Effective C++ Programming

OOA&D ½-Day Excerpt for CNC Software, June-September 2011

Presented by Leor Zolman
BD Software
www.bdsoft.com

Scott Meyers, Ph.D.
Software Development Consultant

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503-638-6028
Fax: 503-974-1887

Last Revised: 10/19/10

Inheritance and Object-Oriented Design

Make Sure Public Inheritance Models "isa"

This is how C++ behaves:

```
class Person { ... };
class Student: public Person { ... };
void dance(const Person& p);          // anyone can dance
void study(const Student& s);         // only students study
Person p;                             // p is a Person
Student s;                            // s is a Student
dance(p);                             // fine, p is a Person
dance(s);                             // fine, s is a Student,
                                      // and a Student isa Person
study(s);                             // fine
study(p);                             // error! p isn't a Student
```


Public Inheritance and Intuition

In English, we say this:

- Birds can fly
- Penguins are birds

In C++, it looks like this:

```
class Bird {
public:
  virtual void fly();                 // birds can fly
  ...
};
class Penguin: public Bird {          // penguins are birds
  ...
};
```

Uh oh: penguins can't fly. "Houston, we have a problem."

Public Inheritance and Intuition

In English, we can be sloppy. In C++, we must be precise:

```
class Bird {
  ...                           // no fly function is declared
};
class FlyingBird: public Bird {
public:
  virtual void fly();
  ...
};
class Penguin: public Bird {
  ...                           // no fly function is declared
};
```


More on Penguins and Flying

The second inheritance hierarchy is *not* necessarily better than the first one:

- For applications dealing only with beaks and wings, the second design is needlessly complex:
  - 2 classes are easier to understand than 3
- Different applications may require different designs
  - Even if they're in the same problem domain

Runtime Errors versus Compile-Time Errors

Another approach to the flying penguin problem:

```
void error(const std::string& msg);              // defined elsewhere

class Bird {
public:
  virtual void fly();
  ...
};

class Penguin: public Bird {
public:
  virtual void fly()
  {
    error("Penguins can’t fly!");
  }

  ...

};
```


Runtime Errors versus Compile-Time Errors

There is an important difference:

- The latter design doesn’t say “penguins can’t fly.”
  - It says, “penguins can fly, but it’s an error for them to try to do so.”
- The three-class hierarchy shown earlier says “penguins can’t fly, period.”
  - You can tell, because programs that try to make penguins fly won’t compile:

```
class Bird { ... };
class FlyingBird: public Bird { ... };
class Penguin: public Bird { ... };

Penguin p;

p.fly();                               // error!
```

Runtime Errors versus Compile-Time Errors

Compile-time error detection is usually superior to runtime errors:

- Much easier to verify that programs are error-free
- No runtime overhead for error detection

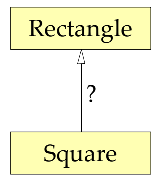
Substitutability

Technically, "isa" implies *substitutability*:

- A derived class object can be used anywhere a base class object can:
  - ➡ Derived class objects are *substitutable* for base class objects
- All code written for base class objects should also work for derived class objects
  - ➡ *All* code!
  - ➡ A derived class object *isa* base class object!

Substitutability and Intuition

Question: should square inherit from rectangle?



- Is a square a rectangle?
- Does all code written for rectangles also work for squares?

Substitutability and Intuition

Consider this code:

```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);

  virtual int height() const;            // return current
  virtual int width() const;             // values

  ...
};

void doubleArea(Rectangle& r)
{
  int oldHeight = r.height();

  r.setWidth(2 * r.width());             // double r's width

  assert(r.height() == oldHeight);
};
```

Clearly the assertion should never fail.

Substitutability and Intuition

Now consider this:

```
class Square: public Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);

  ...
};

Square s;

...

doubleArea(s);

assert(s.width() == s.height());
```

Clearly *this* assertion should never fail.

- How can we implement Square's member functions so both assertions are satisfied?

Substitutability and Intuition

Conclusions:

- "Isa" is a *technical* term, not a conceptual term:
  - It corresponds to *substitutability*
- Many conceptually "isa" relationships fail the substitutability test:
  - This is a common cause of design errors
- Think carefully before employing public inheritance

Guidelines

Make sure public inheritance models "isa."

Prefer compile-time errors to runtime errors.

Differentiate Between Inheritance of Interface and Inheritance of Implementation

There are actually two kinds of inheritance:

- Inheritance of interface. This corresponds to member function *declarations.*
  - ➡ This leads to *design* reuse.
- Inheritance of implementation. This corresponds to member function *definitions*.
  - ➡ This leads to *code* reuse.

An Example

```
class Shape {
public:
  virtual void draw() const = 0;
  virtual void error(const std::string& msg);
  int objectID() const;
  ...
};
class Rectangle: public Shape { ... };
class Ellipse: public Shape { ... };
```

There are three kinds of functions here:

- draw is a pure virtual
- error is an "impure" virtual
- objectID is nonvirtual

What are the implications of these different declarations?

Pure Virtual Functions

Pure virtual functions specify *inheritance of interface only*:

- Makes sense for Shape::draw — how can you write the code to draw something if you don't know what it is?
- Like saying to designers of subclasses, "You must provide a draw function, but I don't know how you'll implement it."
- Allows designers of base classes to specify required functionality of subclasses without specifying how the functionality is to be provided.

Defining Pure Virtual Functions

It is legal to define pure virtual functions, but calls must be fully qualified:

```
Shape *ps = new Shape;              // error! Shape is abstract

Shape *ps1 = new Rectangle;
ps1->draw();                        // calls Rectangle::draw

Shape *ps2 = new Ellipse;
ps2->draw();                        // calls Ellipse::draw

ps1->Shape::draw();                 // calls Shape::draw, even
                                    // though ps1 points to a Rectangle

ps2->Shape::draw();                 // calls Shape::draw, even
                                    // though ps2 points to an Ellipse
```

We'll see a use for this facility soon.

Impure Virtual Functions

Impure virtual functions specify *inheritance of interface plus inheritance of a default implementation*.

- Shape::error provides default error-handling capabilities.
- Derived classes may replace the default behavior if they want to.
- Like saying to designers of subclasses, "You must support an error function, but you don't have to write one if you don't want to."

Coupling Mandatory Interface with Default Implementation

Coupling mandatory interface and default implementation can be dangerous:

```
class Airport { ... };                    // represents airports

class Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};
```

Assume there are only two kinds of airplane, and they are both flown the same way.

Coupling Mandatory Interface with Default Implementation

This seems natural:

```
void Airplane::fly(const Airport& destination)
{
  default code for flying an airplane to the given destination
}

// ModelA doesn't redefine fly
class ModelA: public Airplane { ... };

// Neither does ModelB
class ModelB: public Airplane { ... };
```

The common code in classes ModelA and ModelB has been moved into the Airplane class.

Coupling Mandatory Interface with Default Implementation

Assume we later decide to add a new type of airplane. The class will doubtless be based on the existing classes ModelA and ModelB:

```
class ModelC: public Airplane {
  ...                                   // no fly function is
                                        // declared
};
```

But if a ModelC is flown *differently* than existing models, we might end up with this:

```
Airport JFK(...);
Airplane *pa = new ModelC;
...
pa->fly(JFK);                           // calls Airplane::fly!
```

This doesn't inspire confidence in the traveling public.

A Better Strategy

The problem can be avoided by offering default behavior to subclasses...

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;
  ...
protected:
  void defaultFly(const Airport& destination);
};
void Airplane::defaultFly(const Airport& destination)
{
  default code for flying an airplane to the given destination
}
```

A Better Strategy

...but not giving it to them unless they ask for it explicitly:

```
class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  {
    defaultFly(destination);
  }
  ...
};

class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  {
    defaultFly(destination);
  }
  ...
};
```


A Better Strategy

The writer of class ModelC is now forced to think about whether the default implementation is the appropriate one:

```
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);

  ...
};

void ModelC::fly(const Airport& destination)
{
  code for flying a ModelC airplane to the given destination
}
```

This isn't perfect, but it's safer than the original design.

Default Implementation Functions

- They're an implementation detail, so they should be protected
- They should never be overridden, so they're nonvirtual
  - Making them virtual yields the original problem:
    - What if somebody forgets to redefine them when they're supposed to?
- They can be implemented as the *definitions* of the corresponding pure virtuals:

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;
  ...
};

void Airplane::fly(const Airport& destination)
{
  default code for flying an airplane to the given destination
}
```


Default Implementation Functions

```
class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  {
    Airplane::fly(destination);
  }
  ...
};

class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  {
    Airplane::fly(destination);
  }
  ...
};
```

Default Implementation Functions

```
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);
  ...
};
void ModelC::fly(const Airport& destination)
{
  code for flying a ModelC airplane to the given destination
}
```


Nonvirtual Functions

Nonvirtual functions specify *inheritance of interface plus inheritance of a mandatory implementation*:

- They identify *invariants over specialization.*
- objectID should be implemented the same for all shapes.
- Like saying to designers of subclasses, "You must support an objectID function, and you must use the one I give you."

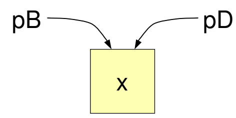
Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Redefining Nonvirtual Functions

Consider this:

```
class B {                        // any random class
public:
  void mf();                     // any random member function
};

class D: public B { ... };

D x;                             // x is an object of type D

B *pB = &x;                      // get pointer to x
D *pD = &x;                      // get another pointer to x
```



Redefining Nonvirtual Functions

We expect these to behave the same way:

```
pB->mf();                        // call mf through pointer
pD->mf();                        // call mf through pointer
```

Unfortunately, they might not.

Redefining Nonvirtual Functions

Nonvirtual function calls are resolved at compile time:

- The compiler chooses which function to call based on the *type of the pointer* to the object, not the object itself.
- If class D defines its own version of mf, then

```
pB->mf();                              // calls B::mf
pD->mf();                              // calls D::mf
```

- Analogous rules apply to references

Moral: never redefine an inherited nonvirtual function.

- This is the general form of a special case we saw earlier:
  - ➡ Make destructors virtual in polymorphic base classes.

Guidelines

Differentiate between inheritance of interface and inheritance of implementation:

- Pure virtual functions specify inheritance of interface only.
- Impure virtual functions specify inheritance of interface plus inheritance of a default implementation.
- Nonvirtual functions specify inheritance of interface plus inheritance of a mandatory implementation.

Never redefine an inherited nonvirtual function.

Avoid Casts Down the Inheritance Hierarchy

Consider an abstract base class for bank accounts:

```
class Person { ... };

class BankAccount {
public:
  BankAccount(const Person *primaryOwner,
              const Person *jointOwner);
  virtual ~BankAccount();
  virtual void makeDeposit(double amount) = 0;
  virtual void makeWithdrawal(double amount) = 0;
  virtual double balance() const = 0;
};
```

Assume there is only one type of bank account:

```
class SavingsAccount: public BankAccount {
public:
  SavingsAccount(const Person *primaryOwner,
                 const Person *jointOwner);
  void creditInterest();                    // add interest to account
  ...
};
```


Keeping Lists of Bank Accounts

The bank could use the standard list template to keep track of its accounts:

```
std::list<BankAccount*> allAccounts;        // all accounts in
                                            // the bank
```

- To avoid resource leaks, a list of tr1::shared_ptrs would often be better.
  - For now, we'll stick with raw pointers.
    - We'll revisit containers of tr1::shared_ptrs later.

To credit interest to each account, you might try this:

```
for (std::list<BankAccount*>::iterator i(allAccounts.begin());
     i != allAccounts.end();
     ++i)
  (*i)->creditInterest();
```

This won't compile. Why not?

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Iterating Over the List of Accounts

A cast lets you tell the compiler what it is too stupid to realize on its own:

```
// a loop that will compile, but that is nonetheless evil
for (std::list<BankAccount*>::iterator i(allAccounts.begin());
     i != allAccounts.end();
     ++i)
  static_cast<SavingsAccount*>(*i)->creditInterest();
```

This is a *downcast*:

- From a base class to a derived class
- It leads to maintenance nightmares
- Casts in general are undesirable:
  - Casts are to C++ programmers as the apple was to Eve

Iterating Over the List of Accounts

Suppose the bank decides to add a second kind of account:

```
class CheckingAccount: public BankAccount {
public:
  void creditInterest();              // add interest to account
  ...
};
```

Now the iteration code has serious problems:

- It continues to cast all BankAccount objects to SavingsAccount objects
- It continues to compile:
  - When you make a cast, the compiler believes you

Fixing the Iteration Code

The most common way to fix the code is like this:

```
for (std::list<BankAccount*>::iterator i(allAccounts.begin());
      i != allAccounts.end();
      ++i) {
  if (*i is of type SavingsAccount*)
    static_cast<SavingsAccount*>(*i)->creditInterest();
  else
    static_cast<CheckingAccount*>(*i)->creditInterest();
}
```

However:

- Virtual functions are the better solution:
  - Type-dependent runtime behavior is why virtual functions were invented!

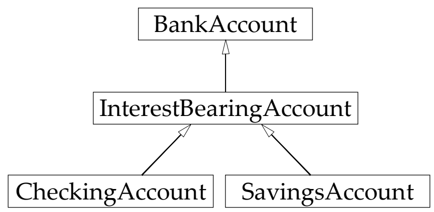
Fixing the Iteration Code

Using virtual functions:

```
class BankAccount { ... };                                    // as before

class InterestBearingAccount: public BankAccount {
public:
  virtual void creditInterest() = 0;
  ...
};

class SavingsAccount: public InterestBearingAccount {
  ...                                                          // as before
};

class CheckingAccount: public InterestBearingAccount {
  ...                                                          // as before
};
```

Fixing the Iteration Code

The new hierarchy looks like this:



And the loop looks like this:

```
for (std::list<BankAccount*>::iterator i(allAccounts.begin());
      i != allAccounts.end();
      ++i)
   static_cast<InterestBearingAccount*>(*i)->creditInterest();
```

This still has a cast, but:

- Type-dependent behavior is handled by the compiler
- If new classes inherit from InterestBearingAccount, the loop still works

Fixing the Iteration Code

The cast can be eliminated by changing the declaration for allAccounts:

```
std::list<InterestBearingAccount*> allIBAccounts;

// a loop that compiles and works, now and forever
for (std::list<InterestBearingAccount*>::iterator i(allIBAccounts.begin());
      i != allIBAccounts.end();
      ++i)
   (*i)->creditInterest();
```

This leads to a general rule for avoiding downcasts:

- Don’t lose the type of the object pointer or reference in the first place

Fixing the Iteration Code

Another approach is to make creditInterest applicable to all accounts:

```
class BankAccount {
public:
  virtual void creditInterest() {}
  ...
};
class SavingsAccount: public BankAccount { ... };
class CheckingAccount: public BankAccount { ... };

std::list<BankAccount*> allAccounts;
for (std::list<BankAccount*>::iterator i(allAccounts.begin());
     i != allAccounts.end();
     ++i)
  (*i)->creditInterest();
```

This is another way to avoid downcasts:

- Move virtual functions up the inheritance hierarchy

Safe Downcasting

Sometime neither of these approaches will work:

- You can’t avoid losing the actual type of an object pointer or reference
- You can’t move virtual functions up the hierarchy
- You really do need to perform a downcast

Use a *safe downcast*:

- Attempt to cast a pointer-to-base to a pointer-to-derived
- If the cast fails, return the null pointer
- Check for the failed downcast in application code

Safe downcasting is provided as part of C++’s RTTI support.

- RTTI = “RunTime Type Identification”

Summary of RTTI Support

Three levels of information:

- Safe downcasting
- Getting object types:
  - Is an object of a particular type?
  - Are two objects of the same type?
- Getting information on an object's type:
  - What is the name of the type (as a `const char*`)?
  - Possibly other information:
    - Names of data members
    - Offsets of non-static data members
    - Names of function members
    - Whether functions are virtual or static
    - Base classes
    - Derived classes
    - Etc.

Safe Downcasting via RTTI

Given these classes again:

```
class BankAccount {
public:
   BankAccount(const Person *primaryOwner,
               const Person *jointOwner);
   virtual ~BankAccount();
   virtual void makeDeposit(double amount) = 0;
   virtual void makeWithdrawal(double amount) = 0;
   virtual double balance() const = 0;
};
class SavingsAccount: public BankAccount {
public:
   SavingsAccount( const Person *primaryOwner,
                   const Person *jointOwner);
   void creditInterest();                    // add interest to account
   ...
};
```

We still want to write a loop to credit interest to all accounts in the bank.

Safe Downcasting via RTTI

We use the dynamic_cast operator:

```
std::list<BankAccount*> allAccounts;          // all accounts in
                                              // the bank
...

for (std::list<BankAccount*>::iterator i(allAccounts.begin());
     i != allAccounts.end();
     ++i) {
  if (SavingsAccount *psa = dynamic_cast<SavingsAccount*>(*i))
    psa->creditInterest();
  else {
    // *i is not a SavingsAccount*
    ...
  }
}
```


Safe Downcasting with References

Very similar to pointer usage, but:

- Cast to a reference, not a pointer
- If the cast fails, an exception is thrown by dynamic_cast:

```
for (std::list<BankAccount*>::iterator i(allAccounts.begin());
     i != allAccounts.end();
     ++i) {
  try {
    SavingsAccount& rsa = dynamic_cast<SavingsAccount&>(**i);
    rsa.creditInterest();
  }
  catch (std::bad_cast&) {
    // **i is not a SavingsAccount
    ...
  }
}
```


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Pointers Versus References

Using exceptions for normal control flow is poor style, so:

- Safe downcast a pointer if there is a chance the cast may fail
- Safe downcast a reference only if the cast is never supposed to fail
- Try to avoid downcasting completely

Beyond Safe Downcasting

The typeid operator:

- Yields information on an object's type:

```
if (typeid(*objectPtr) == typeid(CheckingAccount)) ...
```

The type_info class:

- A class containing information about a type
  - ➡ This is what typeid really returns
- About the only required information is the name of the class:

```
std::cout << "The type of objectPtr is pointer-to-"
          << typeid(*objectPtr).name();
```

  - ➡ The result of type_info::name is implementation-defined!

```
class Widget {};

std::cout << typeid(Widget).name();      // MSVC:    class Widget
                                         // g++:     6Widget
                                         // Comeau:  Widget
```

RTTI and Virtual Functions

RTTI implementations use the virtual function implementation machinery. When RTTI is used on classes with no virtual functions:

- For typeid, results will correspond to the objects' *static* types
  - typeid can hence be safely applied to pointers/references to ints, doubles, pointers, etc.
  - Yet another reason to ensure that base classes declare virtual destructors
- Uses of dynamic_cast won't compile.

tr1::shared_ptrs and Casting

tr1::shared_ptrs are typically preferable to raw pointers, but they complicate casting:

- Casts take and return raw pointers, not smart pointers:

```
std::list<std::tr1::shared_ptr<BankAccount> > allAccounts;            // okay
...

for (std::list<std::shared_ptr<BankAccount> >::iterator i(allAccounts.begin());
     i != allAccounts.end();
     ++i) {
  if (std::tr1::shared_ptr<SavingsAccount> psa =
      dynamic_cast<std::tr1::shared_ptr<SavingsAccount> >(*i)) {      // error!
      ...
  }
}
```

- For cases where casting is necessary, this is a problem.

tr1::shared_ptrs and Casting

TR1 addresses this problem through cast-like templates:

```
std::list<std::tr1::shared_ptr<BankAccount> > allAccounts;
...
for (std::list<std::tr1::shared_ptr<BankAccount> >::iterator i(allAccounts.begin());
      i != allAccounts.end();
      ++i) {
   if (std::tr1::shared_ptr<SavingsAccount> psa =
      std::tr1::dynamic_pointer_cast<SavingsAccount>(*i)) {
      ...
   }
}
```

static_pointer_cast and const_pointer_cast are also supported.

- There is no reinterpret_pointer_cast.

Guideline

Avoid casts down the inheritance hierarchy.

Model "has-a" or "is-implemented-in-terms-of" Through Composition

Composition is where a class contains instances of other classes:

```
class Address { ... };                   // where someone lives

class PhoneNumber { ... };

class Person {
public:
  ...
private:
  std::string name;                      // contained object
  Address address;                       // ditto
  PhoneNumber voiceNumber;               // ditto
  PhoneNumber faxNumber;                 // ditto
};
```

A common synonym for composition is *aggregation*.

The Meanings of Composition

Composition means either:

- "has-a"
  - A person *has a* name, address, voice telephone number, etc.
    - It's not the case that a person *is a* name, address, etc.
  - "has-a" refers to the software's *problem domain*
- "is-implemented-in-terms-of"
  - A Person object is implemented in terms of a string object, an Address object, etc.
  - "is-implemented-in-terms-of" refers to the software's *implementation domain*

isa Versus Is-Implemented-In-Terms-Of

Assume you want to write a Set class:

- A collection without duplicates

You naturally turn to the standard C++ library:

- It contains a set class template:
- Reuse is a wonderful thing

Problem:

- std::set typically implemented as a balanced tree.
  - Allows set to achieve its performance guarantees
  - But overhead is 3-4 words/element.
    - Pointers to parent and children; color of node.
- You don't want this much overhead.
- You decide you must write your own class template

isa Versus Is-Implemented-In-Terms-Of

Reuse is still a wonderful thing:

- Another standard template is list.
  - Only 2 words/element overhead.
    - (Nonstandard) slist would be only 1 word/element.
- A set can be represented as a list

You decide to reuse the existing list code:

- Each Set object will be represented as a list object:
  - In point of fact, each Set object will be a list object
  - Public inheritance seems natural

```
template<typename T>
class Set: public std::list<T> {
 ...
};
```

isa Versus Is-Implemented-In-Terms-Of

But there is a problem:

- For an isa relationship to hold, *everything* applicable to the base class must also be applicable to the derived class:
  - Remember, isa corresponds to *substitutability*
- A list allows duplicates:
  - Inserting the same object twice yields a list with *two* copies of the object
- A set has no duplicates:
  - Inserting the same object twice yields a set with *one* copy of the object
- It is *not true* that a set isa list:
  - Public inheritance is therefore inappropriate

isa Versus Is-Implemented-In-Terms-Of

The real relationship is is-implemented-in-terms-of:

```
template<typename T>
class Set {
public:
  bool member(const T& item) const;

  void insert(const T& item);
  void remove(const T& item);

  std::size_t size() const;

private:
  std::list<T> rep;                    // representation for a set
};
```

isa Versus Is-Implemented-In-Terms-Of

Implementation is straightforward, e.g.,

```
template<typename T>
inline std::size_t size() const
{
   return rep.size();
}
```


Guideline

Model "has-a" or "is-implemented-in-terms-of" through composition.

Use Private Inheritance Judiciously

Private inheritance does not mean "isa":

```
class Person { ... };

class Student:                        // this time we use
  private Person { ... };             // private inheritance

void dance(const Person& p);          // anyone can dance

void study(const Student& s);         // only students study

Person p;                             // p is a Person
Student s;                            // s is a Student

dance(p);                             // fine, p is a Person

dance(s);                             // error! a Student
                                      // isn't a Person
```


The Behavior of Private Inheritance

- No implicit derived-to-base conversions:
  - Not for objects
  - Not for pointers
  - Not for references
- Members inherited from private base classes become private in the inheriting class

```
class Person {
public:
  int age() const;
};

class Student: private Person { ... };

Student s;

std::cout << s.age();              // error!
```

The Meaning of Private Inheritance

Private inheritance means *is-implemented-in-terms-of*:

- It is inheritance of implementation *only*
- It is purely an implementation technique

Composition also means is-implemented-in-terms-of:

- Use composition when you can
- Use private inheritance when you must:
  - To redefine virtual functions
  - To get access to protected members
  - When space is tight and the empty base optimization is possible

A Generic Stack Class for Pointers

Consider a Stack class template:

```
template<typename T>
class Stack {
public:
   Stack();
   ~Stack();
   void push(const T& object);
   T pop();

private:
   struct StackNode { ... };            // nodes in a linked list

   Stack(const Stack& rhs);
   Stack& operator=(const Stack& rhs);

   StackNode *top;
};
```

Each different type will yield a new class:

- This may result in a lot of duplicated code
- You may not be able to afford this kind of code bloat

A Generic Stack Class for Pointers

A class using void* pointers can implement any kind of (pointer) stack:

```
class GenericPtrStack {
public:
  GenericPtrStack();
  ~GenericPtrStack();

  void push(void *object);
  void * pop();

private:
  struct StackNode {
    void *data;                              // data at this node
    StackNode *next;                         // next node in list
  };

  GenericPtrStack(const GenericPtrStack& rhs);              // prevent
  GenericPtrStack& operator=(const GenericPtrStack& rhs);  // copying

  StackNode *top;                            // top of stack
};
```


A Generic Stack Class for Pointers

GenericPtrStack is good for sharing code:

```
GenericPtrStack stringPtrStack;
GenericPtrStack intPtrStack;

std::string *newString = new std::string;
int *newInt = new int;

stringPtrStack.push(newString);              // these execute
intPtrStack.push(newInt);                    // the same code
```

But it's easy to misuse:

```
stringPtrStack.push(newInt);                 // uh oh...

std::string *sp =
  static_cast<std::string*>(intPtrStack.pop());  // uh oh (reprise)...
```

Code-sharing is important, but so is type-safety:

- We want both.

Type-Safe Interfaces

We can partially specialize Stack to generate type-safe void*-based classes:

```
template<typename T>
class Stack<T*> {
public:
  void push(T *ptr) { s.push(ptr); }

  T * pop() { return static_cast<T*>(s.pop()); }

private:
  GenericPtrStack s;                       // implementation
};
```

At runtime, the cost of Stack<T*> instantiations is *zero*:

- All instantiations use the code of the single GenericPtrStack class
- All Stack<T*> member functions are implicitly inline

The cost of type-safety is *nothing*.

Type-Safe Interfaces

How force programmers to use the type-safe classes *only*?

Prevent direct use of GenericPtrStack by making everything protected:

```
class GenericPtrStack {
protected:
  GenericPtrStack();
  ~GenericPtrStack();

  void push(void *object);
  void * pop();

private:
  ...                          // same as before
};

GenericPtrStack stringStack;   // error!
GenericPtrStack intStack;      // error!
```

Type-Safe Interfaces

But now Stack<T*> won't compile:

```
template<typename T>
class Stack<T*> {
public:
  void push(T *ptr)
  { s.push(ptr); }          // error! GenericPtrStack::push
                            // is protected
  ...
private:
  GenericPtrStack s;
};
```


Type-Safe Interfaces

Private inheritance gives access to protected members:

```
template<typename T>
class Stack<T*>: private GenericPtrStack {
public:
  void push(T *objectPtr)
  { GenericPtrStack::push(objectPtr); }

  T * pop()
  { return static_cast<T*>(GenericPtrStack::pop()); }
};
```

Net result:

- Maximal type safety
- Maximal efficiency

How did we get here?

- void* Pointers
- Templates
- Private Inheritance
- Inlining
- Protected Members

Guideline

Use private inheritance judiciously.

(Optionally skip to Slide #81)

Use Multiple Inheritance Judiciously

MI leads to new complexities. One example is ambiguity:

```
class Lottery {
public:
  virtual int draw();
};

class GraphicalObject {
public:
  virtual int draw();
};

class LotterySimulation: public Lottery,
                         public GraphicalObject {
  ...                                  // doesn't declare draw
};

LotterySimulation *pls = new LotterySimulation;

pls->draw();                           // error! — ambiguous
pls->Lottery::draw();                  // fine
pls->GraphicalObject::draw();          // fine
```

Ambiguity and MI

Access restrictions cannot eliminate the ambiguity:

```
class Lottery {
private:
  virtual int draw();
};

class GraphicalObject {
public:
  virtual int draw();
};

class LotterySimulation: public Lottery,
                         public GraphicalObject { ... };

LotterySimulation *pls = new LotterySimulation;

pls->draw();                          // still ambiguous
```

This prevents program semantics from changing only due to changes in access restrictions:

- Consider swapping the accessibility of the two draw functions above

Ambiguity and MI

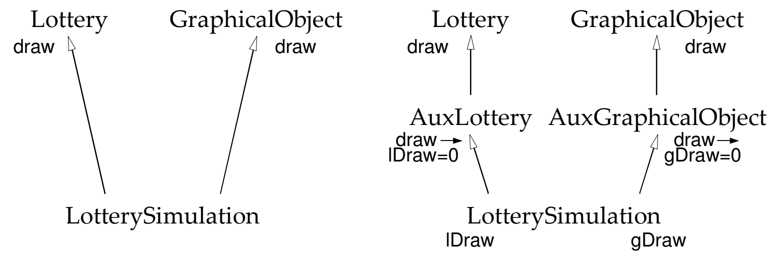
Look at these classes again:

```
class Lottery {
public:
  virtual int draw();
};

class GraphicalObject {
public:
  virtual int draw();
};

class LotterySimulation: public Lottery,
                         public GraphicalObject { ... };
```

How can LotterySimulation redefine *both* versions of draw?

Ambiguity and MI

The standard solution is to add a new pair of classes:



Before　　　After

Each class redefines draw to call a *new* function:

- The new function is declared pure virtual
- Concrete subclasses like LotterySimulation must therefore redefine it

Ambiguity and MI

In C++, it looks like this:

```
class AuxLottery: public Lottery {
public:
   virtual int lotteryDraw() = 0;
   virtual int draw() { return lotteryDraw(); }
};

class AuxGraphicalObject: public GraphicalObject {
public:
   virtual int graphicalObjectDraw() = 0;
   virtual int draw() { return graphicalObjectDraw(); }
};

class LotterySimulation: public AuxLottery,
                         public AuxGraphicalObject {
public:
   virtual int lotteryDraw();
   virtual int graphicalObjectDraw();
};
```

Ambiguity and MI

It works like this:

```
LotterySimulation *pls = new LotterySimulation;

Lottery *pl = pls;
GraphicalObject *pgo = pls;

pl->draw();               // Lottery::draw →
                          // AuxLottery::draw →
                          // AuxLottery::lotteryDraw →
                          // LotterySimulation::lotteryDraw

pgo->draw();              // GraphicalObject::draw →
                          // AuxGraphicalObject::draw →
                          // AuxGraphicalObject::graphicalObjectDraw →
                          // LotterySimulation::graphicalObjectDraw
```

Net effect:

- Lottery::draw has been renamed lotteryDraw
- GraphicalObject::draw has been renamed graphicalObjectDraw

Ambiguity and MI

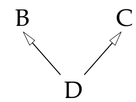
Evaluation of this technique:

- It works
- Requires the addition of two new classes:
  - ➡ They correspond to nothing in the application domain
  - ➡ They correspond to nothing in the implementation domain
- Requires a “clever” combination of pure virtual and simple virtual functions
- Provides evidence that even redefining virtual functions can become complicated in the presence of MI

Virtual Base Classes

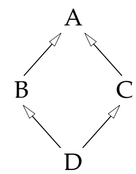
MI hierarchies that start out like this,

```
class B { … };
  class C { … };
  class D: public B, public C { … };
```



sometimes mutate into something like this:

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



The inevitable question: should A be a virtual base class?

The inevitable answer: yes.

Virtual Base Classes

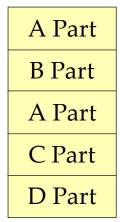
Problems:

- When B and C are written, it may not be clear that A should be a virtual base:
  - D may not have been conceived of yet
- If B and C fail to declare A as a virtual base, it may not be possible to change their declarations:
  - The header files declaring B and C may be read-only
  - The recompilation impact on other clients may be too great

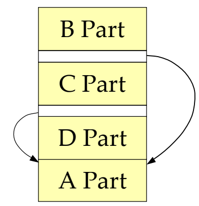
Virtual Base Classes

Declaring all bases virtual has problems, too:

- Virtual bases usually incur both size and speed penalties:



Common memory layout of a D object where A is a nonvirtual base class



Possible memory layout of a D object where A is a virtual base class

Virtual Base Classes

It seems that choosing the right base classes to declare virtual leads to a need for clairvoyance:

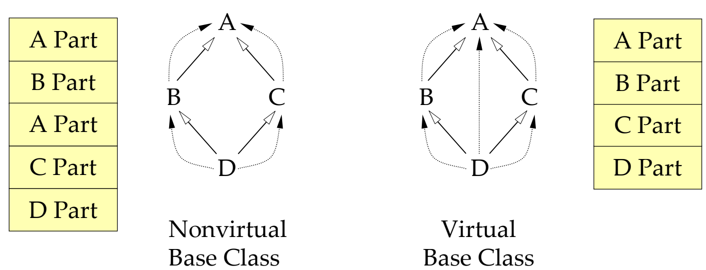
- This is not the same as the choice between virtual and nonvirtual functions:
  - ➡ A pure virtual function means something
  - ➡ An impure virtual function means something
  - ➡ A nonvirtual function means something
  - ➡ The decision between the above can be made within the context of a single class
- The decision to make a base class virtual can be made only in the context of a hierarchy of classes:
  - ➡ But the hierarchy is subject to change!

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Initialization of Virtual Base Classes

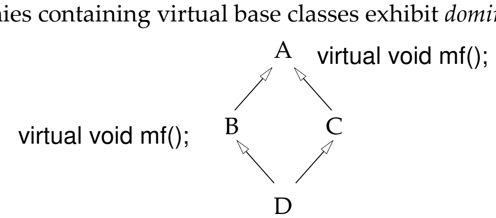
Initialization of virtual bases is different from initialization of nonvirtual bases:

- Nonvirtual bases are initialized by their immediate descendants
- Virtual bases are initialized by their most distant descendants:
  - ➨ Derived classes must know if they indirectly inherit from a virtual base
  - ➨ The class initializing a virtual base changes as the hierarchy changes



Dominance (Skip this slide)

Only hierarchies containing virtual base classes exhibit *dominance*:




```
D *pd = new D;
pd->mf();                        // A::mf or B::mf?
```

This call is:

- Ambiguous if A is a nonvirtual base
- To B::mf if A is a virtual base:
  - ➨ This is *correct behavior* and makes interface-based programming work.
  - ➨ But it's counterintuitive to the uninitiated:

```
C *pc = new D;
pc->mf();                 // calls B::mf!
```

Software Evolution and MI

Sometimes MI can seem to offer a solution to a problem that really calls for a fundamental redesign.

For example, consider a hierarchy for representing cartoon characters:

- All characters can dance and sing
- Different characters do these things differently
- The default behavior is to do nothing

This is easily expressed in C++:

```
class CartoonCharacter {
public:
  virtual void dance() {}
  virtual void sing() {}
  ...
};
```


Software Evolution and MI

Here's one cartoon character:

```
class Grasshopper: public CartoonCharacter {
public:
  virtual void dance();                 // definition is elsewhere
  virtual void sing();                  // definition is elsewhere
  ...
};
```

Later you decide to add another kind of character:

```
class Cricket: public CartoonCharacter {
public:
  virtual void dance();
  virtual void sing();
  ...
};
```

As you get ready to implement Cricket, you realize:

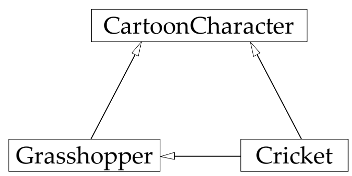
- Its functions are similar to those of Grasshopper; much of the code can be reused
- Grasshopper code needs to be tweaked for it to work for Cricket

Software Evolution and MI

Your plan: privately inherit the implementation of Cricket from Grasshopper!

```
class Cricket: public CartoonCharacter,
               private Grasshopper {
public:
  virtual void dance();
  virtual void sing();
  ...
};
```

The design looks like this:



Software Evolution and MI

Of course, Grasshopper must be modified to allow the tweaking:

```
class Grasshopper: public CartoonCharacter {
public:
  virtual void dance();
  virtual void sing();

private:
  virtual void danceCustomization1();
  virtual void danceCustomization2();
  ...
};
```

Software Evolution and MI

Dancing for grasshoppers is now defined like this:

```
void Grasshopper::dance()
{
  perform common dancing actions;
  danceCustomization1();
  perform more common dancing actions;
  danceCustomization2();
  perform final common dancing actions;
}
```

Grasshopper singing is orchestrated similarly.

This is a manifestation of the *Template Method* design pattern.

Software Evolution and MI

The Cricket class must redefine these new virtual functions:

```
class Cricket: public CartoonCharacter,
               private Grasshopper {
public:
  virtual void dance() { Grasshopper::dance(); }
  virtual void sing() { Grasshopper::sing(); }
private:
  virtual void danceCustomization1();
  virtual void danceCustomization2();
  ...
};
```

This will work fine.

Software Evolution and MI

The new design is needlessly complex:

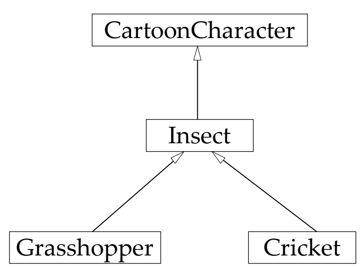
- It uses multiple inheritance
- MI is more complicated than SI
- SI will suffice in this case

Where is the flaw?

- It is not really true that Cricket is implemented in terms of Grasshopper
- Cricket and Grasshopper *share common code*
- Code sharing is represented by a *common base class*

Software Evolution and MI

This is the proper architecture:



Class Insect represents the common features of Grasshopper and Cricket objects.

Software Evolution and MI

```
class CartoonCharacter { ... };

class Insect: public CartoonCharacter {
public:
  virtual void dance();                 // common code for both
  virtual void sing();                  // grasshoppers and crickets

private:
  virtual void danceCustomization1() = 0;
  virtual void danceCustomization2() = 0;
  ...
};

class Grasshopper: public Insect {
private:
  virtual void danceCustomization1();
  virtual void danceCustomization2();
  ...
};

class Cricket: public Insect {
private:
  virtual void danceCustomization1();
  virtual void danceCustomization2();
  ...
};
```


Software Evolution and MI

This is a better design:

- Only single inheritance is used
- Only public inheritance is used
- The commonality between Grasshopper and Cricket objects is directly represented

However:

- It calls for a more extensive redesign than the MI solution:
  - A new class has to be added
  - Existing inheritance relationships have to be modified
- The MI solution can therefore seem quite attractive:
  - Resist the seduction

Guideline

Use multiple inheritance judiciously.

Inheritance and Object-Oriented Design Summary

- Make sure public inheritance models "isa."
- Prefer compile-time errors to runtime errors.
- Differentiate between inheritance of interface and inheritance of implementation.
- Never redefine an inherited nonvirtual function.
- Avoid casts down the inheritance hierarchy.
- Model "has-a" or "is-implemented-in-terms-of" through composition.
- Use private inheritance judiciously.
- Use multiple inheritance judiciously.

About Scott Meyers

Scott is a trainer and consultant on the design and implementation of software systems, typically in C++. His web site,

http://www.aristeia.com/

provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
Copyrighted material, all rights reserved.