What C++ Is and Why

Bjarne Stroustrup

AT&T Labs – Research

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

Overview

- The origins of C++
  - Problems to be addressed
  - Aims and design rules
- Key C++ language features and programming techniques
  - Data abstraction and concrete classes
  - Generic programming and templates
  - Object-oriented programming and class hierarchies
- The current state of C++
  - Use and users
  - Standardization

The problem

- Simula and BCPL
  - program organization
  - run-time efficiency
  - availability/portability
  - inter-operability

Original Idea

Combine

C's strengths as a systems programming language

with

Simula's facilities for program organization

Why C?

- The best systems programming language available
  - flexible, efficient, portable
  - available, known
  - 2nd order flaws not critical

  but: improve static type checking

Why Classes?

- program organization
  - mapping of concepts to classes
  - a class is a type
  - static checking
- Programming as modeling, as understanding
  - Class hierarchies (Object-Oriented Programming)
  - Lack of parameterized classes always seen as problem
    - Roots of generic programming in first paper (but very flawed)

“C with Classes” to C++, why?

- C with Classes was a “medium success”
  - allowed medium improvements (only)
  - user community couldn't support infrastructure

  => more or less (or hype)

C++

- Emphasis on
  - static structure
  - type safety
  - design
  - run-time and space efficiency (where needed)
  - co-existence with other languages
  - applicability in many environments
- Pragmatism/experience valued over theory
  (this does not imply disrespect to theory)

C++ design rules of thumb ("general principles")

- C++ is a language, not a complete system
- What you don't use you don't pay for
  - (zero-overhead rule)
- C++ must be useful NOW
  - (current machines, programmers, and problems)
- Don't get involved in a sterile quest for perfection
- Use traditional (dumb) linkers
- No gratuitous incompatibilities with C
  - (source and linkage)
- No implicit violations of the type system
- Uniform rules for user-defined and built-in types

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

- C++ is a multi-paradigm programming language
  - Some of the most effective programming styles involves more than one programming style (“paradigm”)
  - Recent C++ evolution has focused on the use of templates and exceptions in support for multiple styles

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
  - powerful tools and environments
  - In-depth tutorials
  - reference material

C++ Classes

- Primary tool for representing concepts in code
  - 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, int i)
try
{
    Range v1(0,3,10);
    Range v2(7,9,100);
    v1 = v2;        // ok: 9 is in [0,10)
    arg = i;        // 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) const { 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; }
};
```

Classes as value types

- But I was talking about “ranges” not just about “ranges of integers”!

```
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”);
```

Classes as value types

```
template<class T> class Range {                 // simple value type
    T value, low, high;                         // invariant: low <= value < high
    void check(const T& v) const { 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; }
};
```

Classes as value types

- But I was **not** talking about ranges of any type, just ranges of types with < and <=

```
template<class T> struct Comparable {
    static void constraint(T a, T b) { a<b; a<=b; }    // express constraint
    Comparable() { void(*p)(T,T) = constraint; }     // trigger constraint
};

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

Range<int> ri(a,b,c);       // ok
Range<complex<double> > rc(d,e,f);        // error: Range::constraint: no <, <=
```

Containers: vector

```
template<class T>
class vector {      // simplified standard library vector
    T* v;           // pointer to elements
    T* space;       // end of elements+1, start of free space
    T* end;         // end of free space+1
public:
    vector(size_t n, const T& val = T());     // create/construct
    vector(const vector& a);                  // copy
    vector& operator=(const vector& a);       // copy
    ~vector();                                // cleanup/destroy

    size_t size() const { return space-v; }
    T& operator[](size_t n) { return v[n]; }
    T& at(size_t n) { if (size()<=n) throw range_error(); return v[n]; }
    void push_back(const T& v);   // add copy of v to end of vector
    // …
};
```

Containers: vector

```
vector<Point> cities;

void read_cities(istream& is, Point terminator)
{
    Point p;
    while (is>>p && p!=terminator) {
        // check p
        cities.push_back(p);        // add copy of p to end of vector
    }
}
```

Standard containers

- Containers
  - Vector, list, deque, map, multi_map, set, multi_set
- Adapters
  - Stack, queue, priority_queue
- “Almost containers”
  - String, bitset, valarray, vector<bool>, arrays

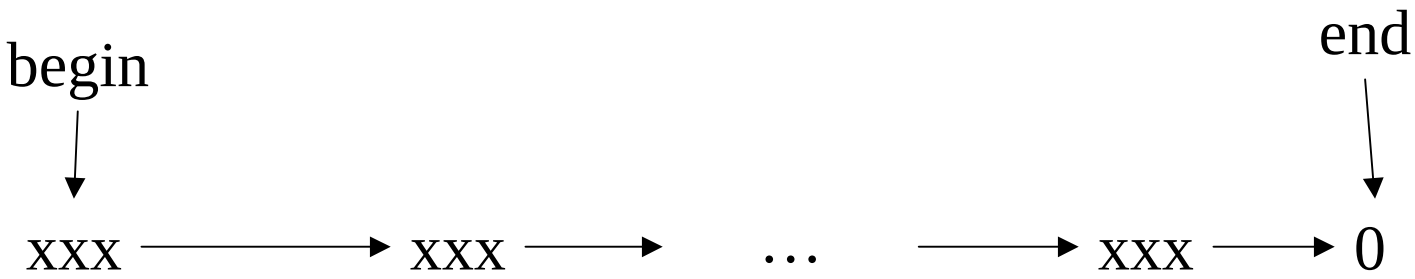
Containers: string

```
string get_address()
{
    cout << “please enter user id\n”;
    string id;
    cin >> id;
    cout << “Please enter stite name\n”;
    string addr;
    cin >> addr;
    // check that id and addr are plausible
    return id + “@” + addr;
}
```

Algorithms: Genericity

- So, once you have containers what do you do with them?
  - 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: Iterators and sequences



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: find()

```
template<class In, class T>
In find(In first, In last, T val)
{
    while (first!=last && *first!=val) ++first;
    return first;
}

void f(vector<int>& v, list<string>& lst)
{
    vector<int>::iterator p = find(v.begin(), v.end(), 42);
    if (p != v.end()) { /* we found 42 */ }

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

Algorithms: find_if()

```
template<class In, class Pred>
In find_if(In first, In last, Pred p)
{
    while (first!=last && !p(*first)) ++first;
    return first;
}

bool less_than_7(int x) { return x<7; }

void f(vector<int>& v)
{
    vector<int>::iterator p = find_if(v.begin(), v.end(), less_than_7);
    if (p != v.end()) { /* we found an element < 7 */ }
}
```

Algorithms: find_if()

```
template<class T> class less_than {
    T v;
public:
    less_than(const T& vv) :v(vv) { }
    bool operator()(const T& a) const { return a<v; }
};

void f(vector<int>& v, int x)
{
    vector<int>::iterator p = find_if(v.begin(), v.end(), less_than<int>(x));
    if (p != v.end()) { /* we found an element < x */ }
}
```

Classes as function types

Function objects are more general than functions

```
template<class Res, class Arg, class State>
class Action {      // function object
    State s;
public:
    Action(const State& ss) : s(ss) { }
    Res operator()(const Arg& a) { /* … */ }
    State examine() const { return s; }
    // …
};

Action<double, int, string> f("hello");      // create a function object
double  r = f(2);                            // call it
```

Algorithms: sort()

```
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

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

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

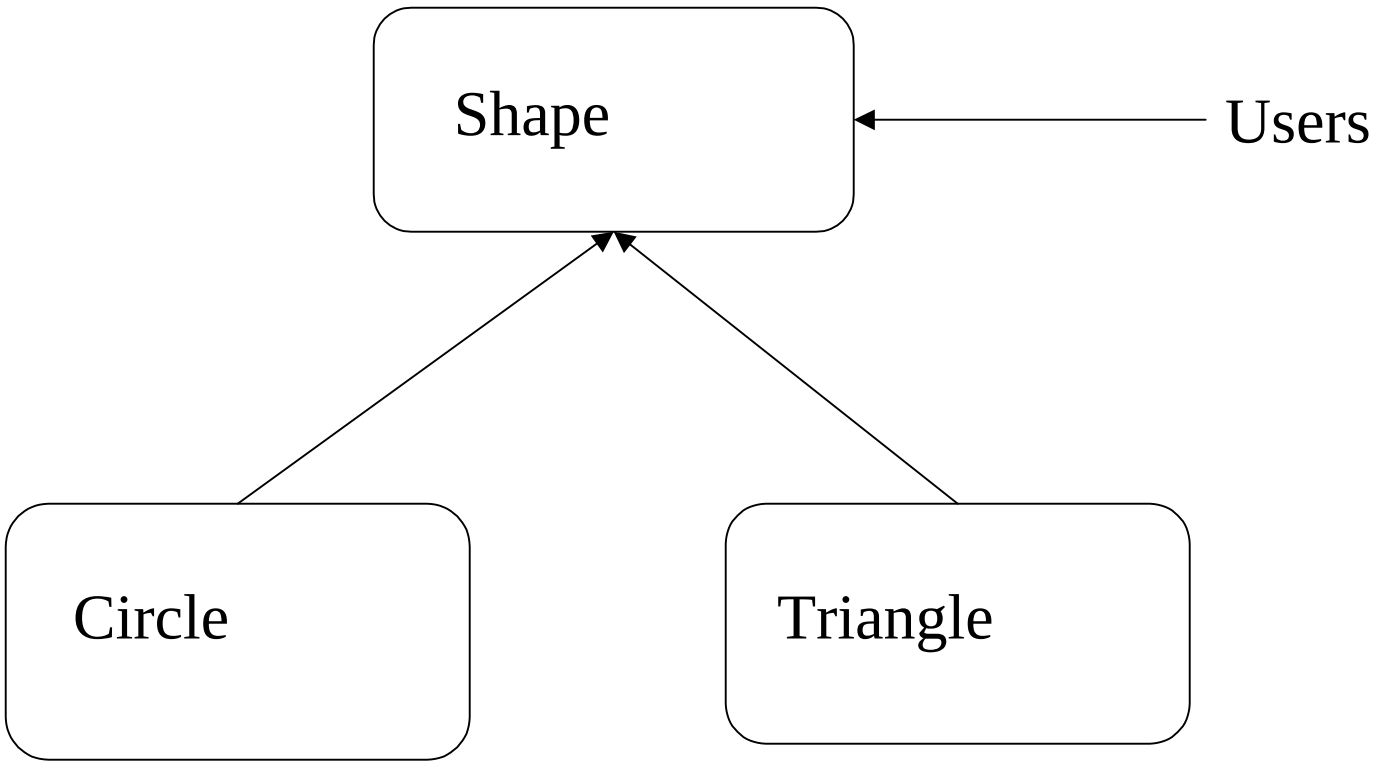
Class Hierarchies

- One way (usually flawed):

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

class Circle : public Shape { /* … */ void draw(); void rotate(double); };
class Triangle : public Shape { / * … */  void draw(); void rotate(double); };
```

Class Hierarchies



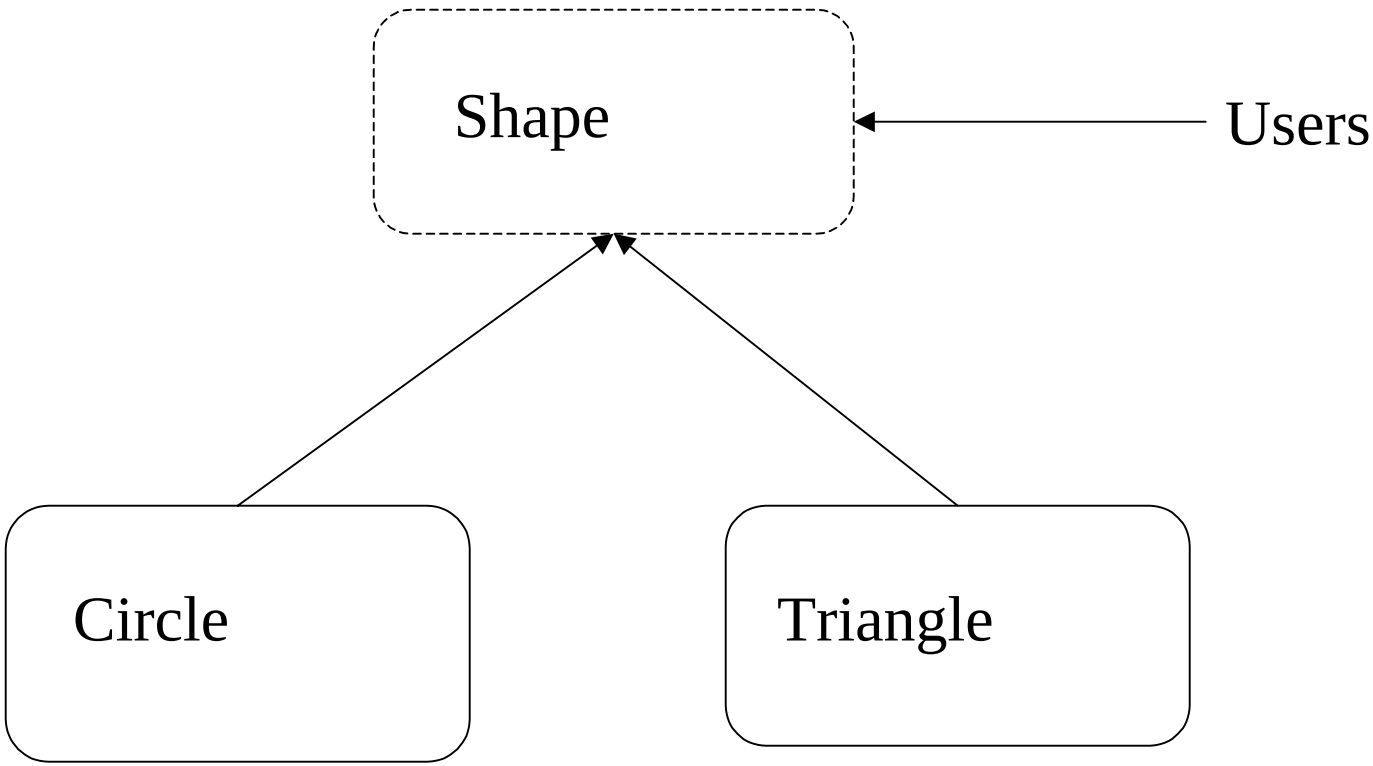
Class Hierarchies

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

Class Hierarchies



Class Hierarchies

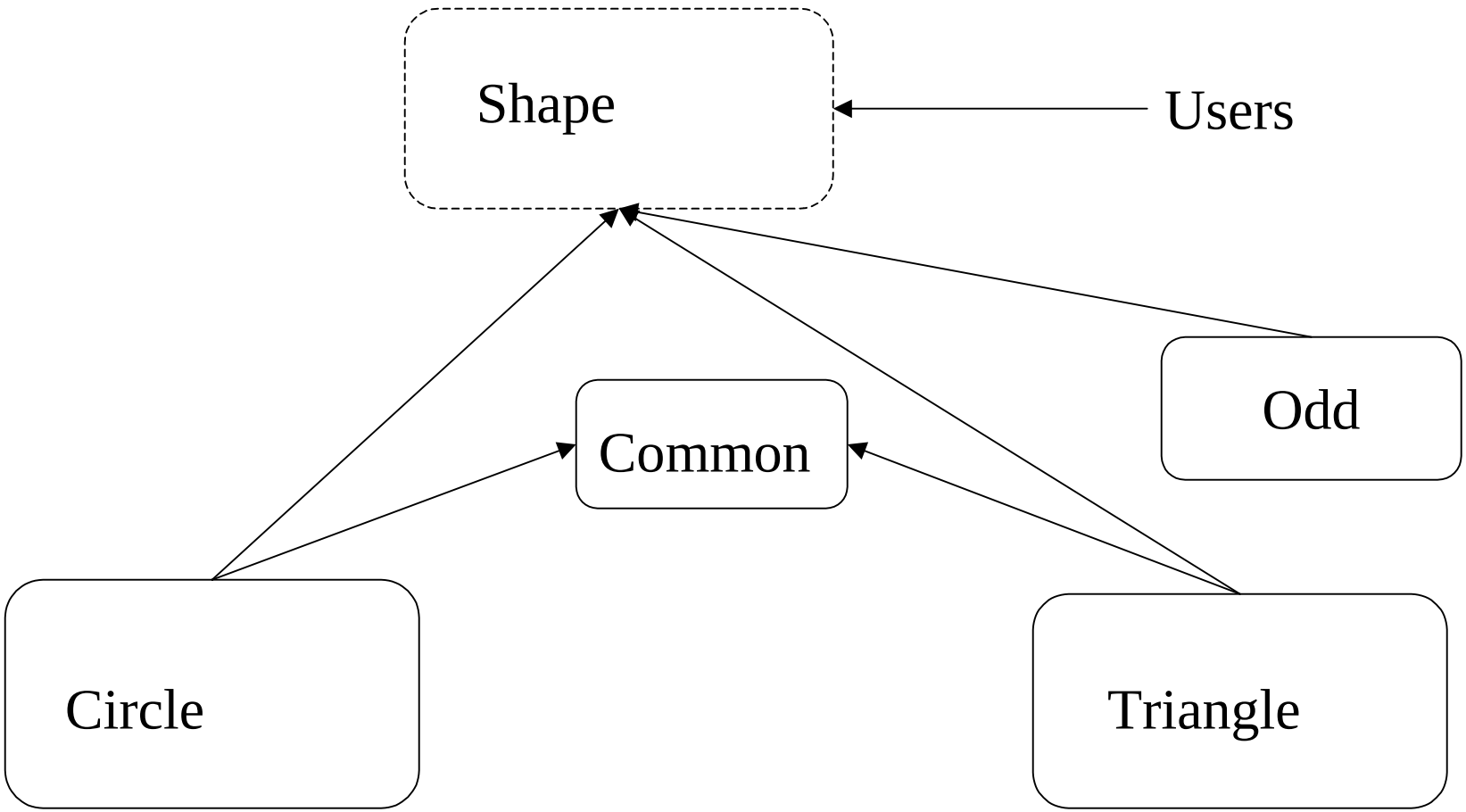
- One way to handle common state:

```
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 Odd : 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<C,Shape*>();                               // of Shape*s
    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<For,Shape*>();                            // of Shape* elements
    for_each(first, last, mem_fun(&Shape::draw));
}
```

Standard C++

- ISO/IEC 14882 – Standard for the C++ Programming Language
  - Core language
  - Standard library
- Implementations
  - Borland, IBM, EDG, DEC, 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, embedded systems, etc.

Standard C++

- The result of a consensus process
  - Canada, France, Germany, Japan, UK, US, …
  - Apple, AT&T, Borland, Ericcsson, IBM, HP, Intel, Microsoft, Siemens, Sun, …
  - Hundreds of individuals
  - Final votes: ANSI: 43-0, ISO: 22-0
  - Maintenance process: Corrigenda, November 2000
- Serves the needs of diverse communities
- The committee serves as neutral ground for technical discussions
- Offers some protection from rapacious vendors
  - Validation suites available (Plum Hall, Perennial, …)

C++ Applications

There are tens of thousands of successful C++ applications:

- Billing systems, browsers, circuit routing, camera control, chemical engineering process control, compilers, database systems, device drivers, electronic trading, engine control, graphics, geometric modeling, image processing, linear algebra, operating systems, operations systems, missile guidance, robotics, switching, simulation, symbolic algebra, telemetry, transaction systems, user interfaces, video games, word processing, …

On average, we can use Standard C++ to write cleaner and more efficient code faster than is commonly done now

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
- 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++
    - Sutter: Exceptional C++
  - 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, book chapters, C++ sites, interviews