Inheritance

Basic Inheritance Syntax

class Super
{
    public:
        Super();
        void someMethod();

    protected:
        int mProtectedInt;

    private:
        int mPrivateInt;
};

class Sub : public Super
{
    public:
        Sub();
        void otherMethod();
};
Using Subclasses – Just like normal classes!

Sub mySub;

mySub.someMethod(); // b/c Sub is a Super
mySub.otherMethod(); // because Sub declares it

Super mySuper;
mySuper.otherMethod(); // BUG! Super is not a Sub
Pointers and References with Subclasses

• Can point/refer to a class or any of its subclasses

• Can only (directly) access methods/members of the declared class

• Mechanism that allows polymorphism (later)

// Create a sub, and store it in a super pointer.
Super* superPointer = new Sub();
What I Can Do As a Subclass?

- Define my own methods and members.

- Access public and protected data of my superclass as though they were my own, but I can't access private data:

```cpp
void Sub::otherMethod()
{
    cout << "I can access superclass' mProtectedInt."
    cout << "Its value is " << mProtectedInt; // OK!
    cout << "But not private: " << mPrivateInt; // BUG!
}
```

- Change the behavior of my superclass' methods by overriding them.
The **virtual** Keyword

In order to be properly overridable, a method must be virtual in the superclass. My suggestion: **make everything virtual except ctors and static methods!**

```cpp
class Super
{
    public:
        Super();

        virtual void someMethod(); // now with overridability!

    protected:
        int mProtectedInt;

    private:
        int mPrivateInt;
};
```
Overriding a Method

// Super.cpp
void Super::someMethod()
{
    cout << "This is Super's someMethod()." << endl;
}

// Sub.h
class Sub : public Super
{
    public:
        Sub();

        virtual void someMethod(); // Override Super someMethod()
        virtual void otherMethod();
};
// Sub.cpp
void Sub::someMethod()
{
    cout << "This is Sub's someMethod()." << endl;
}

What will each of these print?

Super mySuper;
mySuper.someMethod();

Sub mySub;
mySub.someMethod();

Sub mySub;
Super& ref = mySub;
ref.someMethod();
Slicing: Reason #28 Why Java Programmers Complain about C++

Even though a pointer or reference to a Super can actually refer to a Sub, non-pointer/ref objects will be sliced:

```java
Sub mySub;
Super assignedObject = mySub; // Assign Sub to a Super.
assignedObject.someMethod();  // Calls Super's someMethod()
```

Subclasses lose their uniqueness when cast to a superclass. They retain their uniqueness when access by a pointer or reference to the superclass.
Example of Subclassing for Code Reuse

// WeatherPrediction.h

/**
 * Predicts the weather using proven new-age techniques given the current temperature
 * and the distance from Jupiter to Mars. If these values are not provided, a guess is
 * still given but it's only 99% accurate.
 */

class WeatherPrediction
{
    public:
        virtual void setCurrentTempFahrenheit(int inTemp);
        virtual void setPositionOfJupiter(int inDistanceFromMars);
/**
 * Gets the prediction for tomorrow's temperature
 */
virtual int getTomorrowTempFahrenheit();

/**
 * Gets the probability of rain tomorrow. 1 means
 * definite rain. 0 means no chance of rain.
 */
virtual double getChanceOfRain();

/**
 * Displays the result to the user in this format:
 * Result: x.xx chance. Temp. xx
 */
virtual void showResult();

protected:
    int mCurrentTempFahrenheit;
    int mDistanceFromMars;
};
Modify this class so that:

• It works with Celsius

• The output is nicer

// MyWeatherPrediction.h
class MyWeatherPrediction : public WeatherPrediction {
    public:
        virtual void setCurrentTempCelsius(int inTemp);
        virtual int getTomorrowTempCelsius();
        virtual void showResult();    // override

    protected:
        static int convertCelsiusToFahrenheit(int inCelsius);
        static int convertFahrenheitToCelsius(int inFahrenheit);
};
```cpp
void MyWeatherPrediction::setCurrentTempCelsius(int inTemp) {
    int fahrenheitTemp = convertCelsiusToFahrenheit(inTemp);
    setCurrentTempFahrenheit(fahrenheitTemp);
}

int MyWeatherPrediction::getTomorrowTempCelsius() {
    int fahrenheitTemp = getTomorrowTempFahrenheit();
    return convertFahrenheitToCelsius(fahrenheitTemp);
}
```

The new methods *wrap* existing methods defined in the superclass. You can also add completely new, unrelated functionality in the same way.
void MyWeatherPrediction::showResult()
{
    cout << "Tomorrow's temperature will be " << 
        getTomorrowTempCelsius() << " C (" << 
        getTomorrowTempFahrenheit() << " F)" << endl;

    cout << "The chance of rain is " << (getChanceOfRain() 
        * 100) << " percent" << endl;

    if (getChanceOfRain() > 0.5) {
        cout << "Bring an umbrella!" << endl;
    }
}

This method replaces the superclass version.
Call Your Parents at Least Once a Month

Here's how C++ constructs a class:

1. If there is a superclass, construct it first.

2. Construct non-static data members in the order of declaration.

3. Execute the body of the constructor.
class Something {
    public:
    Something() { cout << "2"; }
};

class Parent {
    public:
    Parent() { cout << "1"; }
};

class Child : public Parent {
    public:
    Child() { cout << "3"; }

    protected:
    Something mDataMember;
};

int main(int argc, char** argv) {
    Child myChild;
}
Superclasses with Non-Zero-Arg Constructors

If your parent class has a zero-arg constructor, it will be called automatically. If it doesn't, or if you want to use a different parent constructor, call it on the initializer list:

```cpp
// Super.h
class Super
{
    public:
        Super(int i);
};

// Sub.cpp
Sub::Sub() : Super(7)
{
    // Do Sub's other initialization here.
}
```
You can also pass arguments of one constructor into another:

```cpp
Sub::Sub(int i) : Super(i)
{
    // Do Sub's other initialization here.
}
```

However, you should not pass a data member, because it won't be initialized:

```cpp
Sub::Sub() : Super(mSubDataMember) // BUG!
{
    // Do Sub's other initialization here.
}
```
Destructors, Parents, and Children

- Destruction happens in the reverse order of construction.

- Destructors should *always* be virtual!

- Each class only cleans up its own data. Let your parent clean up theirs.
Referring to Parent Data

- Implied (see earlier weather example)
- Explicit

```cpp
class Book
{
    public:
        virtual string getDescription() { return "Book"; }
};

class Paperback : public Book
{
    public:
        virtual string getDescription() {
            // avoid infinite loop!
            return "Paperback " + Book::getDescription();
        }
};
```
class Romance : public Paperback
{
    public:
        virtual string getDescription() {
            return "Romance " + Paperback::getDescription();
        }
};

class Technical : public Book
{
    public:
        virtual string getDescription() {
            return "Technical " + Book::getDescription();
        }
};
Casting and Subclassing

Upcasting:

Super mySuper = mySub;  // SLICE!
Super& mySuper = mySub;  // No slice!

Downcasting, if you absolutely must:

void presumptuous(Super* inSuper) {
    Sub* mySub = static_cast<Sub*>(inSuper);  // BAD!
    // Proceed to access Sub methods on mySub.
}

void lessPresumptuous(Super* inSuper) {
    Sub* mySub = dynamic_cast<Sub*>(inSuper); // OK!
    if (mySub != NULL) {
        // Proceed to access Sub methods on mySub.
    }
}

Crosscasting == BAD!
Inheritance for Polymorphism

Polymorphism lets you use objects with a common parent interchangeably.

class SpreadsheetCell {
    public:
        SpreadsheetCell();
        virtual void set(double inDouble);
        virtual void set(const std::string& inString);
        virtual std::string getString();

    protected:
        static std::string doubleToString(double inValue);
        static double stringToDouble(const std::string& inString);
        double mValue;
        std::string mString;
};
Class Hierarchy Design Approaches

StringSpreadsheetCell

DoubleSpreadsheetCell

SpreadsheetCell

StringSpreadsheetCell

DoubleSpreadsheetCell
The SpreadsheetCell hierarchy is *polymorphic* because:

- Both subclasses support the same interface (set of methods) defined by the base class.

- Code that uses a SpreadsheetCell can call any of its methods without knowing (or caring) whether the StringSpreadsheetCell implementation will be used or the DoubleSpreadsheetCell will.

- Through virtual methods, the appropriate implementation will be called, as long as we use pointers or references.

- A collection (vector?) of SpreadsheetCells can be created that stores both String cells and double cells.
Defining the Base Class

- All cells need to be able to set with a string and return a string, even cells that actually contain doubles.

- You can never create a generic SpreadsheetCell. It must always be a StringSpreadsheetCell or a DoubleSpreadsheetCell.

class SpreadsheetCell  // an "abstract class"
{
    public:
        SpreadsheetCell() {};
        virtual ~SpreadsheetCell() {};

        virtual void set(const std::string& inString) = 0;
        virtual std::string getString() const = 0;
    
};
class StringSpreadsheetCell : public SpreadsheetCell {
    public:
        StringSpreadsheetCell();
        virtual void set(const std::string& inString);

        virtual std::string getString() const;
    protected:
        std::string mValue;
};

StringSpreadsheetCell::StringSpreadsheetCell() : mValue("#NOVALUE"){}

void StringSpreadsheetCell::set(const string& inString) {
    mValue = inString;
}

string StringSpreadsheetCell::getString() const {
    return mValue;
}
class DoubleSpreadsheetCell : public SpreadsheetCell {
    public:
        DoubleSpreadsheetCell();
        virtual void set(double inDouble);
        virtual void set(const std::string& inString);

        virtual std::string getString() const;

    protected:
        static std::string doubleToString(double inValue);
        static double stringToDouble(const std::string& inValue);

        double mValue;
};
DoubleSpreadsheetCell::DoubleSpreadsheetCell(): mValue(-1) {} 

void DoubleSpreadsheetCell::set(double inDouble) 
{
    mValue = inDouble;
}

void DoubleSpreadsheetCell::set(const string& inString) 
{
    mValue = stringToDouble(inString);
}

string DoubleSpreadsheetCell::getString() const 
{
    return doubleToString(mValue);
}

Result: Each class is self-centered. StringSpreadsheetCell only worries about storing string values. DoubleSpreadsheetCell only worries about storing doubles. No more split personality!
Using Polymorphic Objects

```cpp
int main(int argc, char** argv) {
    SpreadsheetCell* cellArray[3];

    cellArray[0] = new StringSpreadsheetCell();
    cellArray[1] = new StringSpreadsheetCell();
    cellArray[2] = new DoubleSpreadsheetCell();

    cellArray[0]->set("hello");
    cellArray[1]->set("10");
    cellArray[2]->set("18");

    cout << "Array values are [" << 
    cellArray[0]->getString() << "," << 
    cellArray[1]->getString() << "," << 
    cellArray[2]->getString() << "]" << endl;
}

Grid<SpreadsheetCell*> mySpreadsheet;
```
Multiple Inheritance

class Baz : public Foo, public Bar
{
    // Etc.
};

• Baz is a Foo

• Baz is a Bar

• Baz can be cast to a Foo or a Bar
What if both superclasses have a method with the same name?

The compiler will give an error, unless you explicitly say which parent you are referring to by casting or using the following syntax:

```cpp
myBaz.Foo::someMethod();
```

Or inside of a Baz method:

```cpp
void Baz::doSomething() {
    Foo::someMethod();
}
```
What if Two Parents Have a Parent in Common?

Solution: virtual base classes (see book)
Can you…

change the return type in a method you override?  
You *can* if it's a subclass of the original return type, but you probably *shouldn't*.

change the arguments in the overridden version?  
You *can* if you use default args to make it compatible, but you probably *shouldn't*.

override a static method?  
No. It will compile, but static methods are bound to the declared type at compile time. Static methods are *not virtual*!

override some versions of a method, but not others?  
You *can*, but it's bad style. The other versions will be superficially hidden.
override protected or private methods?
   You bet! It's very common.

change the default arguments when I override?
   You can, but this has very strange consequences. You shouldn't.

make a public method private by overriding it?
   Only superficially. You can't truly prevent access to it.

make a private method public by overriding it?
   Yes, but only by creating a new public method that calls the private one.

See Chapter 10 for further discussions of these questions!
Subclassing and Well Behaved Classes

• If you write a copy constructor for the subclass, you must explicitly "chain" to the copy constructor for the superclass.

• If you write operator= for the subclass, you should call operator= on the superclass.

Sub::Sub(const Sub& inSub) : Super(inSub) {}

Sub& Sub::operator=(const Sub& inSub) {
    if (&inSub == this) {
        return *this;
    }
    Super::operator=(inSub) // Call parent's operator=.
    // Do necessary assignments for subclass.
    return (*this);
}
Run Time Type Identification (RTTI)

#include <typeinfo>

// should really be a method on Animal!
void speak(const Animal& inAnimal)
{
    if (typeid(inAnimal) == typeid(Dog&)) {
        cout << "Woof!" << endl;
    } else if (typeid(inAnimal) == typeid(Bird&)) {
        cout << "Chirp!" << endl;
    }
}

void logObject(Loggable& inLoggableObject)
{
    logfile << typeid(inLoggableObject).name() << " ";
    logfile << inLoggableObject.getLogMessage() << endl;
}