Why C++?

- **a better C**
  - type safe, e.g., I/O streams
  - better support for ADTs, encapsulation

- **object-oriented programming**
  - add inheritance to encapsulation
  - OO isn’t a silver bullet, but it helps in dealing with the complexity of software development

- **generic programming**
  - STL, the standard template library
Why inheritance?

- **standard shape example**
  - difficult to extend
  - access to source
  - change *all* functions
  - what is state?

- **inheritance**
  - models *is-a*  
    *Liskov substitution principle*
  - cost? (who pays?)

```cpp
class Shape
{
  public:
    enum Kind {circle, square, ...};
    double area() const;
    void rotate();
  private:
    Kind myKind;
};

double Shape::area() const
{
  switch (myKind)
  {
    case Shape::circle:
      return PI * myDim * myDim;
    case Shape::square:
      //...
  }
}
```
Inheritance

- **virtual functions provide runtime polymorphism**
  - must use pointers or references
  - can extend base class without access to source
- **in C++ there are several kinds of inheritance**
  - public, private, virtual, ...
  - multiple inheritance
- **we’ll use single, public inheritance, trying to model “is-a” as much as possible**
- **General goal: base classes are abstract, have one pure virtual function**
  - model an interface, not an implementation
  - see map hierarchy for example
Shapes: C++ features, what is “is-a”?

class Shape
{
    public:
        virtual double area() const = 0;
};

class Square : public Shape
{
    public:
        virtual double area() const {return mySide * mySide;}
    private:
        double mySide;
};

class Rectangle : public Shape
{
    public:
        virtual double area() const {return myWidth * myHeight;}
};
Designing for change (extension)

- How does inheritance help?

- How to add a new kind of shape to example?
  - what needs to be changed?
  - what do you need to know to add it?
  - how to explain so others can use effectively?

- How to write code to take advantage of changes easily?
  - think about open/closed principle
  - write code generically, allowing for possibility for change
  - heuristic: use a class whenever you think you may change your mind
grep.cc

- **Iterators**
  - internal, external
  - creation/deletion
  - use of inheritance and naming conventions

- **STL, Duke-classes**
  - what is a templated class/function?
  - how are templates instantiated?
  - drawbacks?
Data Structures, STL, Generic Programming

- common container classes/data structures
  - vector  growable array
  - map  dictionary, (search tree, hashtable)
  - set  union, intersection, membership

- STL, Standard Template Library
  - not just implementations, but way of thinking
  - little/no inheritance, lots of templates
  - algorithms and functions generalized too

- default often not-safe, implementations make heavy use of “new C++ features”
Factory Pattern

- **Consider classes for different look-and-feel styles**
  
  ```
  Circle c(100, 100, 10); c.area();
  Triangle t(100, 100, 10); t.area();
  ```

  ➤ problems, how to choose shape, when to choose

- **Does inheritance help?**
  
  ```
  Shape * s = new Circle(100, 100, 10);
  s = new Triangle(100, 100, 10);
  s->area();
  ```

  ➤ is this better? are there problems?

- **Use a Shape Factory**
  
  ```
  Shape * s = factory->makeShape(100, 100, 10);
  s->area();
  ```

  ➤ dependencies exist, but in factory
  ➤ parameters to factory can initialize shape returned
Pattern Essentials

- **Name**
  - good name provides a handle for the pattern, makes it easy to remember and use: vocabulary

- **Problem**
  - when the pattern is applicable, context, criteria to be met, design goals

- **Solution**
  - design, collaborations, responsibilities, and relationships of the classes/design elements

- **Consequences**
  - trade-offs, problems, results from applying pattern: help in evaluating applicability
Design patterns

● Add to vocabulary used to communicate and think about design

● Form part of a tool-kit we use when thinking about design

● Help find solutions to design problems
  ➤ factory
  ➤ iterator  
  ➤ proxy
  ➤ also: adapter, singleton, observer, command
  ➤ also: model-view-controller, client-server, …
  ➤ GOF, gang-of-four, Gamma, Helms, Johnson, Vlissides