Object-Oriented Design (cont).

Object-Oriented Programming in C++

Interface vs. Implementation (EM[#34])

- When using public inheritance in C++ you can inherit both interface and implementation.
  - Interfaces are **always** inherited
    - Makes sense because public inheritance models “is-a”
  - Implementation **may** or **may not** be inherited.
    - Supply **default behavior**, but **can be overridden**.
- In some other OO languages these two uses are separated
  - e.g. Class vs. Interface in Java.

Interface vs. Implementation (cont.)

- Inherit interface **only**
  - Declare an **abstract base class**, using only **pure virtual functions**.
- Inherit **both** interface and implementation
  - Declare a base class providing both **declarations** and **definitions** for functions.
- What if **some** functions are pure virtual but other have definitions?
  - Class still **abstract**; provides an **interface**, **but** also **some default implementations**.

Interface vs. Implementation (cont.)

- Pure virtual functions provide interface only

```
class Shape {
public:
    virtual void draw() const = 0;
};
```

- Simple (non-pure) virtual functions provide both interface and default implementation.

```
class Shape {
public:
    virtual void draw() const;
};

void Shape::draw() const { ... }
```

Interface vs. Implementation (cont.)

- Better to separate the two roles or not?
  - Different views.
- However,
  - Can be dangerous to provide a default implementation, that is implicitly called.
  - Explicit

Interface vs. Implementation: Airplane Example EM[#34]

```
class Airplane {
public:
    virtual void fly();
};

void Airplane::fly() {
    // default fly implementation.
}

class ModelA: public Airplane { };
class ModelB: public Airplane { };

int main()
{
    Airplane *pa = new ModelA();
    Airplane *pb = new ModelB();
    pa->fly();
    pb->fly();
    delete pb; delete pa;
    return 0;
}
```

Now need to add a new airplane to the fleet? The newest model C, but flies somewhat differently.

Airplane Example (implicit default behavior)

```
class Airplane {
public:
    virtual void fly();
};
void Airplane::fly() {
    // default implementation
}
class ModelA: public Airplane { }; // Default fly ok.
class ModelB: public Airplane { }; // Default fly ok.
class ModelC: public Airplane { }; // Forgot new fly implementation.

int main()
{
    Airplane *pa = new ModelA;
    Airplane *pb = new ModelB;
    Airplane *pc = new ModelC; // Add model C, but flies
differently.
    pa->fly(); pb->fly();
    pc->fly(); // OOPS, inadvertently calls default Airplane::fly()
    //
}
```

Sometimes we want to provide default behavior, but **not implicitly**; rather derived classes must be **explicit** about that they want to use it.

Airplane Example (explicit default behavior #1)

```
class Airplane {
public:
    virtual void fly() = 0;
protected:
    void defaultFly();
};
void Airplane::defaultFly() {
    // default implementation.
}
class ModelA: public Airplane {
public:
    virtual void fly() { defaultFly(); }
};
class ModelB: public Airplane {
public:
    virtual void fly() { defaultFly(); }
};
class ModelC: public Airplane {
public:
    virtual void fly() { /* Own implementation */ }
};
```

Must **explicitly** state it if one wants to use the provided default behavior.

However, class interface “polluted” with closely related function names (also error-prone, if wrong function called).

Airplane Example (explicit default behavior #2)

```
class Airplane {
public:
    virtual void fly() = 0;
};
void Airplane::fly() {
    // default implementation.
}
class ModelA: public Airplane {
public:
    virtual void fly() { Airplane::fly(); }
};
class ModelB: public Airplane {
public:
    virtual void fly() { Airplane::fly(); }
};
class ModelC: public Airplane {
public:
    virtual void fly() { /* Own implementation */ }
};
```

Base class can also provide its own implementation of pure virtual functions!

Pros and cons compared to approach #1?

Avoid Hiding Inherited Names (EM[#33])

```
class Base {
public:
    virtual void out( string txt ) { cout << "B:" << txt << '\n'; }
    virtual void out( int num )    { cout << "B:" << num << '\n'; }
};

class Derived : public Base {
public:
    virtual void out( string txt ) { cout << ”D:" << txt << '\n'; }
private:
    virtual void out( int num )    { cout << ”D:" << num << '\n'; }
};

int main()
{
  // Do something . . .
}
```

Public in **Base**, but private in **Derived**.

You are allowed to do this in C++, but from OO-design principles this is simply **wrong** when using public inheritance. **Why?**

Avoid Hiding Inherited Names (cont.)

```
class Base {
public:
    virtual void out( string txt ) { cout << "B:" << txt << '\n'; }
    virtual void out( int num )    { cout << "B:" << num << '\n'; }
};

class Derived : public Base {
public:
};

int main() {
    Base b;
    Derived d;
    b.out( "Hello" );
    b.out( 1  );
    d.out( "Hello" );
    d.out( 1  );
}
```

out (…) overloaded

```
B:Hello
B:1
B:Hello
B:1
```

Avoid Hiding Inherited Names (cont.)

```
class Base {
public:
    virtual void out( string txt ) { cout << "B:" << txt << '\n'; }
    virtual void out( int num )    { cout << "B:" << num << '\n'; }
};

class Derived : public Base {
public:
    virtual void out( string txt ) { cout << "D:" << txt << '\n'; }
    virtual void out( int num )    { cout << "D:" << num << '\n'; }
};

int main() {
    Base b;
    Derived d;
    b.out( "Hello" );
    b.out( 1  );
    d.out( "Hello" );
    d.out( 1  );
}
```

out (...) overridden in derived.

Overload?
Override?
Redefine?

```
B:Hello
B:1
D:Hello
D:1
```

Avoid Hiding Inherited Names (cont.)

```
class Base {
public:
    virtual void out( string txt ) { cout << "B:" << txt << '\n'; }
    virtual void out( int num )    { cout << "B:" << num << '\n'; }
};

class Derived : public Base {
public:
    virtual void out( string txt ) { cout << "D:" << txt << '\n'; }
};

int main() {
    Base b;
    Derived d;
    b.out( "Hello" );
    b.out( 1  );
    d.out( "Hello" );
    d.out( 1  );
}
```

Intend to override only one out(…) method

Hides **all** out(…) in base class.

Only match is Derived out(string), thus invalid conversion.

```
error: invalid conversion
from 'int' to 'const char*'
```

Avoid Hiding Inherited Names (cont.)

```
class Base {
public:
    virtual void out( string txt ) { cout << "B:" << txt << '\n'; }
    virtual void out( int num )    { cout << "B:" << num << '\n'; }
};

class Derived : public Base {
public:
    using Base::out;
    virtual void out( string txt ) { cout << "D:" << txt << '\n'; }
}

int main() {
    Base b;
    Derived d;
    b.out( "Hello" );
    b.out( 1  );
    d.out( "Hello" );
    d.out( 1  );
}
```

Makes all out(...) in base visible.

```
B:Hello
B:1
D:Hello
B:1
```

Avoid Hiding Inherited Names (cont.)

- Try to (if you can)

Avoid Hiding Inherited Names.

- Nonetheless, there are exceptions where doing so is beneficial:
  - e.g. in the Visitor Pattern.

Never Redefine Non-virtual Functions

(EM[#36])

```
class Base {   // Virtual functions.
public:
    virtual void out( string txt ) { cout << "B:" << txt << '\n'; }
};

class Derived : public Base {
public:
    virtual void out( string txt ) { cout << "D:" << txt << '\n'; }
};

int main() {
    Derived *pd = new Derived;
    Base    *pb = pd;
    pd->out( "Hello" );
    pb->out( "Hello" );
}
```

Makes sense. This is just as expected: the same object after all.

```
D:Hello
D:Hello
```

Never Redefine Non-virtual Functions (cont.)

```
class Base {   // Non-virtual functions.
public:
    void out( string txt ) { cout << "B:" << txt << '\n'; }
};

class Derived : public Base {
public:
    void out( string txt ) { cout << "D:" << txt << '\n'; }
};

int main() {
    Derived *pd = new Derived;
    Base    *pb = pd;
    pd->out( "Hello" );
    pb->out( "Hello" );
}
```

What? Behaves differently based on how referenced, but still the same object?
Definitely not as expected!

```
D:Hello
B:Hello
```

Never Redefine Non-virtual Functions (cont.)

- Solution?
- **Do not do it!**

Never redefine an inherited non-virtual function.

Alternative to Virtual Functions (EM[#35])

- Non-Virtual Interface (NVI) idiom
  - Special case of the *Template Method Pattern*
    - Nothing to do with templates!
- Approach:
  - Have private virtual functions and call them indirectly through a public non-virtual member functions.
  - Pros:
    - By wrapping the virtual function we can ensure proper pre- and post conditions, if needed.

Example of NVI

- Game Character
  - has properties to keep track of such as **health**, …

    ```
    class GameCharacter {
    public:
        virtual int getHealthValue() const;
        // ...
    };
    ```

  - What if we, for example, want to:
    - monitor usage of `getHealthValue()`, or
    - first process events possibly affecting the health value

  - We can ensure that this is handled in our default `GameCharacter` implementation, but **not** in derived classes overriding `getHealthValue()`.

Example of NVI (cont.)

- Can ensure this by make
  - `getHealthValue()` **public non-virtual** and use as a wrapper around a **private virtual function**.

```
class GameCharacter {
public:
    int getHealthValue() const
    {
        // .. do pre-processing, e.g., logging.
        int value = doGetHealthValue();
        // .. do post-processing ...
    }
    // ...
private:
    virtual int doGetHealthValue() const {
        // ...
    }
};
```

Example of NVI (cont.)

- NVI assumes that:
  - `getHealthValue()` is not redefined, and it should not be (non-virtual functions should not be redefined).
  - Some OO designers argue that virtual functions should **always** be private.
    - we do not (blindly) adhere to that.
- More alternatives to using public virtual functions
  - Use the Strategy Pattern

Another Example of NVI

```
class Base {
public:
    // Non-member function, cannot be virtual; instead use NVI
    friend ostream& operator<< ( ostream& os, const Base& rhs ) {
        rhs.write( os );
        return os;
    }
private:
    virtual void write( ostream& os ) const { os << "Base”; }
};

class Derived : public Base {
public:
private:
    virtual void write( ostream& os ) const { os << "Derived”; }
};

int main() {
    Base b;
    Derived d;
    cout << b << ' ' << d << '\n';
}
```

```
Base Derived
```

Never Redefine Default Parameters

(EM[#37])

```
class Shape {
public:
    enum Color { Red=0, Green=1, Blue=2 };
    virtual void draw(Color color=Red) const = 0;
};

class Rectangle : public Shape {
public:
  virtual void draw(Color color=Green) const {cout << color << '\n';}
};

class Circle : public Shape {
public:
  virtual void draw(Color color) const {cout << color << '\n';}
};

int main() {
  Shape *pr = new Rectangle();
  pr->draw();
  // Circle *pc = new Circle(); // cannot be declared like this.
}
```

```
0
```

Inheritance Issues (EM[#39, #40])

- Use private inheritance judiciously
  - Rather use composition where possible, and private inheritance only where not.

- Use multiple inheritance judiciously
  - Can lead to ambiguity issues.
  - Legitimate uses:
    - e.g. Inherit interfaces

Summary

- Interface vs. implementation
  - In C++ can inherit as interface only, or interface and implementation
    - Possibly mixed (method level).
- Avoid hiding inherited names
- Never redefine non-virtual functions
- Alternatives to virtual functions
  - NVI idiom.
- Never redefine default parameters