Speaking C++ as a Native

Bjarne Stroustrup

AT&T Labs

http://www.research.com/~bs

Overview

- Standard C++
- Classes and templates
    - Values
    - Constraints checking
    - Resource management
    - Wrapping
- Containers and Algorithms
    - Find, sort
    - Matrices
- Class Hierarchies
    - Abstract classes
    - Algorithms on polymorphic containers

Standard C++

- C++ is a general-purpose programming language with a bias towards systems programming that
  - is a better C
  - supports data abstraction
  - supports object-oriented programming
  - supports generic programming

- A Multi-paradigm programming language

  (if you must use long words)
- The most effective styles use a combination of techniques

Standard C++

- ISO/IEC 14882 – the C++ Programming Language
  - Core language
  - Standard library
- Implementations
  - Borland, IBM, EDG, DEC, GNU, Metrowerks, Microsoft, SGI, Sun, …

    + many ports
  - All approximate the standard: portability is improving
  - Some are free
  - For all platforms:
    - BeOS, Mac, IBM, Linux/Unix, Windows, embedded systems, …

My aims for this presentation

- Here, I want to show small, elegant, examples
  - building blocks of programs
  - building blocks of programming styles
- Elsewhere, you can find
  - huge libraries
    - Foundation libraries: vendor libs, Threads++, ACE, QT, boost.org, …
    - Scientific libraries: POOMA, MTL, Blitz++, ROOT, …
    - Application-support libraries: Money++, C++SIM, BGL, …
    - Etc.: C++ Libraries FAQ: http://www.trumphurst.com
  - powerful tools and environments
  - in-depth tutorials
  - reference material

C++ Classes

- Primary tool for representing concepts
  - Represent concepts directly
  - Represent independent concepts independently
- Play a multitude of roles
  - Value types
  - Function types (function objects)
  - Constraints
  - Resource handles (e.g. containers)
  - Node types
  - Interfaces

Classes as value types

```
void f(Range& r, int n)
try {
    Range v1(0,3,10);
    Range v2(7,9,100);
    v1 = 7;          // ok: 7 is in [0,10)
    v2 = 3;          // will throw exception: 3 is not in [7,100)
    int i = v2;      // extract the value from v2
    r = 7;           // may throw exception
    v2 = n;          // may throw exception
}
catch(Range_error) {
    cerr << “Oops: range error in f()”;
}
```

Classes as value types

```
class Range {                    // simple value type
     int value, low, high;                // invariant: low <= value < high
     void check(int v) { if (v<low || high<=v) throw Range_error(); }
public:
     Range(int lw, int v, int hi) : low(lw), value(v), high(hi) { check(v); }
     Range(const Range& a) { low=a.low; value=a.value; high=a.high; }

     Range& operator=(const Range& a) { check(a.value); value=a.value; }
     Range& operator=(int a) { check(a); value=a; }

     operator int() const { return value; }     // extract value
};
```

Classes as value types: Generalize

```
template<class T> class Range {   // simple value type
    T value, low, high;                    // invariant: low <= value < high
    void check(const T& v) { if (v<low || high<=v) throw Range_error(); }
public:
    Range(const T& lw, const T& v, const T& hi)
            : low(lw), value(v), high(hi) { check(v); }
    Range(const Range& a) { low=a.low; value=a.value; high=a.high; }

    Range& operator=(const Range& a) { check(a.value); value=a.value; }
    Range& operator=(const T& a) { check(a); value=a; }

    operator T() const { return value; }      // extract value
};
```

Classes as value types

```
Range<int> ri(10, 10, 1000);
Range<double> rd(0, 3.14, 1000);
Range<char> rc(‘a’, ’a’, ’z’);
Range<string> rs(“Algorithm”, “Function”, “Zero”);
```

Templates: Constraints

```
Template<class T> struct Comparable {
    static void constraints(T a, T b) { a<b; a<=b; }   // the constraint check
    Comparable() { void (*p)(T,T) = constraints; }   // trigger the constraint check
};

Template<class T> struct Assignable { /* … */ };

template<class T> class Range
    : private Comparable<T>, private Assignable<T> {
    // …
};

Range<int> r1(1,5,10);                          // ok
Range< complex<double> > r2(1,5,10);      // constraint error: no < or <=
```

Templates: Constraints

- How can we check template parameter constraints?
  - The compiler always checks
    - late and gives poor error messages
  - The programmer can specify a check
    - Checking arbitrary constraints
      - Not just subtype/subclass relationships
      - Correspondence between several types
      - Specific properties of types
    - Readable compile-time error messages
    - No spurious code generated when constraints are met

Managing Resources

- Examples of resources
  - Memory, file handle, thread handle, socket
- General structure (“resource acquisition is initialization”)
  - Acquire resources at initialization
  - Control access to resources
  - Release resources when destroyed
- Key to exception safety
  - No object is created without the resources needed to function
  - Resources implicitly released when an exception is thrown

Managing Resources

```
//      unsafe, naïve use:

void f(const char* p)
{
    FILE* f = fopen(p,”r”);
    // use f
    fclose(f);
 }
```

Managing Resources

```
//      unsafe, naïve use:

void f(const char* p)
{
      FILE* f = fopen(p,"r");     // acquire
      // use f
      fclose(f);                  // release
  }
```

Managing Resources

```
//      naïve fix:

void f(const char* p)
{
    FILE* f = 0;
    try {
      f = fopen(p,”r”);
      // use f
    }
    catch (…) {            // handle exception
      // …
    }
    if (f) fclose(f);
 }
```

Managing Resources

```
// use an object to represent a resource (“resource acquisition in initialization”)

class File_handle {      // belongs in some support library
        FILE* p;
public:
        File_handle(const char* pp, const char* r) { p = fopen(pp,r); }
        File_handle(const string& s, const char* r) { p = fopen(s.c_str(),r); }
        ~File_handle() { if (p) fclose(p); }  // destructor
        // access functions
};

void f(string s)
{
        File_handle f(s,”r”);
        // use f
}
```

Wrapping

(simple control abstraction)

- 20+ year old problem, guard/wrap operations
  - A prefix/suffix could be lock/unlock, transaction_start/transaction_commit, trace_on/trace_off, acquire_resource/release_resource
  - **Every** major application uses some form of guard/wrap
  - Simple example of use:

    ```
    void f(X& x)
    {
            Wrap<X> xx(x,prefix,suffix);
            int n = xx->count();         // prefix(); n=xx.count(); suffix();
            xx->g(99);                   // prefix(); xx.g(99); suffix();
    }
    ```
- Optimal performance – inline prefix and suffix
- General: works for any “class X” – even pre-existing ones
- Wrap is 16 lines of standard C++

Wrapper implementation

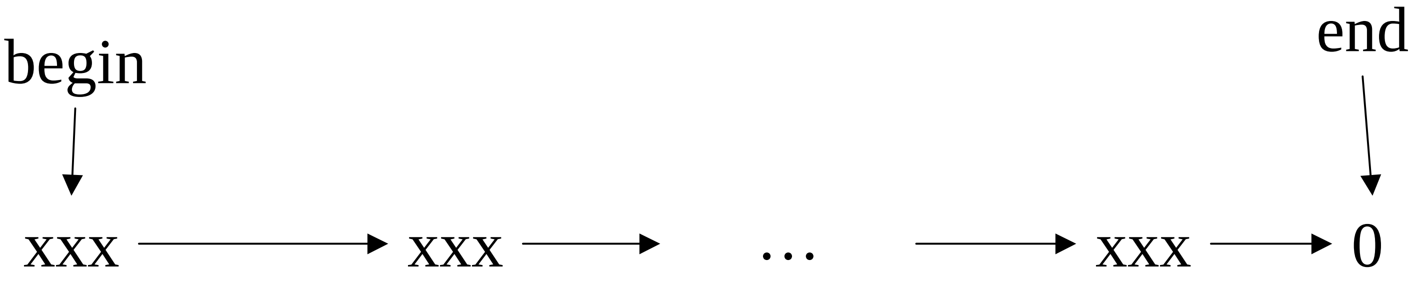
```
template<class T, class Suf> class Wrap_proxy {
     T* p;
      Suf suffix;
public:
     Wrap_proxy(T* pp, Suf s) :p(pp), suffix(s) { }
     ~Wrap_proxy() { suffix(); }
     T* operator->() { return p; }
};


template<class T, class Pre, class Suf> class Wrap {
     T* p;
     Pre prefix;
     Suf suffix;
public:
     Wrap(T& x, Pre pref, Suf s) :p(&x), prefix(pref), suffix(s) { }
     Wrap_proxy<T,Suf> operator->()
              { prefix(); return Wrap_proxy<T,Suf>(p,suffix); }
};
```

Algorithms: Genericity

- Lots of useful containers
  - vector, list, map, …
- What do you do with containers?
  - Use algorithms (Knuth, Sedgewick, …)
    - find elements, sort container, add elements, remove elements, copy container, …
    - In any container: We don’t want to re-do each of the approximately 60 algorithms for each of the approximately 12 containers

Algorithms and Containers: Iterators and sequences



begin

end

xxx → xxx → … → xxx → 0

Conventional C notation

- ++ make iterator point to next element
- * dereference iterator

```
// Pseudo code (we want to make it real code):
copy(begin,end,output)    // copy sequence to output
find(begin,end,value)     // find value in sequence
count(begin,end,value)    // count number of occurrences of value in sequence
```

Algorithms and containers

- Keeps independent concerns independent
  - Kind of “container” (sequence abstraction)
    - containers are not required to be part of a hierarchy
  - Element type
    - elements are not required to be part of a hierarchy
      - containers are nonintrusive
  - Algorithm
    - **not** member of class
  - Comparison criteria

Algorithms: find()

```
template<class In, class T>
In find(In first, In last, T val)           // find val in sequence [first,last)
{
    while (first!=last && *first!=val) // while we haven’t reach the end and haven’t found val
            ++first;                        // carry on
    return first;
}

void f(vector<int>& v, int x, list<string>& lst, string s)
{
    vector<int>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end()) { /* we found x */ }

    list<string>::iterator q = find(lst.begin(), lst.end(), s);
    if (q != lst.end()) { /* we found s */ }
}
```

Algorithms: find_if()

```
template<class In, class Pred>
In find_if(In b, In e, Pred p)
{
    while(b!=e && !p(*b))   // while we haven’t reached the end
                            // and while we haven’t found what we are looking for
          ++b;              // carry on
    return b;
}

void f(vector<string>& v, list<record>& lst, const Record& my_rec)
{
    vector<string>::iterator p =find_if(v.begin(), v.end(), Less_than<string>(“foo”));
    if (p != v.end()) { /* found: *p < “foo” */ }

    list<Record>::iterator q = find_if(lst.begin(), lst.end(), Name_eq(my_rec));
    if (q != lst.end()) { /* found: *q has the same key as my_rec */ }
}
```

Function Objects

```
class Name_eq {
        const string s;
public:
        Name_eq(const Record& r) :s(r.name) { }
        static bool operator()(const Record& r) { return r.n == s; }
};

void f(vector<string>& v, list<record>& lst, const Record& my_rec)
{
        // …
                        find_if(lst.begin(), lst.end(), Name_eq(my_rec));
        // …
}
```

Function Objects

```
template<class S> class F {          // simple, general example of function object
    S s;   // state
public:
    F(const S& ss) :s(ss) { /* establish initial state */ }
    void operator()(const S& ss) { /* do something with ss to s */ }
    operator S() { return s; }       // reveal state
};
```

Note, function objects:

- are more general than functions
- inline better than functions
- Can be generated from “natural” notation (e.g., x=y*z)

Algorithms: avoid temporaries

```
Matrix m;
Vector v, v2, v3;
// …
v = m*v2+v3;      // kindly evaluate without using temporaries
```

We need to generate: mul_add_and_assign(v,m,v2,v3);

Algorithms: Avoid temporaries

```
struct MV {        // object representing the need to multiply
    Matrix* m; Vector* v;
    MV(Matrix& mm, Vector& vv) : m(&mm), v(&vv) { }
};

MV operator*(const Matrix& m, const Vector& v)
    { return MV(m,v); }

MVV operator+(const MV& mv, const Vector& v)
    { return MVV(mv.m,mv.v,v); }

v = m*v2+v3;   // mul_add_and_assign(MVV(MV(m,v2),v3),v);
```

Algorithms: Delayed evaluation

- General technique:
    - collect information until you have everything you need
        - e.g. value, format, and stream
        - e.g. matrix, vector, …
    - optimize, vectorize, etc. given full information
    - relies on functions objects, inlining, pass by value
    - function objects often end up being templates
        - E.g. Matrix<double,Dense>

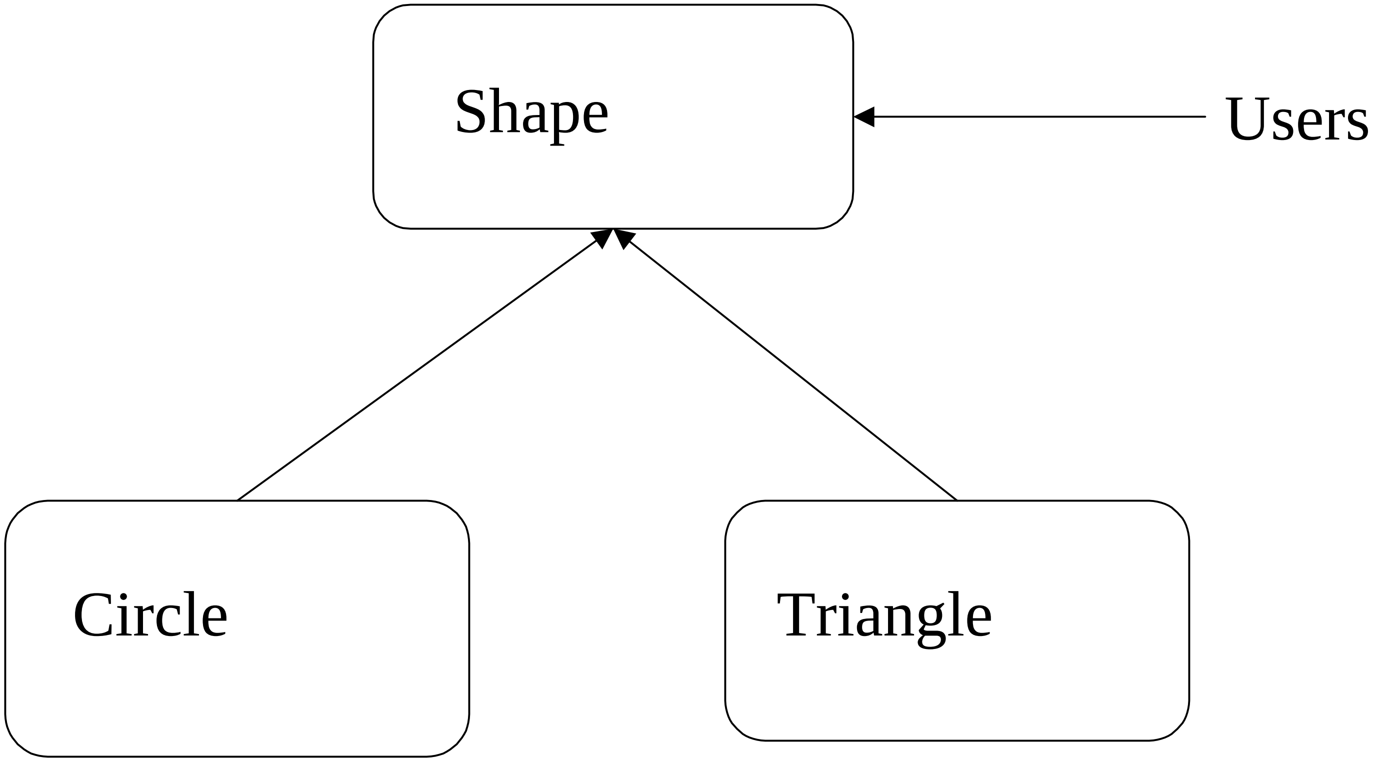
Class Hierarchies

- One way (often flawed):

```
class Shape {       // define interface and common state
     Color c;
     Point center;
     // …
public:
     virtual void draw();
     virtual void rotate(double);
     // …
};

class Circle : public Shape { double radius; /* … */ };
class Triangle : public Shape { Point a, b, c; / * … */ };
```

Class Hierarchies



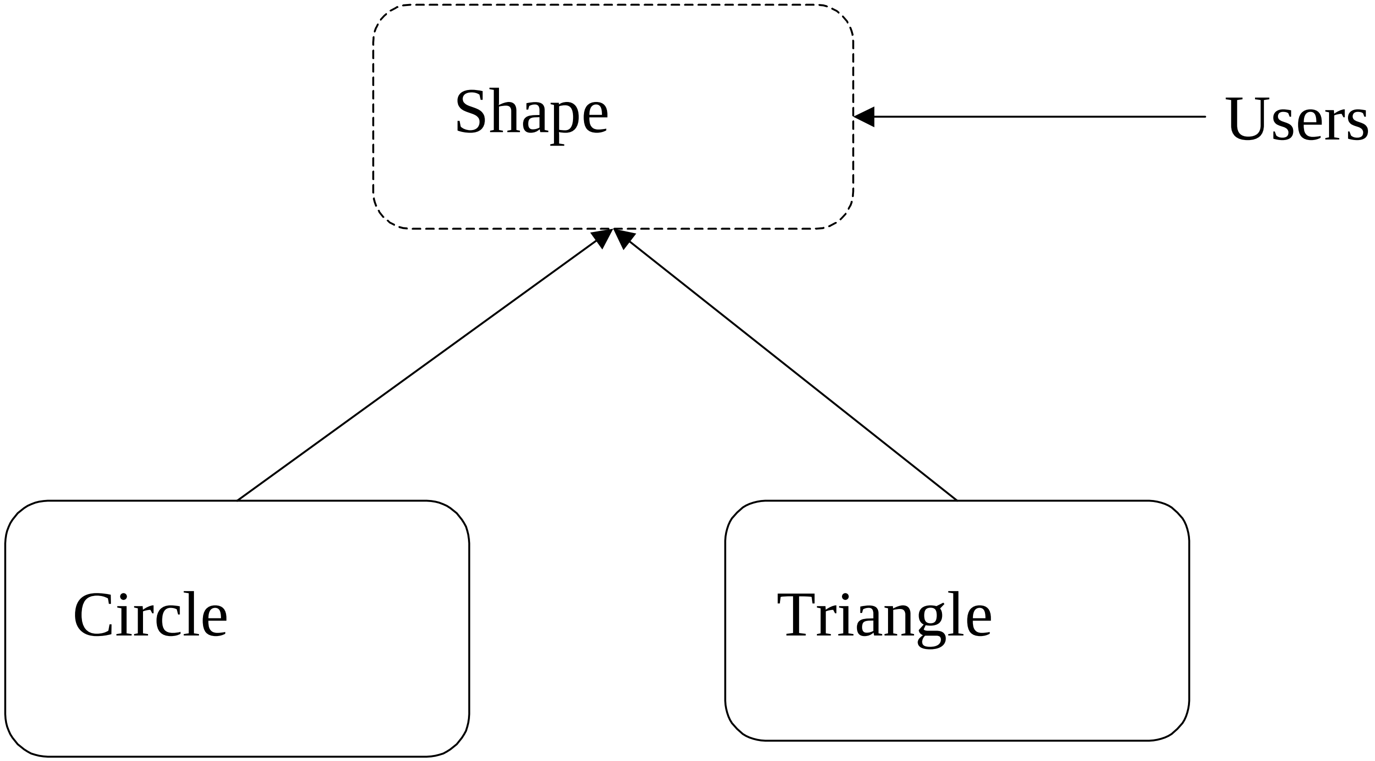
Class Hierarchies

- Another way (usually better):

```
class Shape {       // abstract class: interface only
    // no representation
public:
    virtual void draw() = 0;
    virtual void rotate(double) = 0;
    virtual Point center() = 0;
    // …
};

class Circle : public Shape { Point c; double r; Color c; /* … */ };
class Triangle : public Shape {  Point a, b, c; Color c; / * … */ };
```

Class Hierarchies



Shape

Users

Circle

Triangle

Class Hierarchies

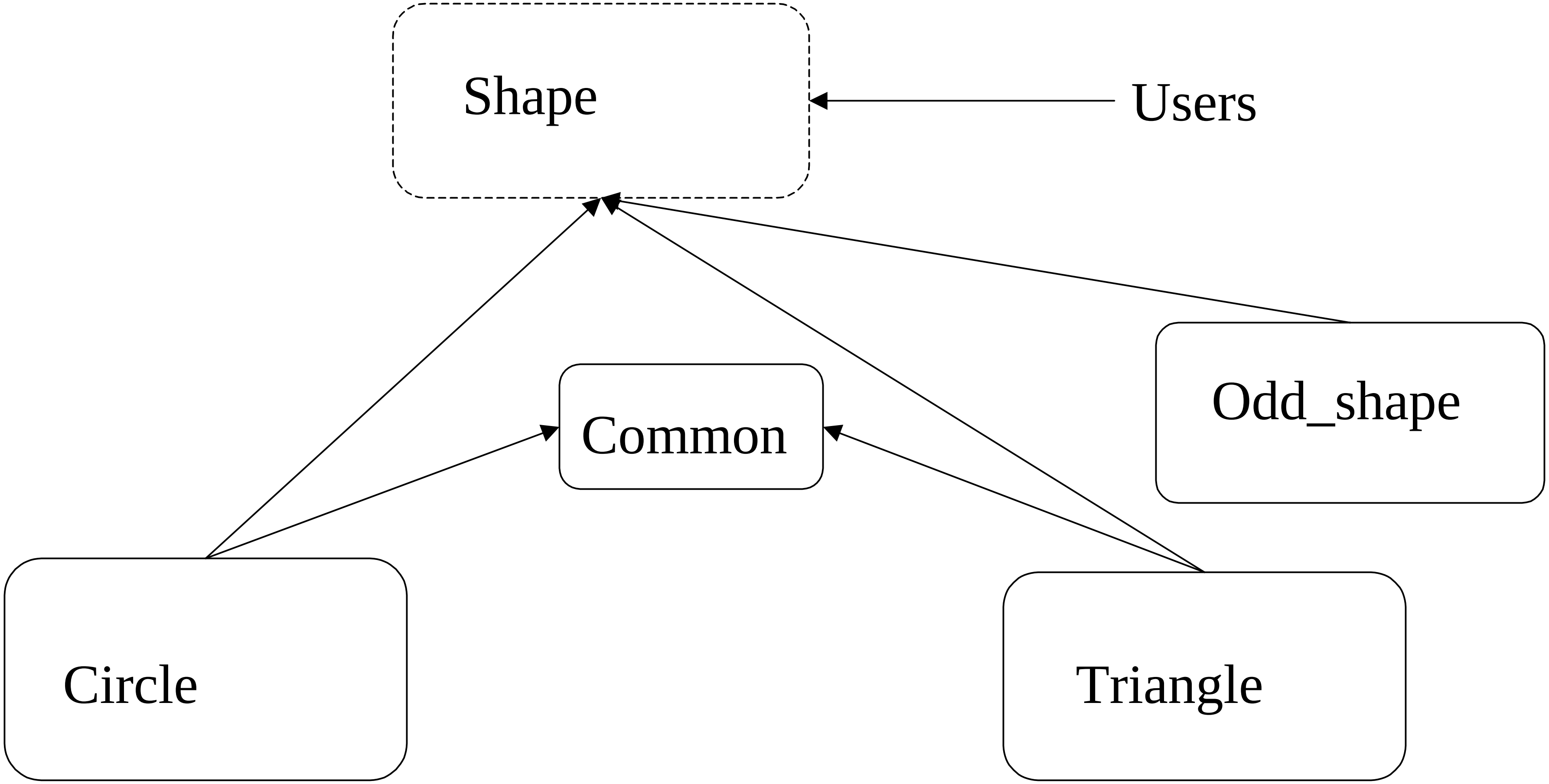
- One way to handle common state:

```
class Shape {       // abstract class: interface only
public:
     virtual void draw() = 0;
     virtual void rotate(double) = 0;
     virtual Point center() = 0;
     // …
};

class Common { Color c; /* … */ };            // common state for Shapes

class Circle : public Shape, protected Common{ Point c; double r; /* … */ };
class Triangle : public Shape, protected Common { point a, b, c; / * … */ };
class Odd_shape : public Shape { /* … */ };
```

Class Hierarchies



Algorithms on containers of polymorphic objects

```
void draw_all(vector<Shape*>& v)                          // for vectors
{
    for_each(v.begin(), v.end(), mem_fun(&Shape::draw));
}

template<class C> void draw_all(C& c)                     // for all standard containers
{
    Contains<Shape*,C>();           // constraints check
    for_each(c.begin(), c.end(), mem_fun(&Shape::draw));
}

template<class For> void draw_all(For first, For last)    // for all sequences
{
    Points_to<Shape*,For>();        // constraints check
    for_each(first, last, mem_fun(&Shape::draw));
}
```

Summary

- Think of Standard C++ as a new language
  - not just C plus a bit
  - not just class hierarchies
- Experiment
  - Be adventurous: Many techniques that didn’t work years ago now do
  - Be careful: Not every technique works for everybody, everywhere
- Prefer the C++ standard-library style to C style
  - vector, list, string, etc. rather than array, pointers, and casts
  - Small free-standing classes are essential for flexibility
  - General algorithms should be free-standing functions for flexibility
- Use abstract classes to define major interfaces
  - Don’t get caught with “brittle” base classes

More information

- Books
  - Stroustrup: The C++ Programming language (Special Edition)
  - Stroustrup: The Design and Evolution of C++
  - C++ In-Depth series
    - Koenig & Moo: Accelerated C++ (innovative C++ teaching approach)
    - Sutter: Exceptional C++ (exception handling techniques and examples)
  - Book reviews on ACCU site
- Papers
  - Stroustrup: Learning Standard C++ as a New Language
  - Stroustrup: Why C++ isn't just an Object-oriented Programming language
- Links: http://www.research.att.com/~bs
  - FAQs libraries, the standard, free compilers, garbage collectors, papers, chapters, C++ sites, interviews