Lab #3 due date extended to today

Lab #4
  ◦ Due date: 3/5 (Thursday) at 11:59PM
  ◦ Mini project: 3/11 (Wednesday) at 11:59PM

Mid-term exam on 3/31 (Tuesday) during lecture
  ◦ Review on 3/19

Final project out this Friday (in UCMCROPS)
  ◦ Due date: 5/11 (Monday) at 11:59PM
  ◦ Presentation date: 5/12 (Tuesday) at 3:00PM

Reading assignment
  ◦ Ch. 15
The Virtual Table

Here are the vptrs and vtables created:

- Each class has 1 vptr point to its vtable.
  - Objects of the same class can share vtables.
- Each vtable keeps pointers to all virtual methods of an object.
The Virtual Table

- Example:
  - when a call to Brass::adjust is made, the compiler will say “call vptr+2”:

  - the correct pointers are stored at object creation
  - the correct methods to call can then be found at run-time even after upcasting (late binding).
Abstract Classes

When a class only presents an interface for derived classes
  ◦ it cannot be instantiated
  ◦ it sets a standard interface for extensions

How to declare an abstract class:
  ◦ just declare at least one “pure virtual method” with the “=0” syntax:

```cpp
virtual void f()=0;
```
Abstract Classes

- Example:
  - Our “Instrument” class is a good candidate for becoming an abstract class.

```
Instrument
virtual void play()
virtual char* what()
virtual void adjust()

Wind
void play()
char* what()
void adjust()

Percussion
void play()
char* what()
void adjust()

Stringed
void play()
char* what()
void adjust()

Woodwind
void play()
char* what()

Brass
void play()
char* what()
```

Let’s make this interface abstract
Abstract Classes

//: C15:Instrument5.cpp - Pure abstract base classes
class Instrument { public:
    // Pure virtual methods, all of them MUST be overridden by a derived class:
    virtual void play(note) const = 0;
    virtual char* what() const = 0;
    virtual void adjust(int) = 0;
};

class Wind : public Instrument { public:
    void play(note) const { cout << "Wind::play" << endl; }
    char* what() const { return "Wind"; }
    void adjust(int) {}
};

class Percussion : public Instrument { public:
    void play(note) const { cout << "Percussion::play" << endl; }
    char* what() const { return "Percussion"; }
    void adjust(int) {}
};

class Woodwind : public Wind { // Woodwind does not need to override all methods
    public: // since it inherits the non-abstract class Wind
    void play(note) const { cout << "Woodwind::play" << endl; }
    char* what() const { return "Woodwind"; }
};
//: C15:Instrument5.cpp - Pure abstract base classes (continue...)

```c++
int main() {
    Instrument i; // -> not possible, will generate an error!
    Wind flute;
    Percussion drum;
    Woodwind recorder;
    ...
}
```
Extending Virtual Methods

- Notice that the virtual method of the derived class (at the bottom of the hierarchy) will override the ones of the base classes

- Sometimes we want to “add” and not really to “override”
  - For that, from your overriding method, you can always explicitly call the base class implementation using the scope operator
Abstract Classes

// SysWindow provides an abstract interface for windows to interact with the system
class SysWindow {
public:
    virtual void draw ()=0;   // Notice that virtual methods may have an implementation!
    virtual int handle ( const Event& e )=0;
};

// in SysWindow.cpp:
void SysWindow::draw ()
{
    // make critical settings (but nothing to draw)
    glViewport ( ... );
    glEnable ( ... );
}

int SysWindow::handle ( const Event& e )
{
    // test if there is a UI attached (but SysWindow itself does not react to events)
    if ( user_interface_attached() ) return ui()->handle(e);
    return 0;
}
Abstract Classes

class MyWindow : public SysWindow { // MyWindow implements my application
 public:
    virtual void draw ();
    virtual int handle ( const Event& e );
};

// in MyWindow.cpp:
void MyWindow::draw ()
{
    // first call the base class settings:
    SysWindow::draw();

    // now draw what you need to draw:
    drawWindowTitleBar();
    drawWindowDecoration();
    drawWindowContents();
}

int MyWindow::handle ( const Event& e )
{
    // first let the base class check for UI events:
    if ( SysWindow::handle ( e ) ) return 1;

    // now check events that are interesting for my window:
    if ( e.type == MouseClick )
    {
        moveToTop();
        return 1;
    }
    else if ()
    {
        ...
    }

    // if event not useful:
    return 0;
}

By calling the base class methods first we are able to ADD functionality to the base class implementation
Abstract Classes

class MyWindow : public SysWindow { // MyWindow implements my application
public:
    virtual void draw ();
    virtual int handle ( const Event& e );
};

// in MyWindow.cpp:
void MyWindow::draw ()
{
    // first call the base class settings:
    SysWindow::draw();
    // now draw what you need to draw:
    ...
}

int MyWindow::handle ( const Event& e )
{
    // here we check a high-priority event that we do not
    // want the base class to handle:
    if ( e.type == UIClick ) { doSomethingElse(); return 1; }

    // ok now let the base class do its work:
    if ( notInFocus() )
        if ( SysWindow::handle ( e ) ) return 1;

    // finally check my events:
    if ( e.type == MouseClick )
        { moveToTop(); return 1; }
    ...
    // if event not useful:
    return 0;
}
Abstract Classes

class MyWindow : public SysWindow { // MyWindow implements my application
public:
    virtual void draw();
    virtual int handle ( const Event& e );
};

// in MyWindow.cpp:
void MyWindow::draw ()
{
    // first call the base class settings:
    SysWindow::draw();
    // now draw what you need to draw:
    ...
}

int MyWindow::handle ( const Event& e )
{
    // check event:
    if ( e.type == UIClick ) { doSomethingElse(); return 1; }

    // check some other events:
    if ( e.type == MouseClick )
    { moveToTop(); return 1; }
    ...
    // ok now let the base class do its work:
    return SysWindow::handle ( e );
}

Or we can give full priority to the events processed by MyWindow
Abstract Classes

// Example with more complex derivation hierarchies:

class AppRect // Generic Graphical Object
public:
    virtual void draw ()=0;
    virtual int handle ( const Event& e )=0;
};

// Here is one specific graphical object:
class RectButton : public AppRect {
public:
    void draw (); // draw a button-like object
    int handle ( const Event& e ); // respond to mouse clicks
};
Abstract Classes

// Example with more complex derivation hierarchies: (continue...)

// Here is a new specialized abstract class:
class RectData : public AppRect {
public:
    bool load ( const char* file );
    bool save ( const char* file );
    Vec computeAverage ();
    ...
    virtual void draw ()=0; // the implementation here shows the data points
    virtual int handle ( const Event& e ); // interact (pan,zoom,etc) with the data
    virtual void swapmem ( void* pt, unsigned bytes )=0; // new pure method for
    // custom memory management
};

class RectLineGraph : public RectData {
public:
    virtual void draw (); // draw a line graph from the data
    virtual void swapmem ( void* pt, unsigned bytes )=0; // may use disk swap
};

class RectBarGraph : public RectData {
public:
    ...
};

New Abstract Classes
may be created at any point
Object Slicing – passing by values

```cpp
///: C15:ObjectSlicing.cpp

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
        favoriteActivity;
  }
};

void describe(Pet p) { // Slices the object
  cout << p.description() << endl;
}

int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
}```
Overloading virtual methods

class Base {
public:
    virtual int f() const { cout << "Base::f()\n"; return 1; }
    virtual void f(string) const {}
    virtual void g() const {}
};

class Derived1 : public Base {
public:
    void g() const {} // ok, only one match for overriding
};

class Derived2 : public Base {
public:
    int f() const { cout << "Derived2::f()\n"; return 2; } // ok, overriding int Base::f()
};

class Derived3 : public Base {
public:
    void f() const { cout << "Derived3::f()\n"; } // ERROR: return type of Base::f() is different
};

class Derived4 : public Base {
public:
    int f(int) const { cout << "Derived4::f()\n"; return 4; } // Here we are NOT OVERRIDING !
};
Overloading virtual methods

Case where overloading return type is ok:

class PetFood {
public:
    virtual string foodType() const = 0;
};

class CatFood : public PetFood {
public:
    string foodType() const { return "Birds"; }
};

class Pet {
public:
    virtual string type() const = 0;
    virtual PetFood* eats() = 0;
};

class Cat : public Pet {
private:
    CatFood cf;
public:
    string type() const { return "Cat"; }
    CatFood* eats() { return &cf; } // Ok to return CatFood, because it inherits PetFood
};