Using Standard C++

Bjarne Stroustrup
AT&T Labs – Research
http://www.research.att.com/~bs

Overview

- Standard C++
  - My view of what it is
  - Ideals for its use
- Classes
  - Values
  - Function objects
- Containers
  - vector
  - string
- Algorithms
  - find, find_if
  - sort
- Class Hierarchies
  - Abstract classes
  - Algorithms on polymorphic containers

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++

- My aims for C++: A language
  - in which I can write programs that are simultaneously elegant and efficient
  - in which a wide range of applications can be built
  - that coexist well with other languages
  - that supports serious programmers well
  - that can be used on essentially every platform
  - that isn't proprietary

For specifics, see "The Design and Evolution of C++"

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 main 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, concurrency, persistence, …
- Smalltalk-style hierarchies
  - "brittle" base classes
  - Overuse of hierarchies

Here, I want to show alternatives:

My main focus is the containers and algorithms part of the standard library (the STL)

Applications

Not to sound negative or condescending:

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, …

but on average, we can write cleaner and more efficient code faster than is commonly done now

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

Caveat

- There is no substitute for
  - Intelligence
  - Experience
  - Taste
  - Hard work

What can a language do for you?

- Help express application concepts
  - Directly
  - Affordable

Classes

- A class should represent a concept

```
class X {
        // data
        // operations
};
```

- **A class is the type of objects**

```
X a;              // a is an X
X();              // make an X on the stack
new X;            // make an X on the free store (heap)
new(Area) X;      // make an X in Area
```

- **Classes are the building blocks of systems**

Kinds/Roles of Classes

- + Value types

  complex, point, range, matrix, record, date, pair, …
- + Containers

  vector, list, map, …
- + Actions
- + Interfaces
- + Nodes
- Handles
- Application frameworks

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

```
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; }
};
```

Classes as value types

```
void f(Range arg)
try
{
    Range v1(0,3,10);
    Range v2(7,9,100);
    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

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

    operator T() const { return 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");
```

Classes as function types

```
class Action {      // function object
    int s;
public:
    Action( int ss) : s(ss) { }
    int do_it(int a) { /* … */ }
    int examine() const { return s; }
    // …
};

Action f(3);                // create a function object
int x = f.do_it(7);         // call it
int y = f.examine();        // examine state (you can't do that for functions)
```

Classes as function types

```
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
string s = f.examine();                      // examine the state
```

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());
    vector(const vector& a);
    vector& operator=(const vector& a);

    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 element at 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

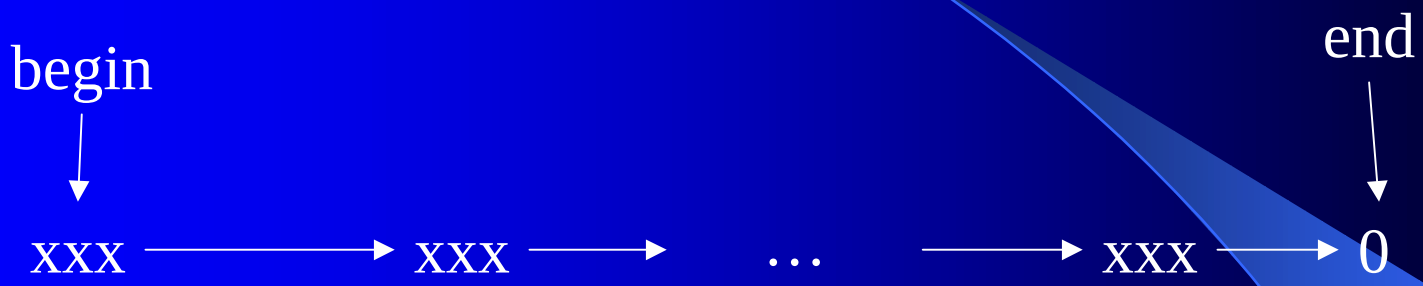
Containers: string

```
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)       // find val in sequence [first,last)
{
    while (first!=last && *first!=val) ++first;
    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 first, In last, Pred p)  // find an element x for which p(x) is true
{
    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 != 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 != end) { /* we found an integer element < x */ }
}
```

Algorithms: sort()

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

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

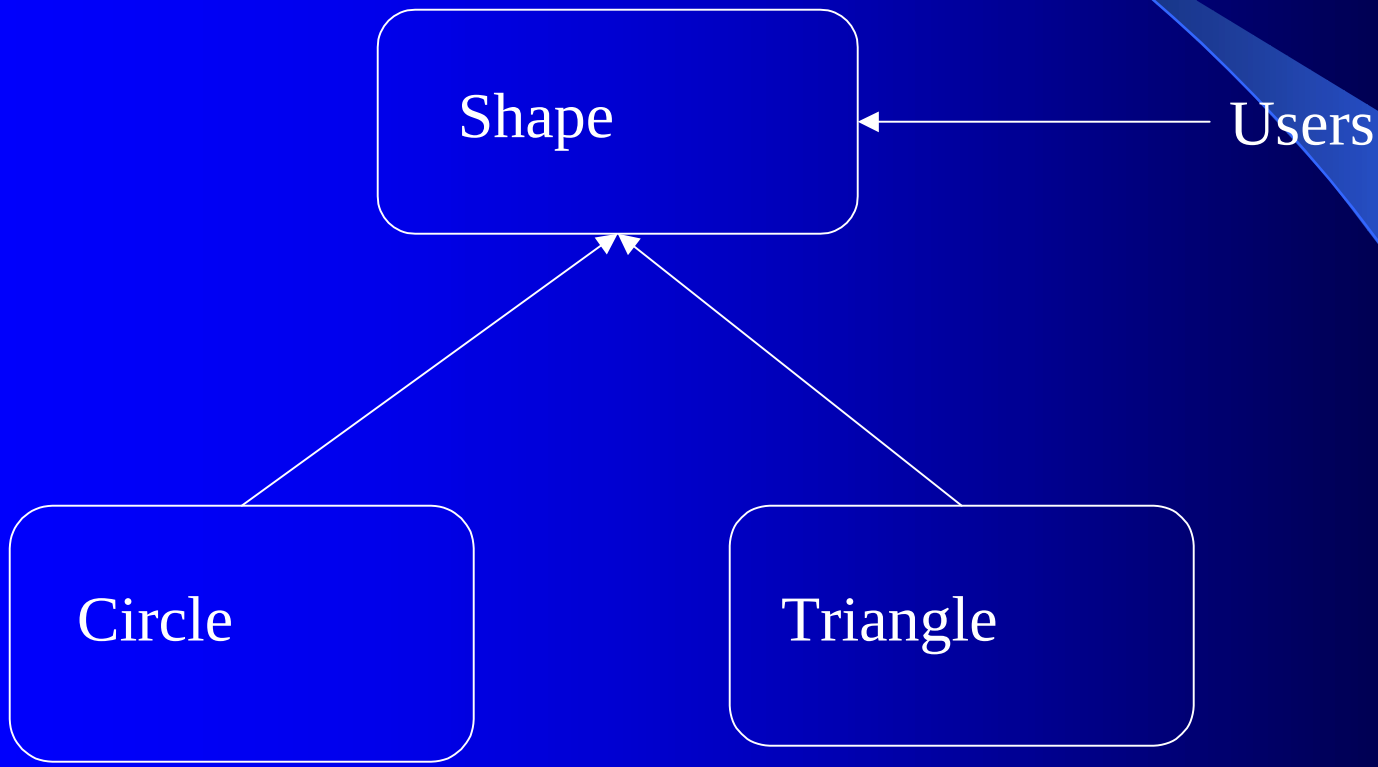
Class Hierarchies

- One way (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 { /* … */ };
class Triangle : public Shape { / * … */ };
```

Class Hierarchies



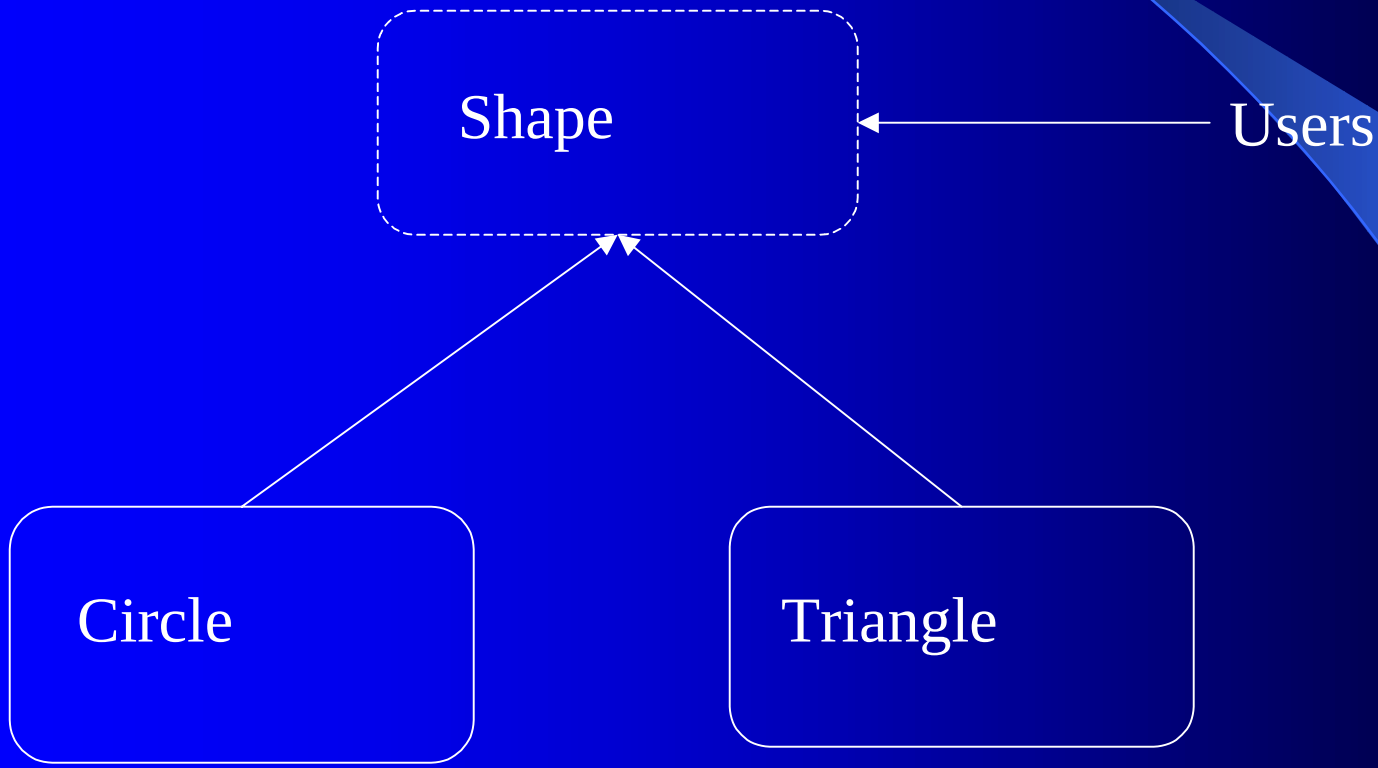
Class Hierarchies

- Another way (usually better):

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

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

Class Hierarchies



Class Hierarchies

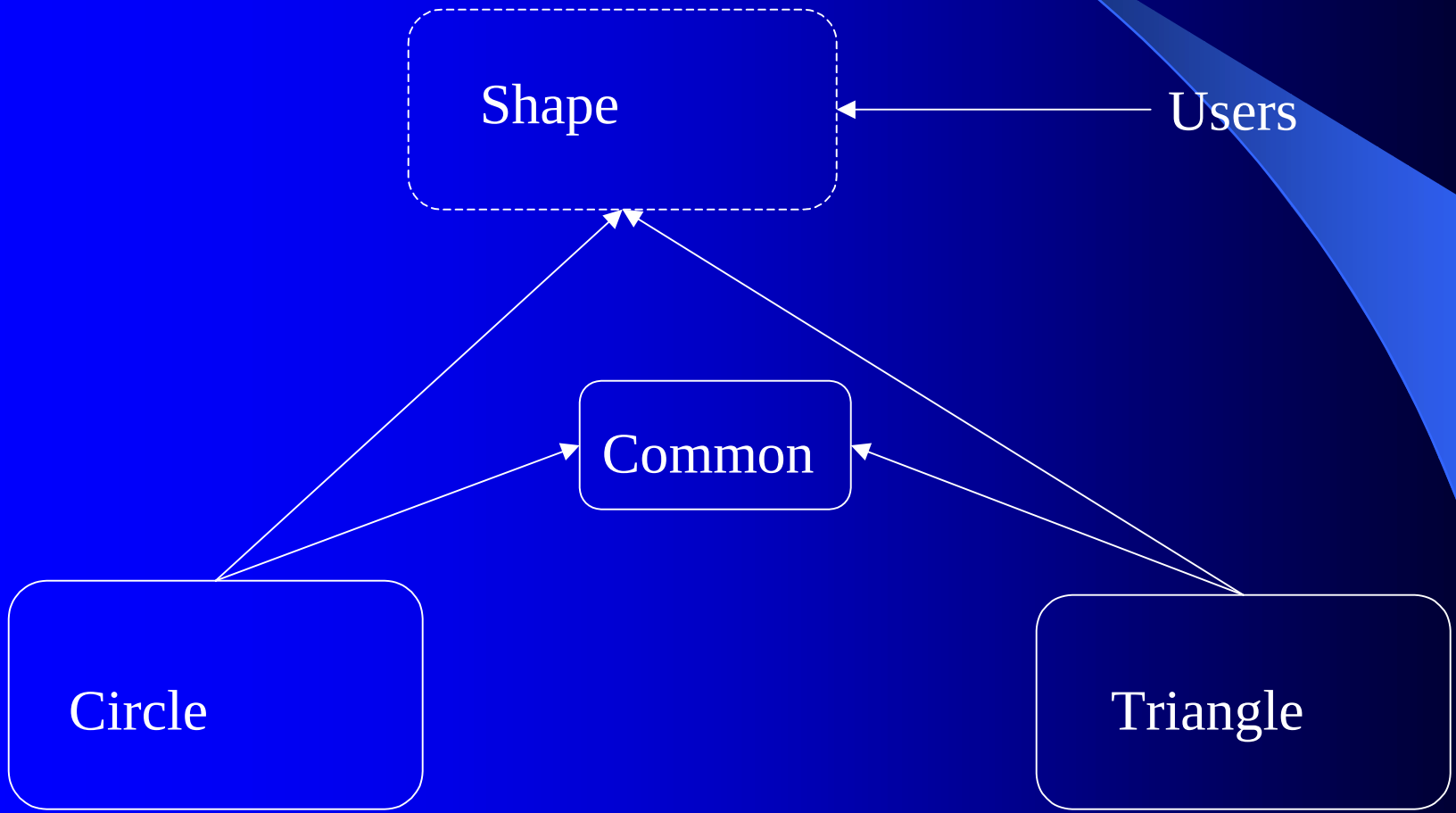
- One way to handle common state:

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

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

class Circle : public Shape, protected Common{ /* … */ };
class Triangle : public Shape, protected Common { / * … */ };
```

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
{
    for_each(c.begin(), c.end(), mem_fun(&Shape::draw));
}

template<class For> void draw_all(For first, For last)      // for all sequences
{
    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
- 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
  - 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