Speaking C++ as a Native

(Multi-paradigm Programming in Standard C++)

Bjarne Stroustrup

Texas A&M University

(and AT&T Labs – Research)

http://www.research.att.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. I'll also make a few remarks about the likely directions for the C++0x ISO standard effort.

- 70 minutes, plus 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 examples
  - Function objects
- Object-Oriented Programming: class hierarchies and interfaces
  - Ye olde shape example
- Multi-paradigm Programming
  - Algorithms on shapes example
  - Universal base class?

Standard C++

- ISO/IEC 14882 – Standard for the C++ Programming Language
  - Core language
  - Standard library
- Implementations
  - Borland, HP/Compaq, IBM, Intel, 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
  - http://www.research.att.com/~bs/applications.html

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 techniques use a combination of styles/paradigms

Elegant, direct expression of ideas

- Declarative information is key:

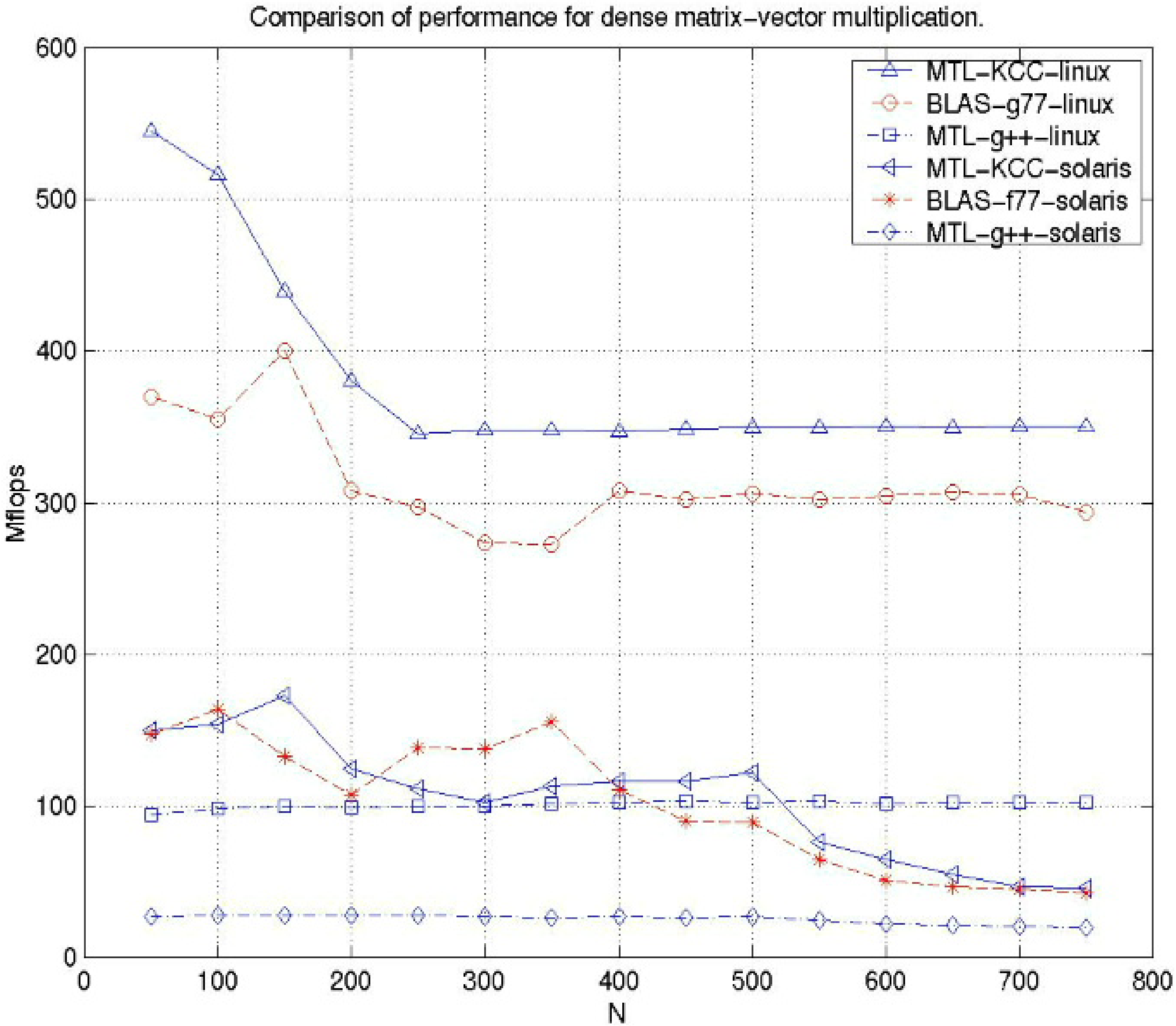
```
Matrix<double,100,50,Sparse> ms;
Matrix<Quad,100,50,Dense,Triangular<upper> > mt;
Matrix<double,100,50> m;     // defaults to rectangular and dense
```

- 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 greatest weakness: poor use

- C style
  - Arrays, **void***, 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

```
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 = 7;                // ok: 7 is in [0,10)
    int i = v1-v2;
    arg = v1;              // may throw exception
    v2 = arg;              // may throw exception
    v2 = 3;                // will throw exception: 3 is not in [7,100)

}
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

```
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); if (p==0) throw File_error(pp,r); }
        File_handle(const string& s, const char* r)
                { p = fopen(s.c_str(),r); if (p==0) throw File_error(pp,r); }
        ~File_handle() { fclose(p); }          // destructor
        // copy operations
        // access functions
};

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

Generic Programming

- First/original aim: Standard Containers
  - Type safe
    - without the need for run-time checking (not even implicit type checking)
  - Efficient
    - Without excuses ("write a **vector** to beat C arrays")
  - 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

Generic Programming

```
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());
}
```

Generic Programming

```
template<class T, class Cmp>       // parameterize with comparison
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
```

Generic Programming

```
struct Record {
    string name;
    char addr[24];          // old style to match database layout
    // …
};

vector<Record> vr;
// …
sort(vr.begin(), vr.end(), Cmp_by_name());
sort(vr.begin(), vr.end(), Cmp_by_addr());
```

Algorithms: comparisons

```
struct  Cmp_by_name {
    bool operator()(const Rec& a, const Rec& b) const
        { return a.name < b.name; }
};

struct  Cmp_by_addr {
    bool operator()(const Rec& a, const Rec& b) const
        { return 0 < strncmp(a.addr, b.addr, 24); }
};
```

Generic Programming: function objects

- A very simple function object

```
struct No_case {
        static bool operator()(char a, char b) const { /* … */ }
};
```

- A very general idea

```
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) const { /* do something with ss to s */ }
        operator S() const { return s; }        // reveal state
};
```

- A very efficient technique
  - inlining very easy (and effective with current compilers)
- The main method of policy parameterization in the C++ standard library
- Key to emulating functional programming techniques

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);
```

Notation and function objects

- You don't have to write function objects
  - libraries can write them for you (generative programming)

```
struct Lambda { };                 // placeholder type (to represent free variable)

template<class T> struct Le {  // represent the need to compare using <=
        T val;
        Le(T t) : val(t) { }
        bool operator()(T t) const { return val<=t; }
};

template<class T> Le<T> operator<=(T t, Lambda) { return Le<T>(t); }

Lambda x;
find_if(b,e,7<=x);    // generates find_if(b,e,Le<int>(7)))
                      // roughly: X x = Le<int>(7); for(I p = b, p!=e; x(*p++));

// Radical simplification - apologies to writers of lambda and expression template 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
- You can add new **Shape**s to a program without affecting user code, such as **draw_all()**

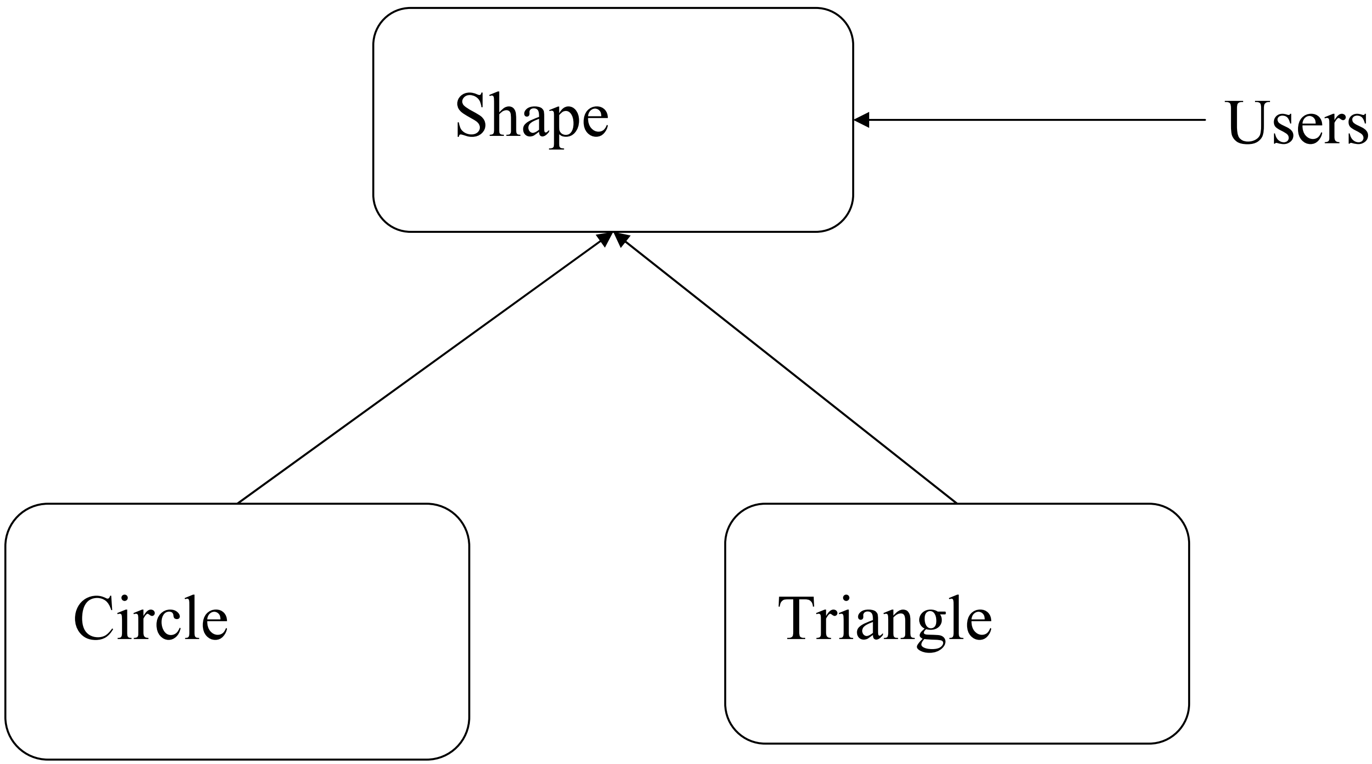
Class Hierarchies

- One way (often flawed):

```
class Shape {      // define interface and common state
    Color col;
    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



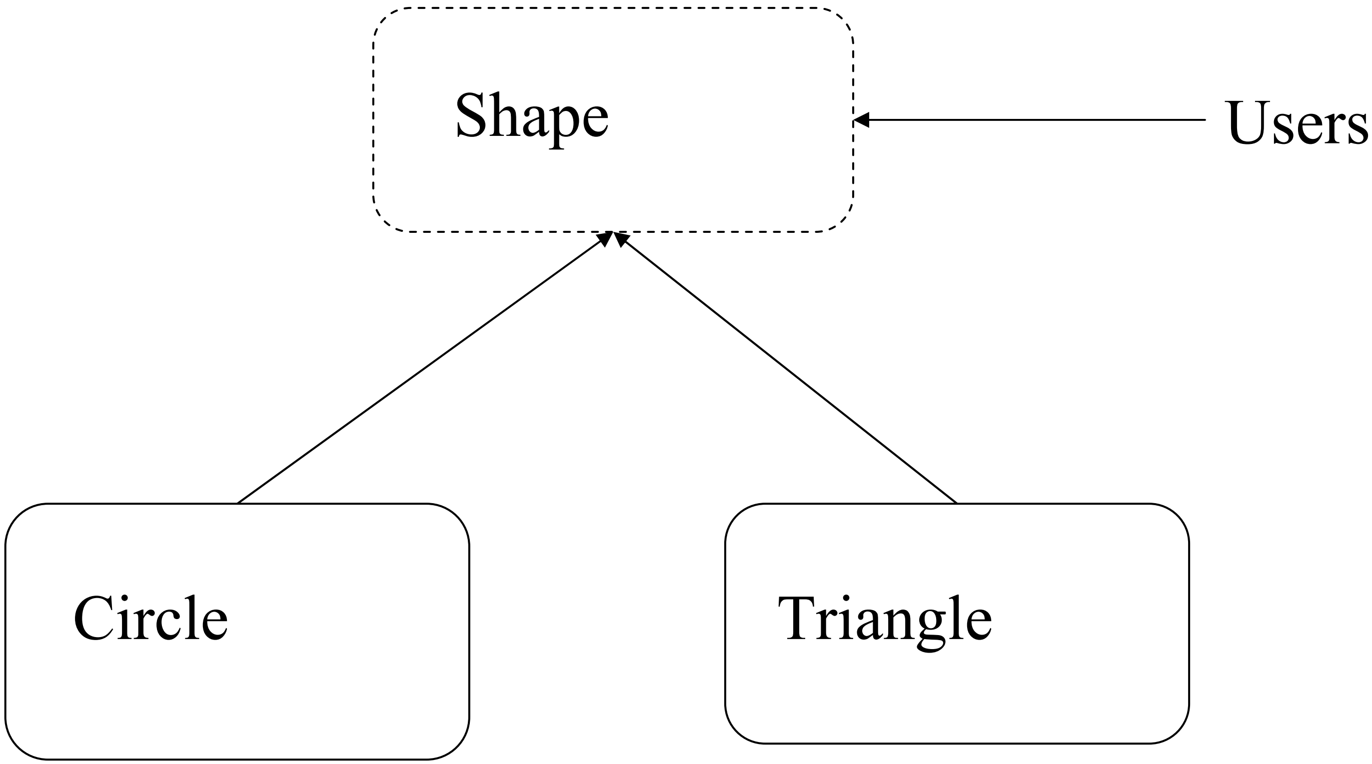
Class Hierarchies

- 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 radius; Color col; /* … */ };
class Triangle : public Shape {  Point a, b, c; Color col; / * … */ };
```

Class Hierarchies



Class Hierarchies

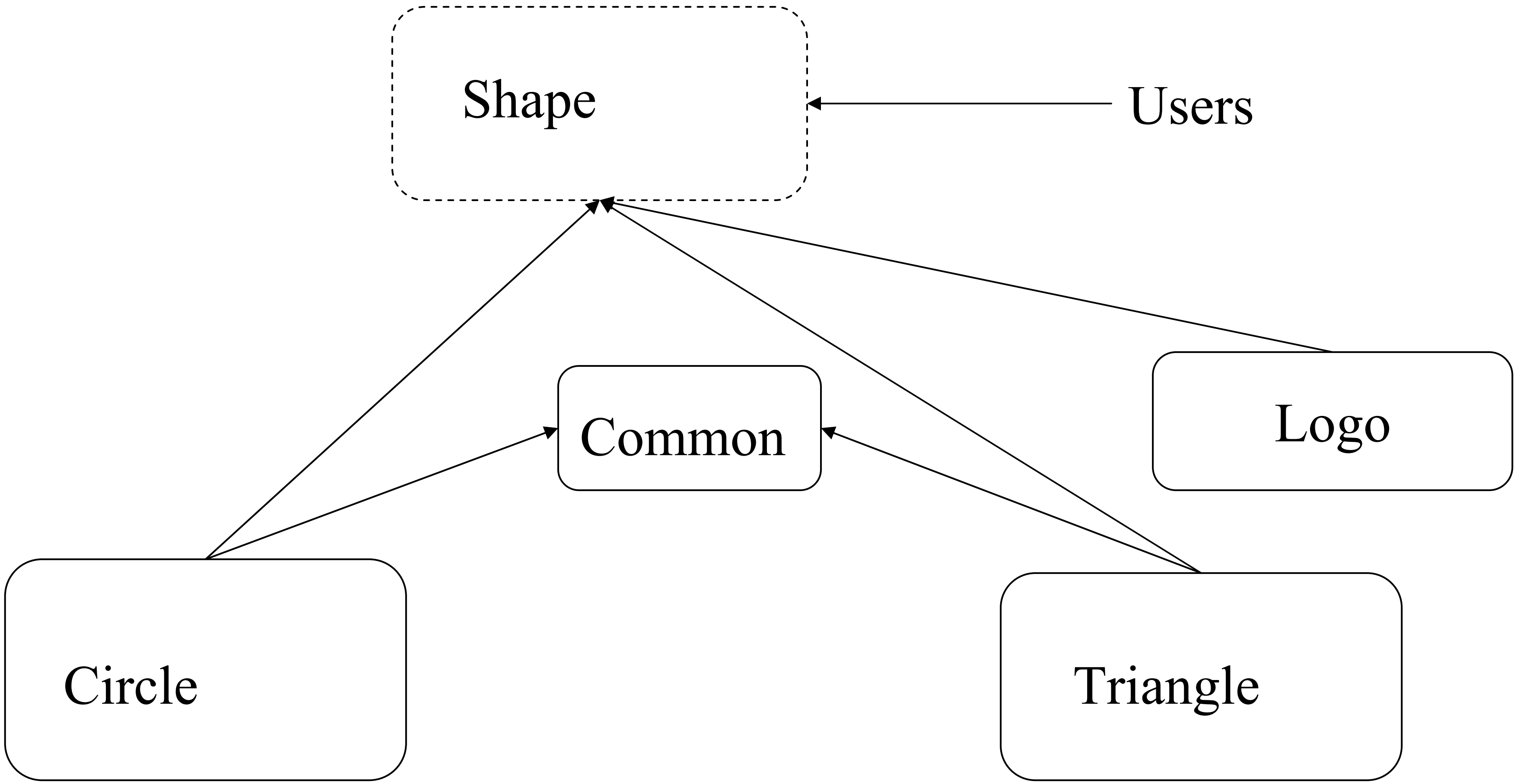
- One way to handle common state:

```
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));
}
```

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, talks on C++0x directions, interviews
  - Open source C++ libraries: Boost.org, ACE, …

C++ ISO Standardization

- Membership
  - About 22 nations (8 to 12 represented at each meeting)
    - ANSI hosts the technical meetings
    - Other nations have further technical meetings
  - About 120 active members (50+ at each meeting)
    - About 200 members in all
    - Down ~50% from its height (1996), up again last two years
- Process
  - formal, slow, bureaucratic, and democratic
  - “the worst way, except for all the rest”

Standardization status

- Standard completed in 1997, ratified 1998
- Performance TR 2003
  - Performance techniques for users and implementers
  - Embedded systems support
  - Stream I/O optimization
  - …
- Library TR 2004
  - Random Numbers
  - Mathematical Special Functions
  - Type Traits
  - Regular Expression
  - Smart Pointers
  - More and better Binders
  - Hash Tables
  - …
- Work on next standard (C++0x) started 2002

Standardization – why bother?

- Directly affects millions
  - Huge potential for improvement
    - So much code is appallingly poor
- Defense against vendor lock-in
  - Only a partial defense, of course
- There are still many new techniques to get into use
  - They require language or standard library support to affect mainstream use

Overall Goals

- Make C++ a better language for systems programming and library building
  - Rather than providing specialized facilities for a particular sub-community (e.g. numeric computation or Windows application development)
- Make C++ easier to teach and learn
  - Through increased uniformity, stronger guarantees, and facilities supportive of novices (there will always be more novices than experts)

General Directions

- Minimize incompatibilities with C++98
  - And if possible minimize C/C++ incompatibility
- Many ideas cut across the language/library barrier
  - Look for minimal language support allowing major library improvement
- Prefer library extension to language extension
  - Make rules more general and uniform
  - Support communities
- Language extensions
  - Maintain or increase type safety
  - Zero-overhead principle
  - Increase expressiveness through general mechanisms
- Library extensions
  - Increase facilities of system-independent platform
  - Support distributed systems programming

Language Directions

- Minimize extensions
  - Be careful, deliberate, conservative, skeptic
- Make rules more general and uniform
  - Improve support for generic programming
  - Improve general guarantees (increase uniformity)
- Look to support whole communities, e.g.
  - improve support low-level embedded programming
  - improve binding to “dynamic” systems?
    - Can we support modern/fashionable GUI/component/system interfaces without major language changes or proprietary extensions?

Library TR

- Hash Tables
- Regular Expressions
- General Purpose Smart Pointers
- Extensible Random Number Facility
- Mathematical Special Functions

- Polymorphic Function Object Wrapper
- Tuple Types
- Type Traits
- Enhanced Member Pointer Adaptor
- Reference Wrapper
- Uniform Method for Computing Function Object Return Types
- Enhanced Binder

Performance TR

- The aim of this report is:
  - to give the reader a model of time and space overheads implied by use of various C++ language and library features,
  - to debunk widespread myths about performance problems,
  - to present techniques for use of C++ in applications where performance matters, and
  - to present techniques for implementing C++ language and standard library facilities to yield efficient code.
- Contents
  - Language features: overheads and strategies
  - Creating efficient libraries
  - Using C++ in embedded systems
  - Hardware addressing interface

What’s out there? Boost.org

- Filesystem Library – Portable paths, iteration over directories, etc
- MPL added – Template metaprogramming framework
- Spirit Library – LL parser framework
- Smart Pointers Library –
- Date-Time Library –
- Function Library – function objects
- Signals – signals & slots callbacks
- Graph library –
- Test Library –
- Regex Library – regular expressions
- Format Library added – Type-safe 'printf-like' format operations
- Multi-array Library added – Multidimensional containers and adaptors
- Python Library – reflects C++ classes and functions into Python
- uBLAS Library added – Basic linear algebra for dense, packed and sparse matrices
- Lambda Library – **for_each(a.begin(), a.end(), std::cout << _1 << ' ');**
- Random Number Library
- Threads Library
- …

What's out there? (Lots!)

- ACE – portable distributed systems programming platform
- Blitz++ – the original template-expression linear-algebra library
- SI – statically checked international units
- Loki – mixed bag of very clever utility stuff
- Endless GUIs and GUI toolkits
  - GTK+/gtkmm, Qt, FOX Toolkit, eclipse, FLTK, wxWindows, …
- … much, much more …

see the C++ libraries FAQ (link on my C++ page)

Core language suggestions (Lots!)

- **decltype/auto** – type deduction from expressions
- Template alias
- **#nomacro**
- Extern template
- Dynamic libraries
- Allow local classes as template parameters
- Move semantics
- **nullptr** - Null pointer constant
- Static assertions
- Solve the forwarding problem
- Variable-length template parameter lists
- Simple compile-time reflection
- GUI
- Defaulting and inhibiting common operations
- Class namespaces
- **long long**
- **>>** (without a space) to terminate two template specializations
- …