

C++ Multiple Inheritance

Multiple Inheritance

An advanced C++ feature

Java does not have it -- Java has "interfaces" which solve 50% of the problem that Multiple Inheritance solves. Interfaces, however, have the advantage of avoiding much of the implementation complication of MI.

MI Concepts

Simple "single" inheritance works well if there is a single domain (e.g. drawing), and the classes are arranged by their position in that domain.

What if there are multiple, conflicting domains?

With single inheritance, you are forced to bring in super classes you do not really want.

MI is complex, but it addresses this problem

Asteroids Example

There is the gravity domain:

Subject to gravity: Asteroid, bomb

No gravity: ship, bullet

There is also the "Explodable" domain -- which does not align with the gravity domain:

Explodable: bomb, bullet

Not explodable: asteroid, ship

Asteroids Solution

Have small rational tree of gravity/no-gravity: root..spaceship, bullet vs. bomb, asteroid

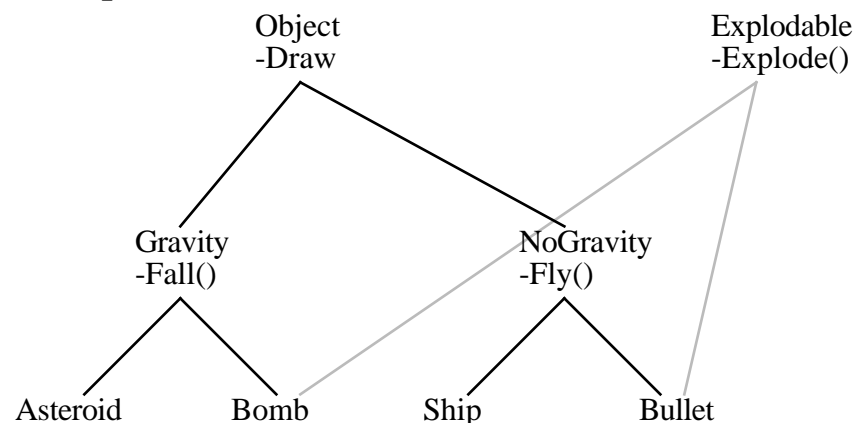
Have a separate "Mixin" class to bring in the "Explodable" property just where necessary

Use MI to bring in qualities which don't fit into the single inheritance logical structure

Point: Not everything can be well described in a single inheritance hierarchy

Point: In a single inheritance hierarchy, you inherit things where you don't want -- EG the Explodable behavior would have to be added way up in Object.

Multiple Inheritance Asteroids



Asteroids Code Features

.h: list multiple superclasses: "class Bomb: public Gravity, public Explodable {"

Constructors: default constructor is still called automatically, or you can use the : syntax to chain up.

Destructors: still called automatically

ISA relationship still holds:

Bomb* or Bullet* -- can send themselves Explode() message

Object* or Ship* -- cannot -- CT err

Casting

Suppose we have..

```
Object* o;
```

```
Gravity *g;
```

```
Bomb* b;
```

```
Explodable* e;
```

1) Up is automatic

```
g=b; // ok, no cast required
```

2) Down is ok, if correct

As before, you can cast down so long as it is correct.

```
g=b; // move up a class
```

```
b=(Bomb*)g; // cast back down -- ok since the object really is a Bomb
```

Use `dynamic_cast<Bomb*>()` to check at runtime

3) Sideways -- NOT OK

```
g=b; // move up a class as before
```

```
e=(Explodable*)g; // Move sideways to the other superclass NO, NOT RELIABLE
```

4) If you need to go sideways, go down instead

```
g=b; // move up a class
```

```
b=(Bomb*)g; // move down (ok, by rule (2))
```

```
e=b; // NOW CAN MOVE UP, in fact no cast is required
```

Remember: just cast down to what it really is -- the other cases will follow automatically

5) Void*

If you ever cast into a void*, cast back out to the exact same thing.

```
Void* v = b;
```

```
<time passes>
```

```
b= (Bomb*)v; // cast back to what it was when you put it in
```

MI Implementation

Object slices -- really complicate the C++ implementation. Multiple base addresses within one object

```
Explode *e = b; // arithmetic here to access the Explodable base addr within a bomb.
```

Conclusion

Pick a "main" single inheritance hierarchy (EG LPane/LView etc in PowerPlant)

Then add in occasional Mixin classes where necessary

Complex, but worthwhile for some problems