More on Inheritance

Polymorphism: using something in multiple ways
• E.g. overloading operator+()
• E.g. std::vector<>

*Dynamic* polymorphism defers the decision about how to do something until runtime.
• C++ provides a form of function overloading that yields dynamic polymorphism
• Works (only!) through pointers or references
• Uses inheritance, meaning that polymorphism is restricted to types derived from some common base (along with the base itself)

The basic idea:

• Use inheritance, deriving from some common base type.
• Redefine common functions in derived types.
• *Construct* objects of specific types
• *Maintain* these objects using pointers or references to the base type.
• Call common member functions on these pointers or references.
• Have these calls result in calls to the “real” types of the object.
Virtual functions

```cpp
class Shape
{
public:
    Shape();
    virtual void draw();
};
class Circle : public Shape
{
public:
    Circle( int radius );
    virtual void draw();
};
class Square : public Shape
{
public:
    Square( int side );
    virtual void draw();
};
```

Virtual functions

```cpp
Shape* shapes[100]; // array of shape pointers
int numShapes = 0;

void pushShape( Shape* ps )
{
    shapes[numShapes++] = ps;
}
...
pushShape( new Circle(10) );
pushShape( new Square(5) );

for ( int i = 0; i < numShapes; i++ )
{
    shapes[i]->draw(); // call virtual function
}
```
Faking it

class Shape
{
public:
    enum Type { Circle, Square };
    Shape( Type t );
    void draw();
private:
    Type m_type;
};

Shape::draw()
{
    switch ( m_type ) {
        case Circle:
        case Square:

Why not fake it?

- Can’t extend hierarchy without disturbing base class. Makes maintenance difficult.
- All functionality is in one type, so specific functionality has no home: e.g. where do we put the radius() function?
Using virtual functions

• Once a function is virtual in some base class, it is virtual in all descendant classes, whether labeled so or not.
• Use the virtual label in derived classes to avoid confusion!

```cpp
class Shape
{
    public:
    virtual void draw();
};
class Circle : public Shape
{
    public:
    void draw();  // Still virtual, see Shape
};
```

Pure virtual functions

• A pure virtual function is one which must be implemented in derived classes.
• A class with a pure virtual function is called an “abstract base class”
• A class with a pure virtual function cannot be instantiated!

```cpp
class Shape
{
    public:
    virtual void draw() = 0;  // pure virtual
};
Shape s;  // Error, cannot create instance
```
Under the covers

- Compilers are free to implement as they see fit.
- Generally, implementation is done with static tables of function pointers in each class with virtual functions, plus a `vtbl` pointer in each object.
- Thus an object of a class with virtual functions may have an “unusual” size.
- Calling virtual functions requires a bit more time (but will still be faster than faking it).

Virtual destructors

- Destructors of base classes should be declared virtual.
- Otherwise, destruction of a derived object via a base reference or pointer is (gasp!) undefined…
- … so even the base part may not be destroyed!
- This means you can’t really safely inherit from classes with non-virtual destructors.
- Maybe just make all destructors virtual? (size issue)
- Any class with a virtual function should certainly have a virtual destructor.
Two forms of inheritance

Inheritance of implementation:
• Base class provides some useful functionality and/or data.
• Consider using encapsulation?

Inheritance of interface:
• Base class specifies common interface for a collection of types that will be used together.
• Consider using abstract base class
• A good way to really hide implementations

Pure interfaces

ishape.h:

```cpp
// Class IShape is a pure interface - no implementation
// here at all.
class IShape
{
    public:
        virtual void draw() = 0;
    ....

    // Factory functions
    static IShape* MakeCircle();
    static IShape* MakeSquare();
};
```
Safer casting with static_cast

Use static_cast when you are converting from one type to another in a way that the compiler will find acceptable (it may do some checking):

• Converting to and from void*
• Converting float to int
• Converting enum to int

```cpp
double x = 3.1414;
int x = static_cast<int>(x);

int* x = static_cast<int*>( malloc( sizeof(int)*100 ) );
```

dynamic_cast

Use dynamic_cast when you are converting up or down a type hierarchy and want runtime checking.

```cpp
Derived* dp;
Base* bp = dynamic_cast<Base*>( dp ); // surely safe

Derived* dp = dynamic_cast<Derived*>( bp );
if ( dp != NULL ) // cast worked
    ...% else // *bp wasn’t really a Derived object!!
    ...%```
reinterpret_cast

Use reinterpret_cast when you know you are doing something correct, and don’t need the compiler to check (it won’t!)

```cpp
static const unsigned int IO_PORT_A = 0x00fc;
Port* ioA = reinterpret_cast<Port*> IO_PORT_A;
```

const_cast

Use const_cast when you want to “cast away constness” on some object. (Try to use mutable instead)

```cpp
void foo( const Account& acct )
{
    Account& back_door = const_cast<Account&> acct;
    back_door.some_NONCONST_function();
}
```
Multiple Inheritance

- Inheriting from two or more classes provides the functionality of all of them.
- Ambiguity between parents must be resolved...
- ... with qualifiers, just like ambiguity between parent and child.
- What if parents are really siblings!?

```cpp
class Parent1 : public Base
class Parent2 : public Base
class Child : public Parent1, public Parent2
```

![Diagram of multiple inheritance](image)
Multiple Inheritance

- Using virtual inheritance, we can get this, instead:

```cpp
class Parent1 : virtual public Base
class Parent2 : virtual public Base
class Child : public Parent1, public Parent2
```

Pure “mixin” interfaces

**ierrkeep.h:**

```cpp
class IErrorKeeper
{
public:
    virtual string getLastError() = 0;
    ....
    // No factory functions, no one makes these!
};
```

**ishape.h:**

```cpp
class IShape: virtual public IErrorKeeper
```

**errkeep.h:**

```cpp
class ErrorKeeper: virtual public IErrorKeeper
```

**shape.h:**

```cpp
class Shape: public ErrorKeeper, public IShape
```
Overriding non-virtual functions

• It seems like these should do the same thing:

Der d;
Der* dp = &d;
Base* bp = &d;
dp->foo();
bp->foo();

• From a theoretical standpoint, we’d like to use public inheritance to model ISA, and use virtual functions to deal with the exceptions.

Inheritance advice

• Think about interface and implementation separately.
• For real modularity, provide a pure interface and a separate function or class for producing instances.
• Always make destructors virtual
• Don’t redefine non-virtual functions.
• Don’t change default values in any redefined function. (default values are statically bound)