Object Oriented Software Design - II

Dynamic casting, Slicing, Private Inheritance, Multiple Inheritance

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

March 5, 2012

Outline

Outline

When inheritance is used

- Inheritance should be used when we have a *isA* relation between objects
  - you can say that a circle is a kind of shape
  - you can say that a rect is a shape
- What if the derived class contains some special function that is useful only for that class?
  - Suppose that we need to compute the diagonal of a rectangle

isA vs. *isLikeA*

- If we put function `diagonal()` only in `Rect`, we cannot call it with a pointer to shape
  - In fact, `diagonal()` is not part of the interface of shape
- If we put function `diagonal()` in `Shape`, it is inherited by `Triangle` and `Circle`
  - `diagonal()` does not make sense for a `Circle`
  - we should raise an error when `diagonal()` is called on a `Circle`
- One solution is to put the function in the `Shape` interface
  - it will return an error for the other classes, like `Triangle` and `Circle`
- another solution is to put it only in `Rect` and then make a *downcasting* when necessary
  - see `./examples/05.multiple-inheritance-examples/shapes_r` for the two solutions

Downcasting

- One way to downcast is to use the `dynamic_cast` construct

```
class Shape { ... };

class Circle : public Shape { ... };

void f(Shape *s)
{
  Circle *c;

  c = dynamic_cast<Circle *>(s);
  if (c == 0) {
    // s does not point to a circle
  }
  else {
    // s (and c) points to a circle
  }
}
```

Dynamic cast

- The `dynamic_cast()` is solved at run-time, by looking inside the structure of the object
- This feature is called *run-time type identification* (RTTI)
- In some compiler, it can be disabled at compile time

Casting

- Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to.
- The subsequent call to member result will produce either a run-time error or a unexpected result.
- There are more safe way to perform casting:

```
dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```

dynamic_cast

- `dynamic_cast` can be used only with pointers and references to objects.
- Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.
- The result is the pointer itself if the conversion is possible;
- The result is `nullptr` if the conversion is not possible:

```
class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };

int main () {
    CBase * pba = new CDerived;
    CBase * pbb = new CBase;
    CDerived * pd;
    pd = dynamic_cast<CDerived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast" << endl;
    pd = dynamic_cast<CDerived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast" << endl;
    return 0;
}
```

static_cast

- `static_cast` can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived.
- however, no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type.
- Therefore, it is up to the programmer to ensure that the conversion is safe.

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

- b would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

reinterpret_cast

- `reinterpret_cast` converts any pointer type to any other pointer type, even of unrelated classes.
- The operation result is a simple binary copy of the value from one pointer to the other.
- All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.
- It can also cast pointers to or from integer types.
- This can be useful in low-level non portable code (i.e. interaction with interrupt handlers, device drivers, etc.)

const_cast

- This type of casting manipulates the constness of an object, either to be set or to be removed.
- For example, in order to pass a const argument to a function that expects a non-constant parameter

```
// const_cast
#include <iostream>
using namespace std;

void print (char * str)
{
  cout << str << endl;
}

int main () {
  const char * c = "sample text";
  print ( const_cast<char *> (c) );
  return 0;
}
```

Outline

What happens?

- Consider the following code snippet

```
class Employee {
  // ...
  Employee& operator=(const Employee& e);
  Employee(const Employee& e);
};

class Manager : public Employee {
  // ...
};

void f(const Manager& m)
{
  Employee e;
  e = m;
}
```

Slicing

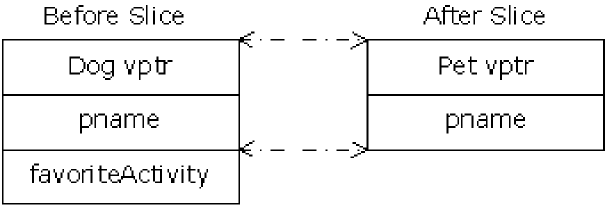
- Only the “Employee” part of m is copied from m to e.
  - The assignment operator of Employee does not know anything about managers!
- This is called “object slicing” and it can be a source of errors and various problems

Another example

- If you upcast to an object instead of a pointer or reference, something will happen that may surprise you: the object is “sliced” until all that remains is the subobject that corresponds to the destination type of your cast.
- Consider the code in `./examples/05.multiple-inheritance-examples/slicin`

Another example

- If you upcast to an object instead of a pointer or reference, something will happen that may surprise you: the object is “sliced” until all that remains is the subobject that corresponds to the destination type of your cast.
- Consider the code in `./examples/05.multiple-inheritance-examples/slicing`
- any calls to `describe()` will cause an object the size of `Pet` to be pushed on the stack
- the compiler copies only the `Pet` portion of the object and slices the derived portion off of the object, like this:



Slicing cont.

- what happens to the virtual function call?

Slicing cont.

- what happens to the virtual function call?
- The compiler is smart, and understand what is going on!
  - the compiler knows the precise type of the object because the derived object has been forced to become a base object.
  - When passing by value, the copy-constructor for a Pet object is used, which initialises the VPTR to the `Pet` VTABLE and copies only the `Pet` parts of the object.
  - There's no explicit copy-constructor here, so the compiler synthesises one.
  - Under all interpretations, the object truly becomes a `Pet` during slicing.

Outline

Private inheritance

- Until now we have seen **public inheritance**
  - A derived class inherits the interface **and** the implementation of a base class
- With **private inheritance** it is possible to inherit only the implementation

```
class Base {
   int p;
protected:
   int q;
public:
   int f();
};
class Derived : private Base {
public:
   int g();
};
int main() {
   Derived obj;
   obj.g();
}
```

Private inheritance

Can access q and f()

I can only call g() but not f()

Private inheritance

- Private inheritance does not model the classical *isA* relationship
- In particular, it is not possible to automatically upcast from derived to base class

```
class Base {};
class DerivedA : public Base {};
class DerivedB : private Base {};

Base *ptr;
DerivedA pub;
DerivedB priv;

ptr = &pub;  // ok
ptr = &priv; // error!!
```

DerivedB cannot be accessed as Base

Inheritance rules

Public Inheritance

In Base	In Derived	Client
private	cannot access	cannot access
protected	as protected members	cannot access
public	as public members	can access

Protected Inheritance

In Base	In Derived	Client
private	cannot access	cannot access
protected	as protected members	cannot access
public	as protected members	cannot access

Private Inheritance

In Base	In Derived	client
private	cannot access	cannot access
protected	cannot access	cannot access
public	as private members	cannot access

Private Inheritance

- Why private inheritance?
  - Because we want to re-use implementation but not the interface
  - It can be seen as a sort of composition
- When to use it
  - It is not a popular technique
  - Composition is better done by declaring a member to another class

Composition

```
class B {
    A* ptr;
public:
    B() {
        ptr = new A();
    }
    ~B() {
        delete ptr;
    }
};
```

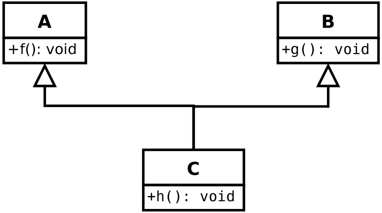
Private Inheritance

```
class B : private A {
public:
    B() : A() {
    }
    ~B() {
    }
};
```

Outline

Multiple inheritance

- A class can be derived from 2 or more base classes



- C inherits the members of A and B

Multiple inheritance

- Syntax

```
class A {
 public:
   void f();
};

class B {
public:
   void f();
};

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

- If both A and B define two functions with the same name, there is an ambiguity
  - it can be solved with the scope operator

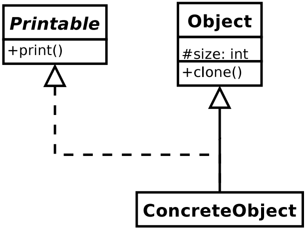
```
C c1;

c1.A::f();
c1.B::f();
```

Why multiple inheritance?

- Is multiple inheritance really needed?
  - There are contrasts in the OO research community
  - Many OO languages do not support multiple inheritance
  - Some languages support the concept of “Interface” (e.g. Java)
- Multiple inheritance can bring several problems both to the programmers and to language designers
- Therefore, the much simpler *interface inheritance* is used (that mimics Java interfaces)

Interface inheritance

- It is called interface inheritance when an onjecy derives from a base class and from an *interface class*
- A simple example

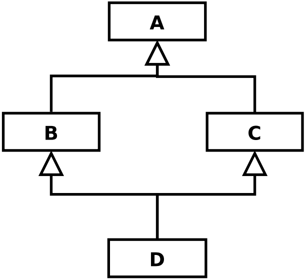


Interface and implementation inheritance

- In *interface inheritance*
  - The base class is abstract (only contains the interface)
  - For each method there is only one final implementation in the derived classes
  - It is possible to always understand which function is called
- Implementation inheritance is the one normally used by C++
  - the base class provides some implementation
  - when inheriting from a base class, the derived class inherits its implementation (and not only the interface)

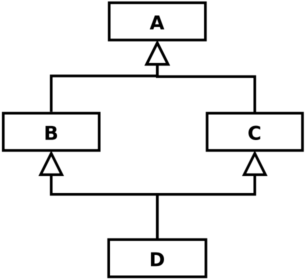
The diamond problem

- What happens if class D inherits from two classes, B and C which both inherith from A?
- This may be a problem in object oriented programming with multiple inheritance!



The diamond problem

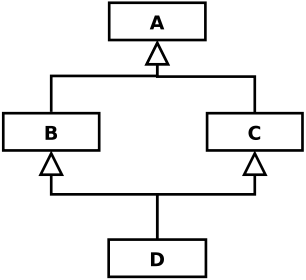
- What happens if class D inherits from two classes, B and C which both inherith from A?
- This may be a problem in object oriented programming with multiple inheritance!



- Problem:
  - If a method in D calls a method defined in A (and does not override the method),

The diamond problem

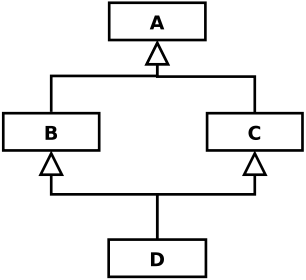
- What happens if class D inherits from two classes, B and C which both inherith from A?
- This may be a problem in object oriented programming with multiple inheritance!



- Problem:
  - If a method in D calls a method defined in A (and does not override the method),
  - and B and C have overridden that method differently,

The diamond problem

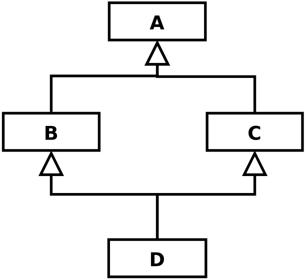
- What happens if class D inherits from two classes, B and C which both inherith from A?
- This may be a problem in object oriented programming with multiple inheritance!



- Problem:
  - If a method in D calls a method defined in A (and does not override the method),
  - and B and C have overridden that method differently,
  - from which class does D inherit the method: B, or C?

The diamond problem

- What happens if class D inherits from two classes, B and C which both inherith from A?
- This may be a problem in object oriented programming with multiple inheritance!

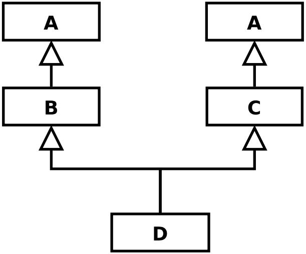


- Problem:
  - If a method in D calls a method defined in A (and does not override the method),
  - and B and C have overridden that method differently,
  - from which class does D inherit the method: B, or C?
  - In C++ this is solved by using keyword “virtual” when inheriting from a class

Virtual base class

- If you do not use virtual inheritance

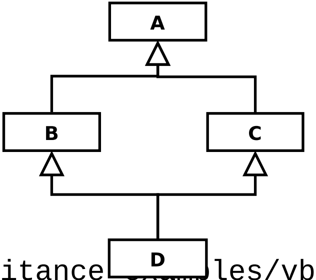
```
class A {...};
class B : public A {...};
class C : public A {...};
class D : public B, public C
{
  ...
};
```



- With public inheritance the base class is duplicated
- To use one of the methods of A, we have to specify which “path” we want to follow with the scope operator
- Cannot upcast!
- see

Virtual base class

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



- With virtual public inheritance the base class is inherited only once
- see `./examples/05.multiple-inheritance-examples/vbase.c` for an example

Initializing virtual base

- The strangest thing in the previous code is the initializer for Top in the Bottom constructor.
- Normally one doesn't worry about initializing sub-objects beyond direct base classes, since all classes take care of initializing their own bases.
- There are, however, multiple paths from Bottom to Top,
  - who is responsible for performing the initialization?
- For this reason, the most derived class must initialize a virtual base.

Initializing virtual base

- The strangest thing in the previous code is the initializer for Top in the Bottom constructor.
- Normally one doesn't worry about initializing sub-objects beyond direct base classes, since all classes take care of initializing their own bases.
- There are, however, multiple paths from Bottom to Top,
  - who is responsible for performing the initialization?
- For this reason, the most derived class must initialize a virtual base.
- But what about the expressions in the Left and Right constructors that also initialize Top?
  - they are ignored when a Bottom object is created
  - The compiler takes care of all this for you

Outline

Pointer to member

- Can I have a pointer to a member of a class?
- The problem with it is that the address of a member is only defined with respect to the address of the object
- The C++ pointer-to-member selects a location inside a class
  - The dilemma here is that a pointer needs an address, but there is no “address” inside a class, only an “offset”;
  - selecting a member of a class means offsetting into that class
  - in other words, a *pointer-to-member* is a “relative” offset that can be added to the address of an object

Usage

- To define and assign a pointer to member you need the class
- To dereference a pointer-to-member, you need the address of an object

```
class Data {
public:
  int x;
  int y;
};

int Data::*pm;            // pointer to member
pm = &Data::x;            // assignment
Data aa;                  // object
Data *pa = &aa;           // pointer to object
pa->*pm = 5;              // assignment to aa.x
aa.*pm = 10;              // another assignment to aa.x
pm = &Data::y;
aa.*pm = 20;              // assignment to aa.y
```

Syntax for pointer-to-member functions

- For member functions, the syntax is very similar:

```
class Simple2 {
public:
  int f(float) const { return 1; }
};

int (Simple2::*fp)(float) const;
int (Simple2::*fp2)(float) const = &Simple2::f;

int main() {
  fp = &Simple2::f;

  Simple2 obj;
  Simple2 *p = &obj;

  p->*fp(.5);      // calling the function
  obj.*fp(.8);     // calling it again
}
```

Another example

```
class Widget {
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
  enum { cnt = 4 };
  void (Widget::*fptr[cnt])(int) const;
public:
  Widget() {
    fptr[0] = &Widget::f; // Full spec required
    fptr[1] = &Widget::g;
    fptr[2] = &Widget::h;
    fptr[3] = &Widget::i;
  }
  void select(int i, int j) {
    if(i < 0 || i >= cnt) return;
    (this->*fptr[i])(j);
  }
  int count() { return cnt; }
};

int main() {
  Widget w;
  for(int i = 0; i < w.count(); i++)
    w.select(i, 47);
}
```