. Then:

- A D object has three parts, in this order: data members defined in A, members defined in B, members defined in D.
- You can cast a D object to either type A or type B.
- You can use a D object with functions defined in all three classes.

Java does not allow multiple inheritance because the B portion cannot start at the first byte of the object.

Object structure

Suppose class `D` is multiply derived from both `A` and `B`.
We write this as `class D : A, B { ... };`.

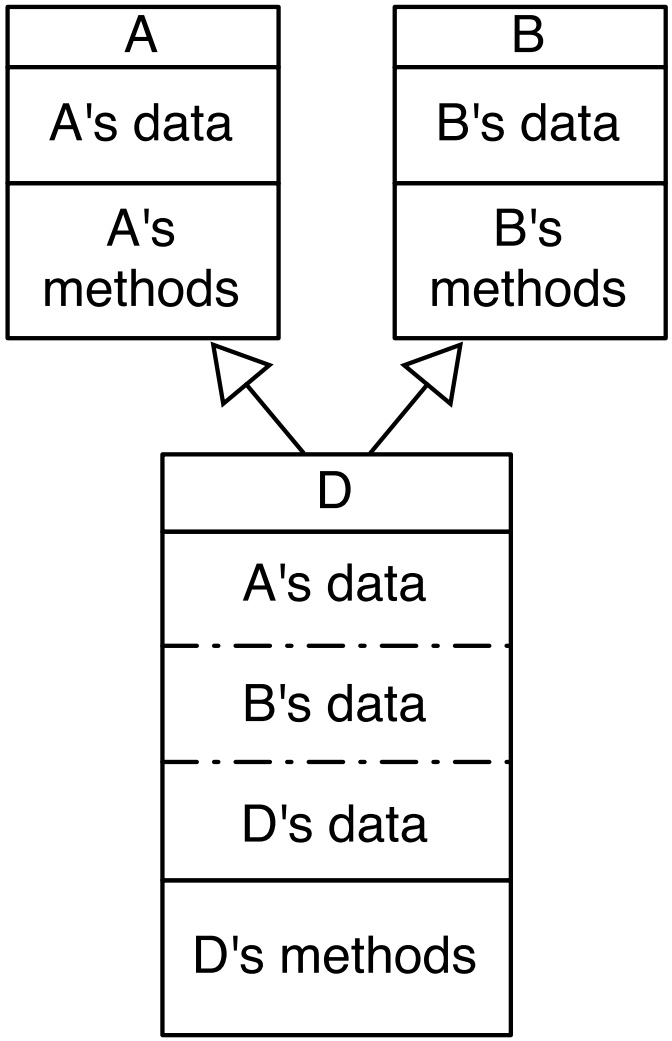
Each instance of `D` has "embedded" within it an instance of `A` and an instance of `B`.

All data members of both `A` and `B` are present in the instance, even if they are not visible from within `D`.

Derivation from each base class can be separately controlled with privacy keywords, e.g.:

```
class D : public A, protected B { ... };.
```

True Multiple Inheritance



Diamond-pattern Inheritance

One interesting kind of derivation is the donut pattern.

```
class C                 { ... x ... };
class A : public C { ... };
class B : public C { ... };
class D : public A, public B { ... };
```

An instance of `D` contains *two* instances of `C`: one in `A` and one in `B`.

These can be distinguished using qualified names.
Suppose `x` is a public data member of `C`.
Within `D`, we can write `D::A::x` to refer to the first copy, and `D::B::x` to refer to the second copy.

Virtual Inheritance

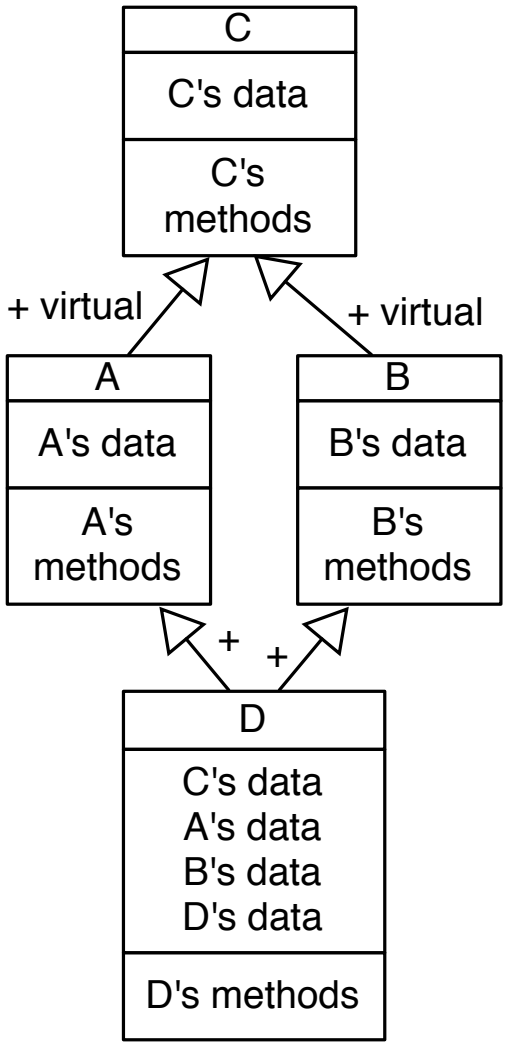
Use virtual inheritance if you do not want two copies of C's data:

```
class C                          { ... x ... };
class A : public virtual C { ... };
class B : public virtual C { ... };
class D : public A, public B { ... };
```

Now an instance of `D` contains *one* instance of `C`'s data members.

Why did they use the word "virtual"? Probably because it was already there and they did not like introducing a new keyword. It has little or nothing to do with virtual functions.

Donut Inheritance



Template Example

Using templates with polymorphic derivation

To illustrate templates, I converted `20a-Multiple` to use template classes. The result is in `20b-Multiple-template`.

There is much to be learned from this example.
Today I point out only a few features.

Container class hierarchy

As before, we have `PQueue` derived from `Linear` derived from `Container`.

Now, each of these have become template classes with parameter `class T`.
`T` is the item type; the queue stores elements of type `T*`.

The main program creates a priority queue using
`PQueue<Item> P;`

Item class hierarchy

As before, we have `Item` derived from `Exam, Ordered`.

`Item` is an *adaptor* class.
It bridges the requirements of `PQueue<T>` to the `Exam` class.

Ordered template class

`Ordered<KeyType>` describes an abstract interface for a total ordering on elements of abstract type `KeyType`.

`Item` derives from `Ordered<KeyType>`, where `KeyType` is defined in `exam.hpp` using a `typedef`.

An `Ordered<KeyType>` requires the following: Colorblue

```
virtual const KeyType& key() const                          =0;
virtual bool            operator <  (const KeyType&) const =0;
virtual bool            operator == (const KeyType&) const =0;
```

That is, there is the notion of a sort key. `key()` returns the key from an object satisfying the interface, and two keys can be compared using < and ==.

Alternative `Ordered` interfaces

As a still more abstract alternative, one could require only comparison operators on abstract elements (of type `Ordered`). That is, the interface would have only two promises:] Colorblue

```
virtual bool operator <  (const Ordered&) const  =0;
virtual bool operator == (const Ordered&) const  =0;
```

This has the advantage of not requiring an explicit key, but it's also less general since keys are often used to locate elements (as is done in the demo).

Casts in C++

- Type identity
- Syntax for calling casts:
  - C-style syntax, with parentheses and with coercion.
  - Function-call syntax
  - Explicit angle-bracket syntax
- Four types of casts:
  - C-style static casts.
  - C-style reinterpret casts.
  - Const casts.
  - Dynamic casts.

typeid operator

```
#include <typeinfo>
```

- typeid allows a program to check the type of an expression: `typeid (expression)`
- This operator returns a reference to a constant object of type `type_info` that is defined in the standard header file `<typeinfo>`. This is a null-terminated character string with a human-readable name for the type, in an implementation dependent format.
- The returned value can be compared with another one using operators $==$ and $!=$:
  `if (typeid(a) != typeid(b)) ...`

Cast Syntax

C++ provides four ways to call a cast. The first two ways are supported in C. The third and fourth are new with C++:

- The old C syntax: `k = (int) f;`
- Implicit syntax (coercion): `k = f;`
- Function call syntax: `k = int(f);`
- Explicit cast syntax: `k = static_cast<int> f;`

Each syntax can be used with all four kinds of cast semantics.

Use these declarations in the following examples:

```
float f, *pflo;
int k, *pint; const int ck, *pcint;
```

C++ has Four Kinds of Cast Semantics

- Static cast: a type conversion, such as changing a float to an int.
- Reinterpret cast: relabel the base type of a pointer to some other pointer type.
- Const cast: remove the const property of a pointer for the duration of one line of code, to permit the program to assign a value to the underlying const variable.
- Dynamic cast: given a pointer to an object in a derivation hierarchy, relabel it as a type that is higher or lower on the derivation chain.

Static Casts

- Static casts permit us to write mixed-type arithmetic expressions: `f = 3.1416*k;`
- A static cast changes the representation of a value without changing the meaning.
- It takes one representation of a value and lengthens it, or shortens it, or moves around the bits to arrive at a different representation with approximately the same meaning.
- The compiler will compile unconditional code to do the bit-shifting, whether you call the cast the old C way, implicitly, or using the new C++ syntax.
- This is the only kind of cast that can be done on a non-pointer.

Reinterpret Casts

- Reinterpret casts relabel the base type of a pointer to some other pointer type. The bit pattern in the underlying object is not changed: `pflo = (float*)pint;`
- Reinterpretation happens at compile time; no run-time code is generated.
- Reinterpret casts are used in three important cases,
  - As the final step in conversion of an int value to a float value, after shifting the bits around and incorporating an exponent.
  - To make characters look like integers so that integer arithmetic can be used on them in the process of computing a hash function.
  - To make an integer look like an array of characters so that parts of the integer can be used independently for a radix sort.

Reinterpretation Semantics

- The result of reinterpretation is controlled, repeatable nonsense.
- C and C++ support reinterpretation casts because sometimes we NEED to create controlled nonsense.
- Reinterpretation is dangerous.
- It is like the wolf in Little Red Riding Hood putting on the grandmother's clothing. He appeared to be Granny, but underneath, he was still a wolf, so he did not function like a Granny.

Const Casts

- The const cast works on a pointer:

```
*const_cast<int>(pcint) = 20;
```

- It happens fully at compile time – no run-time code is generated.
- If C++ had Java's `final` property, instead of `const`, this kind of cast would not be needed.
- Its purpose is to permit a const member of an object to be set in the body of the constructor, rather than in a ctor. This may be necessary if the value of the class member cannot be determined at ctor time.
- There is no excuse for using a const cast outside of a constructor.

Dynamic Cast Syntax

- Dynamic casts work on pointers.
- They move up or down a class derivation hierarchy:
- Upward cast: `bp = (Base*) dp;`
- Downward cast: `dp = (Deri*) bp;`

Use these declarations:

```
class Deri :  Base
class Eeri :  Base
Base* bp = new Deri();
Deri* dp;
Eeri* ep,;
```

Dynamic Cast Semantics

- The upward cast is a compile-time reinterpretation: no code is generated.
- It is always legal but rarely needed, since it will be done implicitly any time you use a Deri* object where a Base* parameter was declared.
- The downward cast will cause a run-time type test on the object, O, that bp points at.
- Here O is an object of type Deri and the cast will succeed.
- However, `ep = (Eeri*) bp;` will throw an exception.
- You can't make an object of type Deri into an object of type Eeri by relabeling because these two types probably have different data parts.

Handling Circularly Dependent Classes

Tightly coupled classes

Class `B` *depends on* class `A` if `B` refers to elements declared within class `A` or to `A` itself.

The class `B` definition must be read by the compiler after reading `A`.

This is often ensured by putting `#include "A.hpp"` at the top of file `B.hpp`.

A pair of classes `A` and `B` are *tightly coupled* if each depends on the other.

It is not possible to have both read after the other.
Whichever the compiler reads first will cause the compiler to complain about undefined symbols from the other class.

Example: `List` and `Cell`

Suppose we want to extend a cell to have a pointer to a sublist.

```
class Cell {
  List* sublist;
  Cell* next;
  ...
};
class List {
  Cell* head;
  ...
};
```

This won't compile, because `List` is used (in `class Cell`) before it is defined. But putting the two class definitions in the opposite order also doesn't work since then `Cell` would be used (in `class List`) before it is defined.

Circularity with #include

Circularity is less apparent when definitions are in separate files.

File list.hpp:

```
#pragma once
#include "cell.hpp"
class List { ... };
```

File cell.hpp:

```
#pragma once
#include "list.hpp"
class Cell { ... };
```

File main.cpp:

```
#include "list.hpp"
#include "cell.hpp"
int main() { ... }
```

What happens?

In this example, it appears that `class List` will get read before `class Cell` since `main.cpp` includes `list.hpp` before `cell.hpp`.

Actually, the opposite occurs. The compiler starts reading `list.hpp` but then jumps to `cell.hpp` when it sees the `#include "cell.hpp"` line.

It jumps again to `list.hpp` when it sees the `#include "list.hpp"` line in `cell.hpp`, but this is the second attempt to load `list.hpp`, so it only gets as far as `#pragma once`. It then resumes reading `cell.hpp` and processes `class Cell`.

If there were no references to the List class in Cell, this would work. However, if Cell's .hpp file refers to List or to List functions, those are not yet declared, and compilation ends.

Resolving circular dependencies

Several tricks can be used to allow tightly coupled classes to compile. Assume `A.hpp` is to be read first.

1. Suppose the only reference to `B` in `A` is to declare a pointer. Then it works to put a "forward" declaration of `B` at the top of `A.hpp`, for example:

```
class B;
class A { B* bp; ... };
```

2. If a function defined in `A` references symbols of `B`, then the *definition* of the function must be moved outside the class and placed where it will be read after `B` has been read in, e.g., in the `A.cpp` file.

Resolving circular dependencies – continued

3. Sometimes it works to put the `#include B.hpp` command at the top of A's .cpp file, instead of in the .hpp file.
4. If A's function needs to be inline, this is still possible, but it's much trickier getting the inline function definition in the right place.