Inheritance and the Yahtzee program

- In version of Yahtzee given previously, scorecard.h held information about every score-card entry, e.g., fullhouse, small straight, etc.
  - Changing the .h requires recompiling all files that include it, either directly or indirectly
  - Consequences of large-scale recompiling? What about *building* large programs (word, XP, etc.)
- Changes made in several places in scorecard.cpp as well
  - String for description, code for scoring, order of entries in .h file
  - Code in different places, related, must be synchronized
- Inheritance is an answer to problem of avoiding recompiling, facilitating testing, keeping related code together

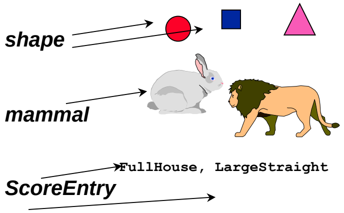
Benefits of inheritance, interfaces

- Suppose you learn about a new class WebStream that conforms to the input stream interface (cin, ifstream, …)
  - Can you write code to read words from a web page?
  - Can you write code to read lines from a web page? Chars?
- Can you use existing word counting code to read from a web page instead of from a file, e.g., in readwords.cpp?

```
void readWords(istream& input) {…}
```

- Why can we pass cin, ifstream, WebStream, etc.?
  - Inheritance, combined with late-binding
  - What type of variable according to compiler? Runtime?

Why inheritance?



User's eye view: think and program with *abstractions*, realize different, but conforming *implementations,*

don't commit to something concrete until as late as possible

- Add new shapes easily without changing much code
  - `Shape * sp = new Circle();`
  - `Shape * sp2 = new Square();`
- abstract base class:
  - interface or abstraction
  - pure virtual function
- concrete subclass
  - implementation
  - provide a version of all pure functions
- "is-a" view of inheritance
  - Substitutable for, usable in all cases as-a

Code snippets from old version

- Old version of scoreentry.h

```
class ScoreEntry
{
  public:
    enum Kind{
     ones, twos, threes, fours, fives, sixes, kind3, kind4
     fullhouse, smallstraight,largestraight,yahtzee, chance
    };
    // …
```

- Old version of scorecard.cpp

```
ScoreCard::ScoreCard()
{
    myCount = ScoreEntry::numEntries();
    for(int k=0; k < myCount; k++) {
       myEntries.push_back(
           ScoreEntry(static_cast<ScoreEntry::Kind>(k)));
    }
}
```

Yahtzee specifics

- In new version each score-card entry (almost) is a class
  - Similar entries might be one class, e.g., ones, twos, ... sixes
  - See aboveline.h, what about three/four/five of a kind?
- In ScoreCard how do create all the entries on a card?
  - Allocate an instance of each entry using new
  - Creates object pointed to by a ScoreEntry pointer
    - How can ScoreEntry pointer point at SmallStraight?
    - How can ScoreEntry pointer point at FullHouse? Nothing?
- In creating a new score-card entry, do we modify existing header files? Existing .cpp files? Benefits?
  - What must be recompiled when adding small straight?

Guidelines for using inheritance

- Create a base/super/parent class that specifies the *behavior* that will be implemented in subclasses
  - Most/All functions in base class may be virtual
    - Often pure virtual (= 0 syntax), subclasses *must* implement
  - Subclasses do not need to specify virtual, but good idea
    - May subclass further, show programmer what's going on
  - Subclasses specify inheritance using `: public Base`
    - C++ has other kinds of inheritance, stay away from these
  - Must have virtual destructor in base class
- Inheritance models "is-a" relationship, a subclass is-a parent-class, can be used-as-a, is substitutable-for
  - Standard examples include animals and shapes

Inheritance guidelines/examples

- Virtual function binding is determined at *run-time*
  - Non-virtual function binding (which one is called) determined at compile time
  - Need compile-time, or *late*, or polymorphic binding
  - Small overhead for using virtual functions in terms of speed, design flexibility replaces need for speed
    - Contrast Java, all functions "virtual" by default
- In a base class, make all functions virtual
  - Allow design flexibility, if you need speed you're wrong, or do it later
- In C++, inheritance works only through pointer or reference
  - If a copy is made, all bets are off, need the "real" object

See students.cpp, school.cpp

- Base class student doesn't have all functions virtual
  - What happens if subclass uses new `name()` function?
    - `name()` bound at compile time, no change observed
- How do subclass objects call parent class code?
  - Use class::function syntax, must know name of parent class
- Why is data protected rather than private?
  - Must be accessed directly in subclasses, why?
  - Not ideal, try to avoid state in base/parent class: trouble
    - What if derived class doesn't need data?

Inheritance (language independent)

- First view: exploit common interfaces in programming
  - Streams in C++, iterators in Tapestry classes
    - Iterators in STL/C++ share interface by convention/templates
  - Implementation varies while interface stays the same
- Second view: share code, factor code into parent class
  - Code in parent class shared by subclasses
  - Subclasses can *override* inherited method
    - Can subclasses override and call?
- Polymorphism/late(runtime) binding (compare: static)
  - Actual function called determined when program runs, not when program is compiled

Inheritance guidelines in C++

- Inherit from Abstract Base Classes (ABC)
  - one pure virtual function needed (=0)
    - Subclasses must implement, or they're abstract too
  - must have virtual destructor implemented
  - can have *pure* virtual destructor implemented, but not normally needed
- Avoid protected data, but sometimes this isn't possible
  - data is private, subclasses have it, can't access it
  - keep protected data to a minimum
- Single inheritance, assume most functions are virtual
  - multiple inheritance ok when using ABC, problem with data in super classes
  - virtual: some overhead, but open/closed principle intact

Inheritance Heuristics

- A base/parent class is an interface
  - Subclasses implement the interface
    - Behavior changes in subclasses, but there's commonality
  - The base/parent class can supply some default behavior
    - Derived classes can use, override, both
  - The base/parent class can have state
    - Protected: inherited and directly accessible
    - Private: inherited but not accessible directly
  - Abstract base classes are a good thing
- Push common behavior as high up as possible in an inheritance hierarchy
- If the subclasses aren't used polymorphically (e.g., through a pointer to the base class) then the inheritance hierarchy is probably flawed

Inheritance Heuristics in C++

- One pure virtual (aka abstract) function makes a class abstract
  - Cannot be instantiated, but can be constructed (why?)
  - Default in C++ is non-virtual or *monomorphic*
    - Unreasonable emphasis on efficiency, sacrifices generality
    - If you think subclassing will occur, all methods are virtual
  - Must have virtual destructor, the base class destructor (and constructor) will be called
- We use public inheritance, models *is-a* relationship
  - Private inheritance means is-implemented-in-terms-of
    - Implementation technique, not design technique
    - Derived class methods call base-class methods, but no "usable-as-a" via polymorphism