

Multiparadigm Programming in Standard C++

Bjarne Stroustrup

AT&T Labs – Research

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

Abstract

- Multi-paradigm programming is programming applying different styles of programming, such as object-oriented programming and generic programming, where they are most appropriate. This talk presents simple example of individual styles in ISO Standard C++ and examples where these styles are used in combination to produce cleaner, more maintainable code than could have been done using a single style only.
- 60 minutes, incl Q&A

Overview

- **Standard C++**
 - C++ aims, standardization, overview
- **Abstraction: Classes and templates**
 - Range example
 - Resource management
- **Generic Programming: Containers and algorithms**
 - Vector and sort example
 - Function objects
- **Object-Oriented Programming: class hierarchies and interfaces**
 - Ye olde shape example
- **Multi-paradigm Programming**
 - Algorithms on shapes example
- **Implications for Large Systems**
 - Libraries, coupling, and lock-in

Standard C++

- ISO/IEC 14882 – Standard for the C++ Programming Language
 - Core language
 - Standard library
- Implementations
 - Borland, Compaq, IBM, EDG, GNU, Metrowerks, Microsoft, SGI, Sun, Etc.
 - + many ports
 - All approximate the standard: portability is improving
 - Some are free
 - For all platforms: BeOS, Mac, IBM, Linux/Unix, Windows, Symbion, Palm, embedded systems, etc.
- Probably the world's most widely used general-purpose programming language

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

Elegant, direct expression of ideas

- Declarative information is key:

```
Matrix<double,100,50,Sparse> ms;
```

```
Matrix<Quad,100,50,Dense,Triangular<upper> > mt;
```

```
Matrix<double,100,50> m; // defaults to rectangular and dense
```

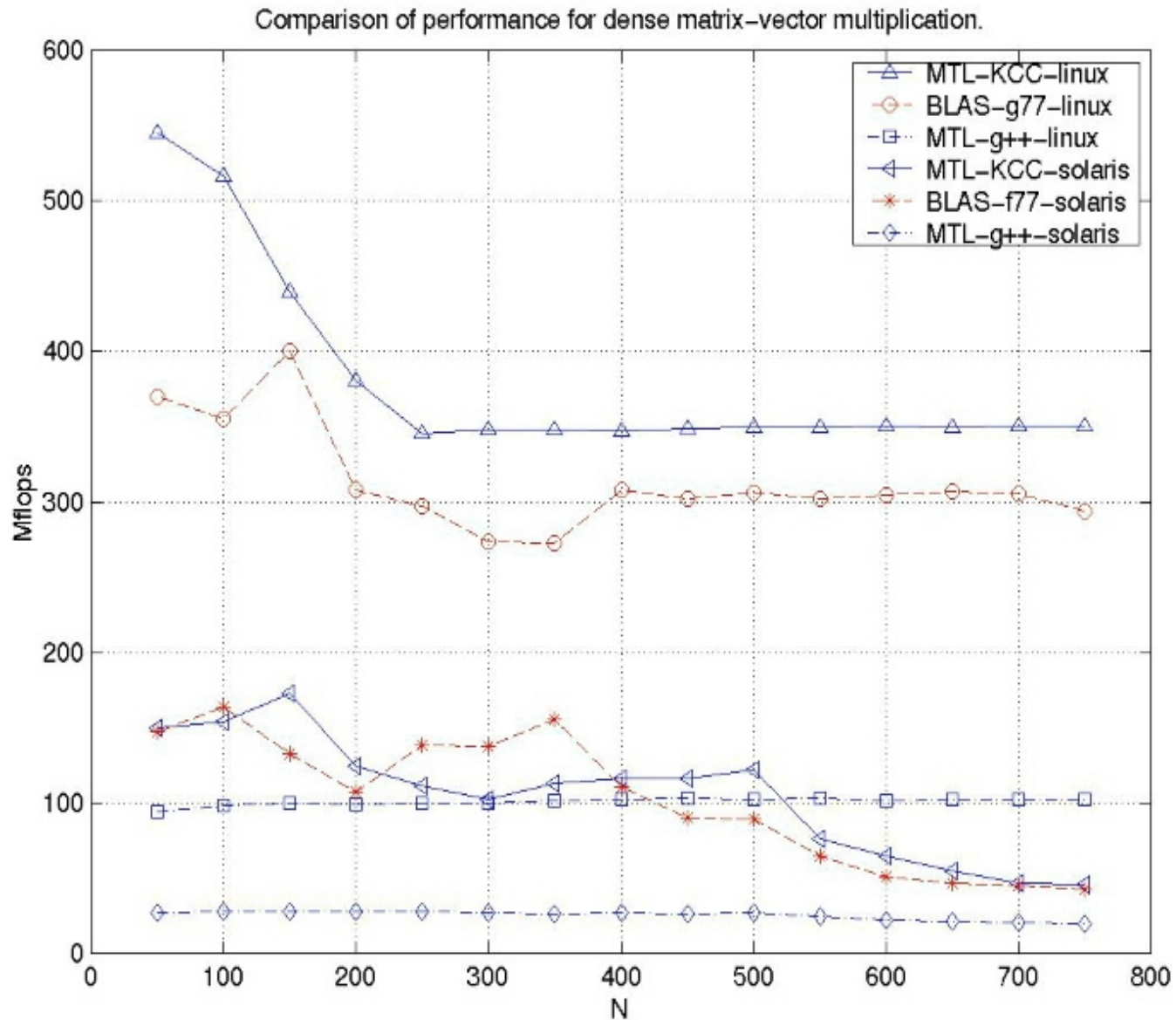
- Write expressions using “natural” notation:

```
m3 = add(mul(m,v),v2); // functional
```

```
m2 = m*v+v2; // algebraic
```

- Execute without spurious function calls or temporaries

Uncompromising performance



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++'s weakness: poor use

- C style
 - Arrays, pointers, casts, macros, complicated use of free store (heap)
- Reinventing the wheel
 - Strings, vectors, lists, maps, GUI, graphics, numerics, units, concurrency, graphs, persistence, ...
- Smalltalk-style hierarchies
 - “brittle” base classes
 - Overuse of hierarchies

Here, I focus on alternatives

- Primarily relying on abstract classes, templates, and function objects

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

- Built-in types
 - bool, char, int, float, double, unsigned char, long int, pointers, arrays, ...
- Standard-library types
 - string, vector, list, map, set, ostream, complex, priority_queue, auto_ptr, ...
- User-defined types
 - Date, Socket, hash_map, Point, Range, Real, Buffer, Line_segment, Person, ...

Classes as value types

```
void f(Range arg)          // Range: y in [x,z)
try
{
    Range v1(0,3,10);      // 3 in range [0,10)
    Range v2(7,9,100);    // 9 in range [7,100)
    v1 = v2;              // ok: 9 is in [0,10)
    v2 = v1;              // will throw exception: 3 is not in [7,100)
    v1 = v2-v1;          // ok: 9-3 is in [0,10)
    arg = v1;            // may throw exception
    v2 = arg;            // 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; return *this; }  
    Range& operator=(int a) { check(a); value=a; return *this; }  
  
    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(T v) { if (v<low || high<=v) throw Range_error(); }
public:
    Range(T lw,T v, 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; return *this; }
    Range& operator=(const T& a) { check(a); value=a; return *this; }

    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");
```

```
Range< complex<double> > rc(0,z1,100); // error: < is not defined for complex
```

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

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 <=
```

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");    // 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 every exception  
        // ...  
    }  
    if (f) fclose(f);  
}
```

Managing Resources

// use an object to represent a resource (“resource acquisition is 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
    // copy operations
    // access functions
};

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

Generic Programming

- First aim: Standard Containers
 - Type safe
 - without the need for run-time checking
 - Efficient
 - Without excuses
 - Interchangeable
 - Where reasonable
- Consequential aim: Standard Algorithms
 - Applicable to many/all containers
- General aim: The most general, most efficient, most flexible representation of concepts
 - Represent separate concepts separately in code
 - Combine concepts freely wherever meaningful

Read and sort example

```
int n;  
while (cin>>n) vi.push_back(n);    // read integers  
sort(vi.begin(), vi.end());        // sort integers
```

```
string s;  
while (cin>>s) vs.push_back(s);    // read strings  
sort(vs.begin(),vs.end());         // sort strings
```

```
template<class T> void read_and_sort(vector<T>& v) // use < for comparison  
{  
    T t;  
    while (cin>>t) v.push_back(t);  
    sort(v.begin(),v.end());  
}
```

Read and sort example

```
template<class T, class Cmp>
void read_and_sort(vector<T>& v, Cmp c = less<T>())
{
    T t;
    while (cin>>t) v.push_back(t);
    sort(v.begin(), v.end(), c);
}
```

```
vector<double> vd;
read_and_sort(vd);           // sort using the default <
```

```
vector<string> vs;
read_and_sort(vs, No_case()); // sort case insensitive
```


Generality/flexibility is affordable

- Read and sort floating-point numbers

- C: read using stdio; `qsort(buf,n,sizeof(double),compare)`
- C++: read using iostream; `sort(v.begin(),v.end());`

#elements	C++	C	C/C++ ratio
500,000	2.5	5.1	2.04
5,000,000	27.4	126.6	4.62

- How?

- clean algorithm
- inlining

(Details: May'99 issue of C/C++ Journal; <http://www.research.att.com/~bs/papers.html>)

Matrix optimization example

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

Function Objects

- Function objects
 - Essential for flexibility
 - Efficient
 - in practice, more so than inline functions
 - important: **sort()** vs. **qsort()**
 - Some find them tedious to write
 - Standard function objects
 - e.g., **less**, **plus**, **mem_fun**
 - Can be automatically written/generated
 - **Vector v2 = m*v+k;** // matrix and vector libraries
 - **find_if(b,e, 0<x && x<=max);** // lambda libraries

Object-oriented Programming

- Hide details of many variants of a concepts behind a common interface

```
void draw_all(vector<Shape*>& vs)  
{  
    typedef vector<Shape*>::iterator VI;  
    for (VI p = vs.begin(); p!=vs.end(), ++p) p->draw();  
}
```

- Provide implementations of these variants as derived classes

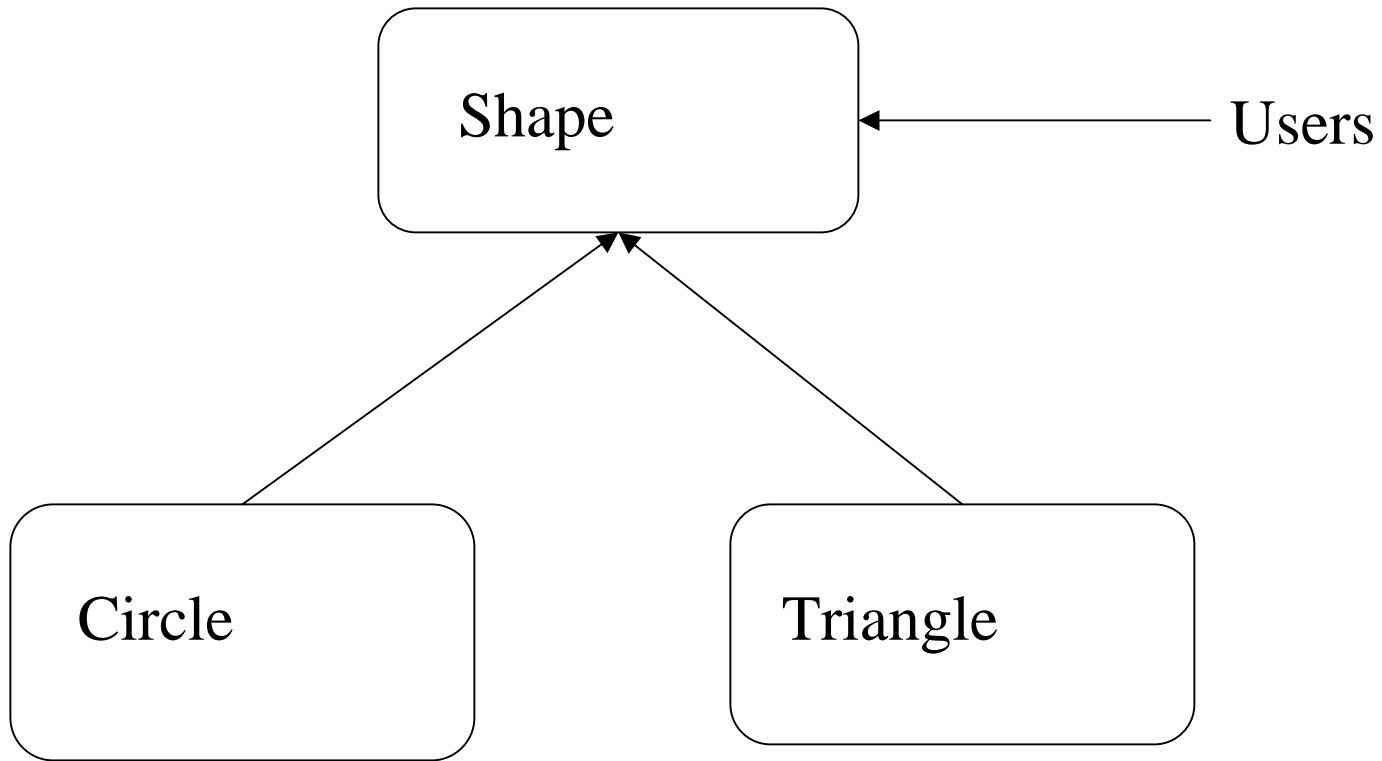
Class Hierarchies

- 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; /* ... */ void rotate(double) { } };
class Triangle : public Shape { Point a,b,c; /* ... */ void rotate(double); };
```

Class Hierarchies



Class Hierarchies

- Fundamental advantage: you can manipulate derived classes through the interface provided by a base:

```
void f(Shape* p)
{
    p->rotate(90);
    p->draw();
}
```

- You can add new **Shapes** to a program without changing or recompiling code such as **f()**

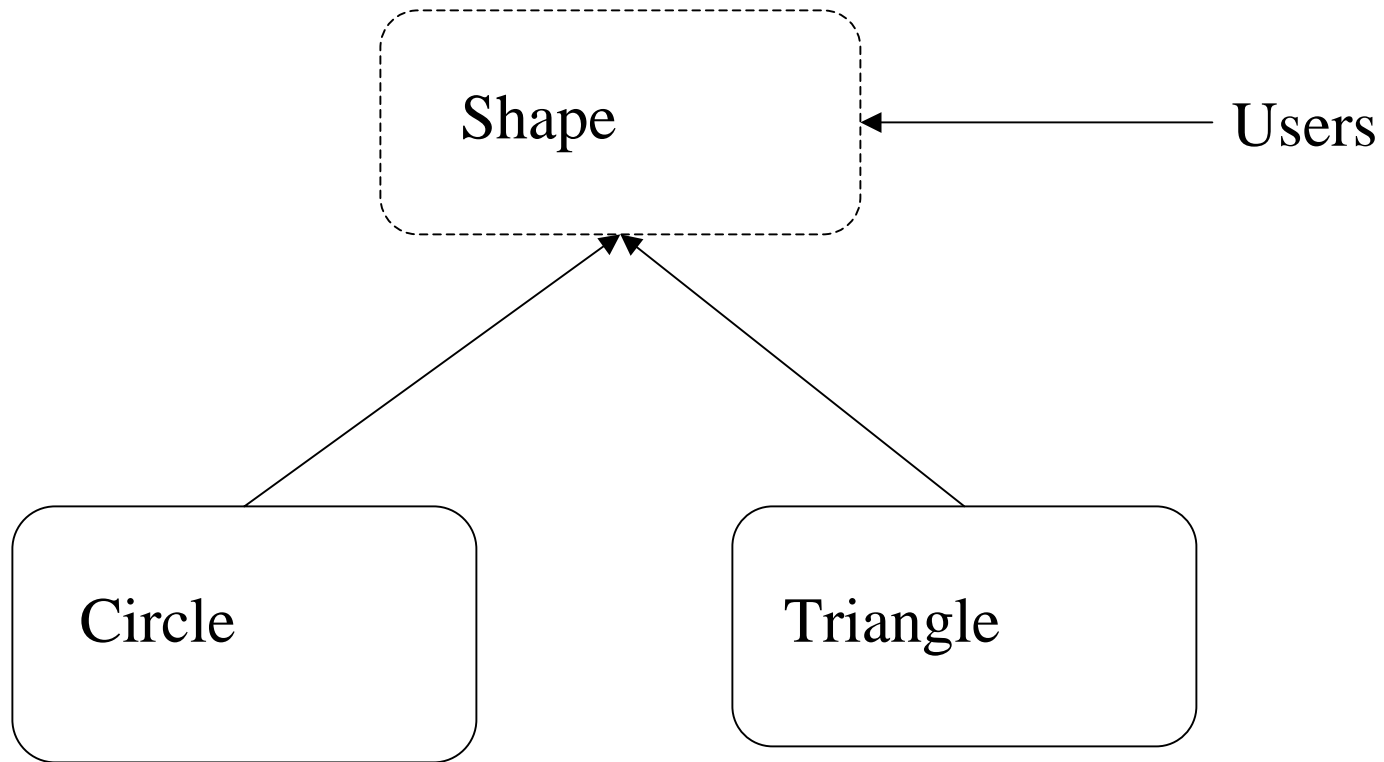
Class Hierarchies

- 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 center; double radius; Color c; /* ... */ };
class Triangle : public Shape { Point a, b, c; Color c; / * ... */ };
```


Class Hierarchies



Class Hierarchies

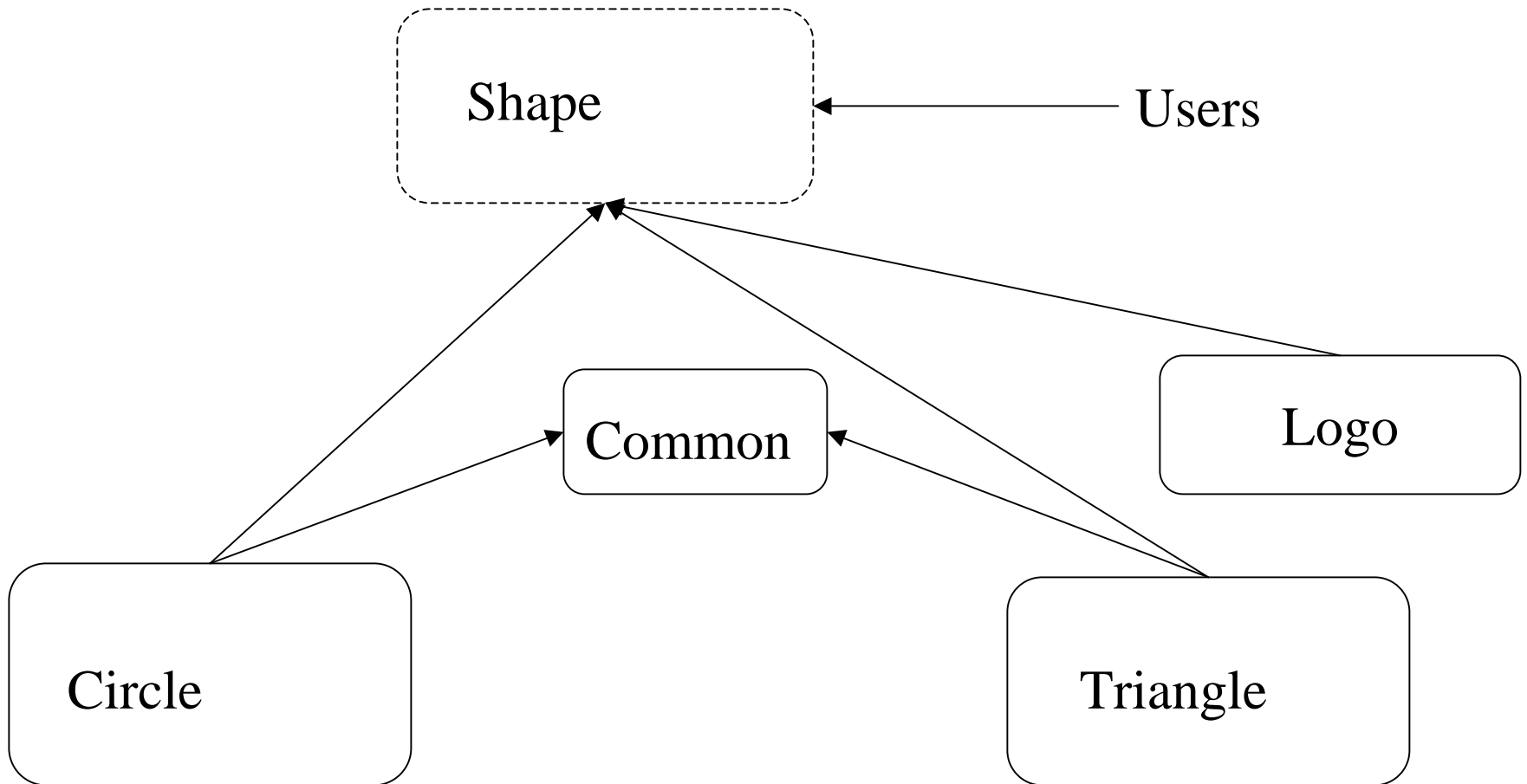
- One way to handle common state:

```
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{ /* ... */ };
class Triangle : public Shape, protected Common { /* ... */ };
class Logo: public Shape { /* ... */ };    // Common not needed
```

Class Hierarchies



Multiparadigm Programming

- The most effective programs often involve combinations of techniques from different “paradigms”
- The real aims of good design
 - Represent ideas directly
 - Represent independent ideas independently in code

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));  
}
```

Implications for Larger Systems

- First build or buy extensive libraries
 - Without suitable libraries everything is difficult
 - With suitable libraries most things are easy
 - Where possible, build on the C++ standard library
- Focus design/implementation on
 - Abstract classes as interfaces
 - Templates for type safety and efficiency
 - Function objects for flexible parameterization
- Avoid large single-rooted hierarchies
 - More generally, avoid unnecessary dependencies/coupling

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 (not members) 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)
 - New appendices: Standard-library Exception safety, Locales
- 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
 - Why C++ isn't just an Object-oriented Programming Language
- Higley and Powell: Expression templates ... (The C++ Report, May 2000)

- Links: <http://www.research.att.com/~bs>

- FAQs libraries, the standard, free compilers, garbage collectors, papers, chapters, C++ sites, interviews
- Open source C++ libraries: Boost.org, ACE, ...