# Making Inheritance Work: C++ Issues

Steven Zeil

September 19, 2013

# Outline

# Recording

These slides accompany a recorded video: *Play Video*

# Outline I

# Base Class Function Members

Even if you override a function, the inherited bodies are still available to you.

```
class Person {
public:
  string name;
  long id;
  void print(ostream& out) {
     out << name << " " << id << endl;}
};

class Student: public Person {
public:
  string school;
  void print(ostream& out) { Person::print(out);
                             out << school << endl;}
}
```

# Base Class Constructors

This technique is often used in constructors so that subclasses will only need to initialize their own new data members:

```
class Person {
public:
  string name;
  long id;
  Person (string n, long i)
     : name(n), id(i)
  {}
};

class Student: public Person {
public:
  string school;
  Student (string name, long id, School s)
     : Person(name, id) , school(s)
  {}
}
```

# Outline I

# Implementing Data Member Inheritance

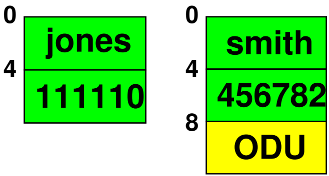
Inheritance of data members is achieved by treating all new members as extensions of the base class:

```
class Person {
public:
  string name;
  long id;
};

class Student: public Person {
public:
  string school;
}
```

# Extending Data Members

- When a compiler processes data member declarations, it assigns a byte offset to each one.
- Inherited members occur at the same byte offset as in the base class




  - so code like `p->name` can translate the same whether `p` points to a Person or a Student.
    - `p->name` is translated as "add 0 to the address in *p*"
    - `p->id` is translated as "add 4 to the address in *p*"

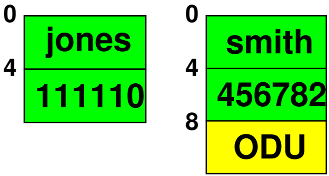
# Assignment & Extension

In most OO languages, we can do

```
superObj = subObj;
```

but not

```
subObj = superObj;
```




- Assigning to a superclass object discards the extra data
  - Presumably, `(Smith, 456782)` is still a valid *Person*
  - Even if it loses the information about Smith being a student
- Assigning to a subclass object requires the system to invent data.
- If we assign Jones to a student object, what value should the system copy into the `school`?

# Outline I

# Virtual Destructors

As we've seen, subclasses can add new data members.

What happens if we add a pointer:

gradStudent.cpp

and we don't want to share?

# Deleting Pointers and Inheritance

Consider the following two delete statements:

```
Person* g1 = new GraduuateStudent(...);
GraduateStudent* g2 = new GraduateStudent(...);
   ⋮
delete g1;  // compiler-generated ~Person() is called
delete g2;  // ~GraduateStudent() is called
```

- Both calls are resolved by compile-time binding
  - Therefore the first delete leaks memory - undergraduateRecords is not cleaned up
- Fix would seem to be to force dynamic binding on the destructors

# Making the Destructor Virtual

The trick is that this has to be done at the top of the inheritance hierarchy

virtualDestruct.cpp

even though,

- at the time we wrote that class, there may have been no obvious need for a destructor
- this seems to violate the Rule of the Big 3
    - We'll look at the other two in just a moment

# Outline I

# Virtual Assignment

If subclasses can introduce new data members, should assignment be virtual so that we can guarantee proper copying of those extended data members?

# Virtual Assignment Example

```
void foo(Person& p1, const Person& p2)
{
   p1 = p2;
}

GraduateStudent g1, g2;
  ⋮
foo(g1, g2);
```

- If *p1* and *p2* "really" have *underGraduateRecord* fields, shouldn't we make sure those get copied properly during assignment?
  - Seems reasonable in this case.
  - But it means that assignment and copying will behave very differently, which is likely to catch programmers by surprise.

# What's the Problem with Virtual Assignment?

If you try it, the inherited members aren't what you might expect:
inherAsst.cpp
You actually wind up with multiple overloaded assignment operators in the subclasses.

# What's the Problem with Virtual Assignment? (cont.)

- To make this work, you will need to implement both the virtual and the normal operators
- Implementing the virtual one is tricky because you might not get a GraduateStudent on the right:

```
void foo(Student& s1, const Student& s2)
{
   s1 = s2;
}

Student s;
GraduateStudent g;
   ...
foo(g, s); // problem: s has no undergraduateRecords
```

# Recommendation

- There's no clear consensus in the C++ community about making assignment virtual.
- I recommend against it just because it's potentially confusing.
  - Try to avoid using assignment in situations where the "true" data type on the left is uncertain.

# Outline I

# Virtual constructors

- Constructors can never be made virtual
- This can lead to problems when we need to create a copy.

# Example: evaluating a cell reference

## cellrefnode.h

```
// Evaluate this expression

Value* CellReferenceNode::evaluate(const SpreadSheet& s) const
{
  Cell* cell = s.getCell(value);
  Value* v = (Value*)cell->getValue();
  if (v == 0)
    return new ErrorValue();
  else
    return v;
}
```

- We would be better off returning a copy of the spreadsheet cell's value rather than the actual one.
  - Each Cell owns (does not share) its Value
  - Cell may therefore delete that Value
    - don't want to risk some other code doing so
- But how do we make a copy?

# Not like this!

```
Value* theCopy = new Value(*v);
```

- How big is a Value?
- Would lose all data members in *v* required for its particular subtype of *Value*

# Better, but Not the "OO Way"

```
Value* newCopy;
if (typeid(*v) == typeid(NumericValue)) {
  newCopy = new NumericValue (v->getNumericValue());
} else if (typeid(*v) == typeid(StringValue)) {
  newCopy = new StringValue (v->render(0));
} else if (typeid(*v) == typeid(ErrorValue)) {
  newCopy = new ErrorValue();
  ⋮
```

(We'll see how `typeid` works shortly.)

# Cloning

Solution is to use a simulated "virtual constructor", generally referred to as a `clone()` or `copy()` function.

```
Value* CellReferenceNode::evaluate(const SpreadSheet& s) co
{
  Cell* cell = s.getCell(value);
  Value* v = (Value*)cell->getValue();
  if (v == 0)
    return new ErrorValue();
  else
    return v->clone() ;
}
```

# clone()

`clone()` must be supported by all values:

```
class Value {
public:
      ⋮
   virtual Value* clone() const;
      ⋮
};
```

# Implementing clone()

Each subclass of Value implements `clone()` as a copy construction passed to new.

```
Value* NumericValue::clone() const
{
  return new NumericValue(*this);
}


Value* StringValue::clone() const
{
  return new StringValue(*this);
}
```

# Outline I

Suppose that I want to be able to test any two Values to see if they are equal

- We'll define "equal" here as meaning that they are the same kind of value and would appear the same when rendered.

# Example: Value::==

We want to explicitly require all subclasses of Value to provide this test:

```
class Value {
      ⋮
   virtual bool operator== (const Value&) const;
};
```

The operator == compares two shapes. Its signature is: `(const Value*, const Value&)` ⇒ `bool`

# Inheriting ==

```
class NumericValue: public Value {
   ⋮
class StringValue: public Value {
    ⋮
```

Both classes inherit the == operator. The signatures are

(const NumericValue*, const Value&) ⇒ bool
(const StringValue*, const Value&) ⇒ bool

# Using the Inherited ==

```
NumericValue n1, n2;
StringValue s1, s2;
bool b = (n1 == n2)
      && (s1 == s2)
      && (n1 == s1);
```

The last clause suggests a problem.

- How do we compare values of different subtypes?
- Should we even allow it?

## Implementing an asymmetric operator

We might implement == for NumericValue as:

```
bool NumericValue::operator==
   (const Value& v)
{
  return d == v.d;
};
```

- But in a call like (n1 == s1), v.d does not make sense.
  - In fact, this will get a compile error

# Implementing an asymmetric operator (cont.)

The problem is that we can easily define

```
bool NumericValue::operator== (const NumericValue& v)
```

but

```
bool NumericValue::operator== (const Value& v)
```

seems impossible, as we cannot anticipate all the values that will ever be defined.

# Working around the == asymmetry

The C++ standard defines a mechanism for *RTTI* (Run Time Type Information).

```
bool NumericValue::operator== (const Value& v)
{
  if (typeid(v) == typeid(NumericValue)) {
       const NumericValue &nv =
                    (const NumericValue&)v;
      return d == nv.d;
  } else
     return false;
};
```

- Note that `typeid()` can be applied both to objects and to types.
- But it can only be used with types/objects that have at least one virtual function.

# RTTI: typeid and downcasting

RTTI also allows you to test to see if v is from a subclass of `NumericValue`

```
if (typeid(NumericValue).before(typeid(v)))
```

or to perform safe *downcasting*:

```
NumericValue* np = dynamic_cast<NumericValue*>(&v);
if (np != 0)
   {// v really was a NumericValue or
    //   subclass of NumericValue
          ⋮
   }
```

- The term "downcasting" refers to the fact that we are moving "down" in hour inheritance hierarchy (assuming we draw the base class at the tops).
    - Upcasting is always safe (and usually is done implicitly)
    - Downcasting can be dangerous if we don't check to see if the object really is waht we think it will be.

# Downcasting Should Not Be a Crutch

Downcasting is often a tempting way to patch a poor initial choice of virtual "protocol" functions.

- 95% of the time, it's a bad idea
  - often leads to subtle, hard to trace bugs

Oddly, though, downcasting is far more widely accepted in Java than in C++.

# Outline I

# Equality Again

Earlier, we looked at the problem of comparing two spreadsheet Values:

```
class Value {
    ⋮
  virtual bool isEqual (const Value&) const;
};
```

```
class NumericValue: public Value {
    ⋮
  virtual bool isEqual (const Value&) const;
};
```

We saw that problems are caused by NumericValue::isEqual getting a parameter of type Value& rather than NumericValue&.

# Why is this so hard?

Why can't we select the best fit from among:

```
class NumericValue: public Value {
    ⋮
  virtual bool isEqual (const NumericValue&) const;
  virtual bool isEqual (const StringValue&) const;
  virtual bool isEqual (const ErrorValue&) const;
};
```

The answer stems from how dynamic binding is implemented.

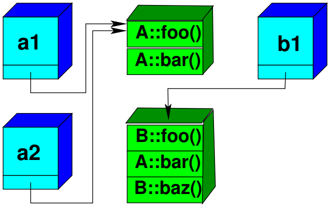
# Single Dispatching

Almost all OO languages offer a *single dispatch* model of message passing:

- the dynamic binding is resolved according to the type of the single object to which the message was sent ("dispatched").
  - In C++, this is the object on the left in a call: `obj.foo(...)`
- There are times when this is inappropriate.
  - But it leads to a fast, simple implementation

# VTables

- Each object with 1 or more virtual functions has a hidden data member.
  - a pointer to a *VTable* for it's class
  - this member is always at a predictable location (e.g., start or end of the object)



# Compiling Virtual Function Declarations

- Each virtual function in a class is assigned a unique, consecutive index number.
- `(*VTable)[i]` is the address of the class's method for the i'th virtual function.

# Example of VTable Use

```
class A {
public:
  A();
  virtual void foo();
  virtual void bar();
};

class B: public A {
public:
  B();
  virtual void foo();
  virtual void baz();
};
```

```
A* a = ???;  // might point to an A or a B object
a->foo();
```

`foo()`, `bar()`, and `baz()` are assigned indices 0, 1, and 2,

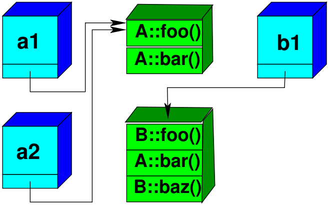
# Example: VTable Structure

- The call `a->foo()` is translated as

```
*(a->VTABLE[0])();
```

- The call `a->bar()` is translated as

```
*(a->VTABLE[1])();
```




Notice that this works regardless of whether *a* points to an *A* object or a *B* object.

- "works" in this case means "does dynamic binding"

# Implementing RTTI

- The address of the VTable is a unique identifier for each class known to the compiler
- This makes the vtable an ideal for implementing RTTI
  - and explains why RTTI is only available for classes with at least one virtual function