Inheritance

Goals:
• Support re-use of common code and data...
• ...while allowing “natural” class designs
• Support runtime polymorphism

What is inheritance?
• A relationship between two classes
• Base/parent/superclass and Derived/child/subclass
• The existence of relationship is determined by the derived class.
• Some aspects of the relationship are controlled by the base class.
• In a sense, the derived class is a client of the base class.
• Derived classes have the data and functions of the base class.
Inheritance

class Account
{
public:
  typedef std::string Symbol;
  Account();

  bool sell( const Symbol& s, unsigned int nShares );
  bool buy( const Symbol& s, unsigned int nShares );
  bool deposit( unsigned int amount );
  bool withdraw( unsigned int amount );
private:
  int m_cashBalance;
  typedef std::map <Symbol, int > HoldingMap;
  HoldingMap m_holdings;
};

Inheritance

class GoldAccount : public Account
{
public:
  GoldAccount();
  virtual ~GoldAccount();
  bool creditInterest();
private:
  int m_marginBalance;
  std::string m_personalRepName;
};
Inheritance interactions

- Parents are oblivious to existence of children.
- Members of parent which are private are \textit{not} visible to children.
- New protection level: \texttt{protected}
- Members which are \texttt{protected} are visible to children, but not to anything else.
- Children can override member functions of parents...
- ...but parent functions are still available with explicit qualification (this is very useful).

```cpp
class Account
{
 public:
  typedef std::string Symbol;
  Account();

  bool sell( const Symbol& s, unsigned int nShares );
  bool buy( const Symbol& s, unsigned int nShares );
  bool deposit( unsigned int amount );
  bool withdraw( unsigned int amount );

 protected:
  void setCashBalance( unsigned int amount );

 private:
  int m_cashBalance;
  typedef std::map < Symbol, int > HoldingMap;
  HoldingMap m_holdings;
};
```
Constructing a derived class

• A derived class incorporates a base part...
• ...which must be constructed
• By default, the base part is constructed using the the base default constructor.
• What if there is no default constructor for the base?
• Solution: use the initializer list in the derived constructor(s).
• Note: base class construction happens first!

```cpp
class Account {
public:
    Account(int bal);
};

class GoldAccount : public Account {
public:
    GoldAccount( int bal );
private:
    int m_marginBalance;
};
GoldAccount::GoldAccount( int bal ) : Account( bal ),
    m_marginBalance( 0 )
};
```
Access levels for inheritance

```cpp
class Derived : public Base

• This means public members of Base are also public in Derived, so they can be used by everyone.
```

```cpp
class Derived : private Base

• This means all members of Base are private within Derived, so they can be used only by members of Derived.
```

```cpp
class Derived : protected Base

• This means all members of Base are protected within Derived, so they can only be used by members of Derived and members of classes derived from Derived.
• All non-public inheritance is suspect - have a good reason before doing it.
```

Inheritance and friendship

• Basically, these two ideas don’t interact.
• Being a friend of A doesn’t make you a friend of A’s child...
• ... nor of A’s parent.
Inheritance vs. encapsulation

- Two ways to be a client of an existing class A
- Have a member of type A
- Inherit from type A

*Is-a vs. Has-a*

<table>
<thead>
<tr>
<th>Employee</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>has-a</td>
<td>has-a</td>
</tr>
<tr>
<td>string Name</td>
<td>vector&lt;Employee&gt;</td>
</tr>
<tr>
<td>int Salary</td>
<td>underlings</td>
</tr>
<tr>
<td></td>
<td>is-a</td>
</tr>
<tr>
<td></td>
<td>Employee</td>
</tr>
</tbody>
</table>

Inheritance vs. encapsulation

- Inheritance automatically provides interface of base class.
- Encapsulation allows restricting the interface, but requires *republication* of needed parts.
- Usually, it’s pretty obvious which is correct.
- Class libraries often rely on inheritance (MFC)
Inheritance vs. templates

• Inheritance allows code reuse, so do templates
• Inheritance implies an asymmetric relationship between classes – template instantiations are unrelated.
• Templates support compile-time polymorphism
• Inheritance can support run-time polymorphism
• Again, it’s usually clear which one to use
• Combinations are possible, but seldom used

Things to watch out for

• Compiler-generated assignment and copy constructors chain to the base class automatically.
• If you make your own, you have to chain the base class items yourself!

    Derived& Derived::operator=( const Derived& other )
    {
        if ( &other == this )
            return *this;
        Base::operator=(other);
        ...
    }
Things to watch out for

• An object of a derived type will be implicitly converted to an object of the base type if necessary.

```cpp
class Base {}
class Derived : public Base {}

void e( Base b );
void f( Base* bp );
void g( Base& br );
void h( Derived* dp );

Derived d;
Base b;
e( d ) // OK, implicit conversion via “upcast”
f( &d ); // OK, implicit conversion
g( d ); // OK, implicit conversion
h( &b ); // Error
```

Things to watch out for

```cpp
class Base  
class Derived : public Base

void f( Base* bp );
void g( Base bp );
void h( Derived* dp );

Derived d;
Base b;
Base* bp1 = &d; // OK
f( bp1 ); // OK
g( b ); // OK
h( (Derived*) bp1 ); // Accepted, but not advised
h( (Derived*) &b ); // Accepted, but surely wrong
```
Ways to use inheritance

• Extend functionality of base class with new members.
• Provide identical functionality, but enforce additional constraints on state or inputs, use different algorithms, etc.
• Incorporate a small amount of useful behavior by deriving from a class that provides it (mix-ins).

Inheritance of mix-ins

```cpp
class ErrorKeeper
{
  public:
    std::string getLastError() const
    { return m_lastError; }
  protected:
    ErrorKeeper(){}
    // Note: bogus constness here makes this more usable
    void resetError() const { m_lastError.erase(); }
    void formatError( const char* format,...) const;
    void formatError( const std::string format,...)const;
  private:
    mutable std::string m_lastError;
};
```
Object Oriented Programming in C++
Assignment 5 - Due May 1

In this assignment we will create a game of Tic-Tac-Toe to be played at the command prompt (if you don't know the Tic-Tac-Toe rules, ask a friend). A player may be human, a dumb computer opponent or a smart computer opponent. It should be possible to play with any combination of two players (human vs. human, human vs. dumb computer, dumb computer vs. smart computer, etc.). However, because this is a demanding assignment, we don’t expect the smart computer play to be any different than the dumb computer play.

You’ll want to download the following files (see website under assignments):

- TTTGame.h
- TTTPlayerFactory.h
- TicTacToe.cpp

Part 1.

Note: In part 1 you should assume completely correct user input and you can fail in whatever way you like if you don’t get it.

1) Implement TTTPlayerFactory. The TTTPlayerFactory public methods should prompt for input and create a player of the type the user requests. std::auto_ptr is a template class that takes care of cleaning up single objects allocated with new, and has a bunch of operator overloads that make it behave in many ways like a simple pointer. It’s also transferable; i.e. when you assign an auto_ptr to another auto_ptr, the auto_ptr being assigned from becomes null and void, and the auto_ptr being assigned to now owns the underlying pointer.

2) Implement TTTGame. The TTTGame interface required by the driver is already declared, but you should feel free to add other functions as you see fit. Note that the players for a game are to be determined when the game is constructed. Subsequent Play() calls will all use the same two players.

3) Implement the TTTPlayer hierarchy. You can define this interface entirely as you see fit.

Please abide by these requirements in your implementation:

Player X should always go first.
The dumb computer opponent should just pick a random empty square.
The game should end when a player has won or when there are no empty squares, whichever comes first.
Human players will enter their square selections in the format a1 ... c3, where a1 is upper left, c3 is lower right, and c1 is upper right.
After each player's turn, scroll a few blank lines and print out the new Tic-Tac-Toe board. Do not pause or prompt before or after a computer player's turn (which means that in computer vs. computer mode the game will all dump out to the screen and be over before you can blink).

Once you have finished the implementation, you may make the smart computer opponent distinct from the dumb one. Again, this is not a requirement; you can get full credit for the assignment without doing this.

Don't waste time trying to draw a fancy Tic-Tac-Toe grid to cout. Something like this is fine:

```
  X  *
 * X O
  * *
```

Part 2.

Handle invalid user input by throwing exceptions, catching them in code that outputs an explanatory message to cerr, and prompting for the input again. Code that reaches a catch in the driver isn't going to get full credit.

Grading Policy for Assignment 5:

- no bugs - 2 points
- no memory leaks - 1 point
- code reuse - 1 point
- code clarity - 3 points
- OO design - 4 points
- grader discretion - 1 point
CSCI E-225 -- Object Oriented Programming in C++

Final Project Guidelines

Final projects are due by Midnight, Sunday, May 17th.

We have deliberately left room for lots of creativity in the final project; we want you to work on something that interests you. These guidelines are meant to help you decide how ambitious to be and suggest some pitfalls to avoid. The final project is worth 40% of you grade, which makes it equivalent to 3 1/3 homeworks. You should plan to have a final project that reflects this. That is, you don't have to write a new OS for this project, but whatever you do should be substantially more than any of the assignments you've done.

We expect that projects will fall mostly into one of two types. First, an application that does something interesting in a polished manner. Such a project will almost certainly involve writing some new classes, but they may be quite simple. Hopefully, it also makes good use of the library. For such a project, a good UI counts. A second type is a project focusing on the development of one of more complex classes. Here the emphasis would be on providing a useful and polished interface. We'd expect some kind of test application, at least, but it might be simple and less robust to user error. Of course, you can do both things if you are really ambitious.

Here are a few pitfalls to avoid. If you plan to port some existing application or library to C++, think carefully about the design before you start. Obviously, such a port could be made almost trivial, which is far from what we're looking for even if the original software was very impressive. If you are working on complex classes, try to maintain a broad vision about how they'll be used, rather than tailoring them only to one application.

Grading:

Our grading scheme is inspired by the scheme used in figure skating (or at least my perception of it). On the one hand, there are required elements. Your project should have (at least) 4 elements from the list below, worth 4 points each, for a total of 16 points. We expect just about anything you do will involve at least 4 of these things. On the other hand, there is the difficulty and quality of your project. We will score quality from 0-16 and difficulty from 0-8 and take the sum to compute the other 24 points of your score.

List of elements:

- Use of inheritance
- Creation of a new template
- Use of file I/O (note: both I and O)
- Use of some STL container template
- Use of some STL algorithm
- Overloading of 4 or more operators for a class
- Use of both public and private functions in a class
- Use of dynamic memory allocation within a class
- Use of a nested class or struct (i.e. within another class)
- Use of a non-trivial assignment operator and copy constructor
- Non-trivial interaction with user (i.e. "hit any key to exit" doesn't count)
- Use of overloading to improve interface

If you're stuck for ideas, here a few:

Clever apps:

- Text Indexer - Write an application that reads a text file and generates an index. Allow the user to specify how many lines will fit on one page, then figure out what pages a word will end up on. You might want to index only less-frequent words, or allow a list of words to exclude, etc. Write out the index as a separate file. A more ambitious, related idea is a source-code indexer. Figure out what pages functions are declared on, and what pages they are called on and build an index of this material.
- Boggle or other word game - Pick you favorite word game and implement it. Either support playing by humans or build a computer analyzer for finding good solutions. An ambitious project might do both.

Useful classes:

- Number classes - Implement a class for some kind of number. For example, infinite length integers are an interesting exercise (warning: multiplication and division are pretty tricky).
- Simulation classes - Classes that "behave" like real-world objects make a good basis for a simulator. For example, consider simulating cars and gates in a parking garage.