C++ Tutorial: Advanced Programming Techniques

Bjarne Stroustrup
Texas A&M University
http://www.research.att.com/~bs

General idea

- Without libraries (using only the core language) every task is difficult and tedious
  - maybe even unmanageable
- With suitable libraries every task is manageable
  - maybe even pleasant

- This tutorial focuses on the language features and programming we use to design, implement, and use good libraries

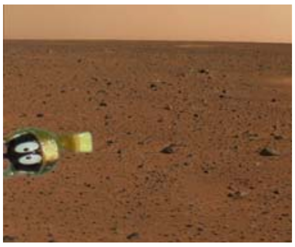
- My aim is improved understanding
  - Not specific detailed skills
- My assumption is that you are a programmer who wants to deliver quality systems
  - Not an academic

Overview

- Part 1
  - C++
  - Mapping to the machine
  - Error handling
- Part 2
  - Generic programming
  - Classes and class hierarchies
- Part 3
  - C++0x summary

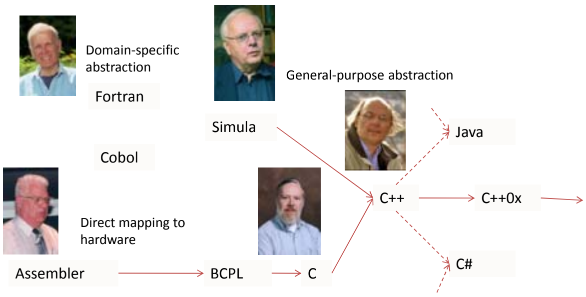
Programming languages

- A programming language exists to help people express ideas
- Programming language features exist to serve design and programming techniques
- The primary value of a programming language is in the applications written in it



- The quest for better languages has been long and must continue

Programming Languages



Ideals

- Work at the highest feasible level of abstraction
  - More general, correct, comprehensible, and maintainable code
- Represent
  - concepts directly in code
  - independent concepts independently in code
- Represent relationships among concepts directly
  - For example
    - Hierarchical relationships (object-oriented programming)
    - Parametric relationships (generic programming)
- Combine concepts
  - freely
  - but only when needed and it makes sense

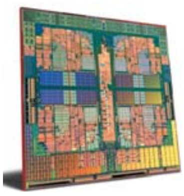
Part 1 Overview

- Mapping to the machine
- Error handling
  - Using exceptions
- We'll touch upon an amazingly large part of the most useful C++ features
  - Ask when needed

Ideals

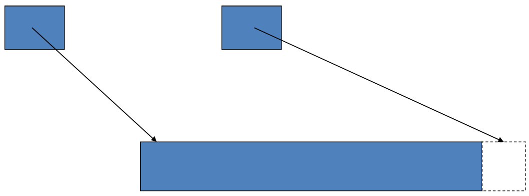
- Work at the highest feasible level of abstraction
  - More correct, comprehensible, and maintainable code
- Represent
  - concepts directly in code
  - independent concepts independently in code
- Represent relationships among concepts directly
  - For example
    - Hierarchical relationships (object-oriented programming)
    - Parametric relationships (generic programming)
- Combine concepts
  - freely
  - but only when needed and it makes sense

C++ maps directly onto hardware



- Mapping to the machine
  - Simple and direct
  - Built-in types
    - fit into registers
    - Matches machine instructions
- Abstraction
  - User-defined types are created by simple composition
  - Zero-overhead principle:
    - what you don't use you don't pay for
    - What you do use, you couldn't hand code any better

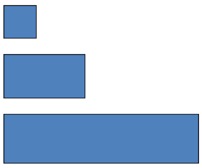
Memory model



Memory is sequences of objects addressed by pointers

Memory model (built-in type)

- char
- short
- int
- long
- (long long)
- float
- double
- long double
- T* (pointer)
- T& (implemented as pointer)



Memory model ("ordinary" class)

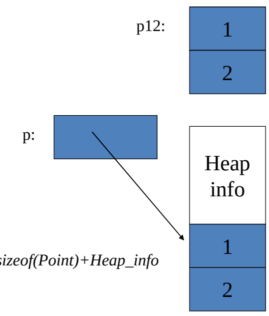
```
class Point {
    int x, y;
    // …
};
```

*// sizeof(Point)==2*sizeof(int)*

Point p12(1,2);

Point* p = new Point(1,2);

// memory used for "p":sizeof(Point)+sizeof(Point)+Heap_info*

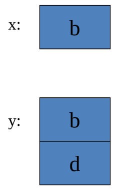


Memory model – class hierarchy

```
class B {
    int b;
};

class D : public B {
    int d:
};

B x;
D y;
```

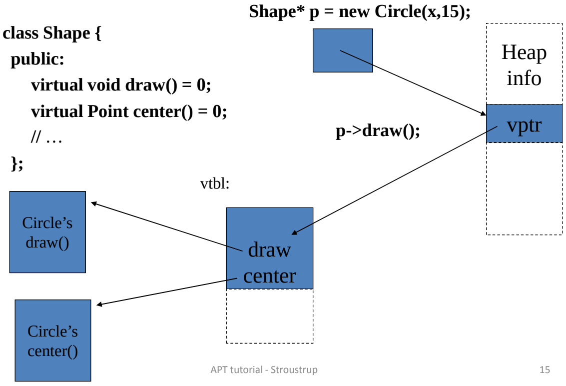


Memory model (polymorphic type)

```
class Shape {
 public:
    virtual void draw() = 0;
    virtual Point center() = 0;
    // …
};
```

Shape* p = new Circle(x,15);

p->draw();



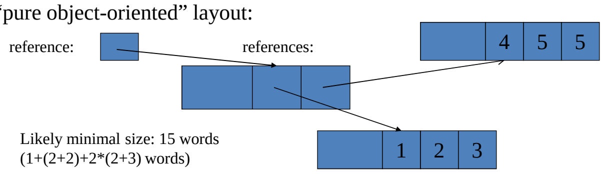
Not all memory models are that direct

- Consider a pair of coordinates
  - **class Coord { double x,y,z; /*** *operations* ***/ };**
  - **pair<Coord> xy= { {1,2,3}, {4,5,6} };**

C++ layout:

Likely size: 6 words
(2*3 words)

"pure object-oriented" layout:



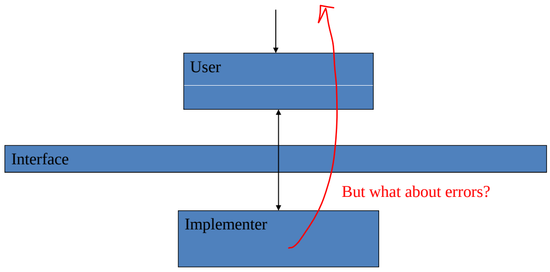
Likely minimal size: 15 words
(1+(2+2)+2*(2+3) words)

Abstraction

- Simple user-defined types (“concrete types”)
  - classes
    - Amazingly flexible
    - Zero overhead (time and space)
- Hierarchical organization (“abstract types”)
  - Class hierarchies, virtual functions
    - Object-oriented programming
    - Fixed minimal overhead
- Parameterized abstractions (“generic types and functions”)
  - Templates
    - Generic programming
    - Amazingly flexible
    - Zero overhead (time and space)

Interfaces

- *interface* is the central concept in programming



Traditional error handling

- Error state
```
double res = sqrt(x);           // may set errno (e.g. x==-1)
if (errno) { /* handle error */ }
```
- Error return codes
```
int res = area(lgt,w);     // can return any positive int
if (res<=0) { /* handle error */ }
int res2 = read_int();     // can return any int (bummer!)
```
- (error_code,value) pairs
```
pair<Error_no,int> r = area(lgt,w);
if (r.first) { /* handle error */ }
int res = r.second;  // good value
```
- Give up
```
int compute(arguments)
{
   if (bad arguments) exit(1);
   // …
}
```

Exception Handling

- The problem:
  provide a systematic way of handling run-time errors
  - C and C++ programmers use many traditional techniques
    - Error return values, error functions, error state, …
    - Chaos in programs composed out of separately-developed parts
  - Traditional techniques do not integrate well with C++
    - Errors in constructors
    - Errors in composite objects
  - Code using exceptions can be really elegant
    - And efficient

Exception Handling

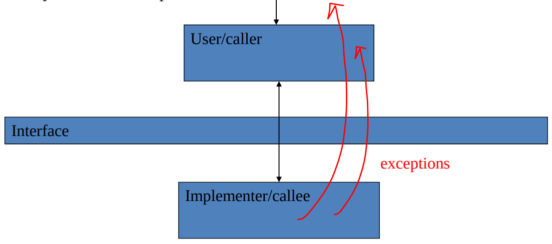
- General idea for dealing with non-local errors:
  - Caller knows (in principle) how to handle an error
    - But cannot detect it (or else if would be a local error)
  - Callee can detect an error
    - But does not know how to handle it
  - Let a caller express interest in a type of error

```
try {
        // do work
} catch (Error) {
        // handle error
}
```

  - Let a callee exit with an indication of a kind of error
    - **throw Error();**

Exception handling

- A caller asks for a task to be done
- A callee throws if unable to do the requested task
- A caller may chose to handle an exception
  - By default an exception is also an error for the caller



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 error
    }
    if (f) fclose(f);
}
```


Managing Resources

- use an object to represent a resource
  - "resource acquisition in initialization": RAII

```
class File_handle {     // belongs in some support library
        FILE* p;
public:
        File_handle(const char* pp, const char* r)        // constructor: acquire
        {       p = fopen(pp,r);
                if (p==0) throw Bad_file();
        }
        ~File_handle() { if (p) fclose(p); }              // destructor: release
        // copy operations
        // access functions
};

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

RAII for mutexes: std::lock

- From the C++0x standard library
- A lock represents local ownership of a resource (the **mutex**)

```
std::mutex m;
int sh; // shared data

void f()
{
    // ...
    std::unique_lock<mutex> lck(m);   // grab (acquire) the mutex
    // manipulate shared data:
    sh+=1;
}  // implicitly release the mutex
```


What is a “resource”?

- A resource is something
  - You acquire
  - You use
  - You release/free
  - Any or all of those steps can be implicit
- Examples
  - Free store (heap) memory
  - Sockets
  - Locks
  - Files
  - Threads

Invariants

- To recover from an error we must leave our program in a “good state”
  - Of individual objects and their relations
- Each class has a notion of what is its “good state”
  - Called its invariant
- An invariant is established by a constructor

```
class Vector {
    int sz;
    int* elem;  // elem points to an array of sz ints
public:
    vector(int s) :sz(s), elem(new int(s)) { }  // I'll discuss error handling elsewhere
    // …
};
```

Exception-safety guarantees

- Basic guarantee (for all operations)
  - The basic library invariants are maintained
  - No resources (such as memory) are leaked
- Strong guarantee (for some key operations)
  - Either the operation succeeds or it has no effects
- No throw guarantee (for some key operations)
  - The operation does not throw an exception

Provided that destructors do not throw exceptions
  - Further requirements for individual operations

Exception-safety guarantees

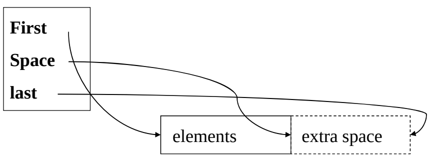
- Keys to practical exception safety
  - Partial construction handled correctly by the language

```
class X { X(int); /* … */ };
class Y { Y(int); /* … */ };
class Z { Z(int); /* … */ };
class D : X, Y { Y m1; Z m2; D(int); /* … */ };
```

  - "Resource acquisition is initialization" technique
  - Define and maintain invariants for important types

Exception safety: vector

vector:



Best **vector<T>()** representation seems to be (0,0,0)

Exception safety: vector

```
template<class T, class A = allocator<T> > class vector {
    T* v;        // start of allocation
    T* space;    // end of element sequence, start of free space
    T* last;     // end of allocation
    A alloc;     // allocator
public:
    // …
    vector(size_type n, const T& val =T(), const A& a =std::allocator());
    vector(const vector&);                // copy constructor
    vector& operator=(const vector&);     // copy assignment
    void push_back(const T&);             // add element at end
    size_type size() const { return space-v; } // calculated, not stored
    size_type capacity() const { return last-v; }
};
```


Unsafe constructor (1)

- Leaks memory and other resources
  - but does *not* create bad vectors

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
    :alloc(a)                                          // copy allocator
{
    v = a.allocate(n);                                 // get memory for elements
    space = last = v+n;
    for (T* p = v; p!=last; ++p) a.construct(p,val);   // copy val into elements
}
```


Unititialized_fill()

- offers the strong guarantee:

```
template<class For, class T>      // a standard-library algorithm
void uninitialized_fill(For beg, For end, const T& val)
{
    For p;
    try {          // construct elements:
        for (p=beg; p!=end; ++p) a.construct(&*p) T(val); // construct val in *p
    }
    catch (…) {    // undo construction:
        for (For q = beg; q!=p; ++q) q->~T();    // destroy
        throw;                                   // rethrow
    }
}
```

Unsafe constructor (2)

- Better, but it still leaks memory

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
    :alloc(a)                                   // copy allocator
{
    v = a.allocate(n);                          // get memory for elements
    space = last = uninitialized_fill(v,v+n,val);   // copy val into elements
}
```


Represent memory explicitly

```
template<class T, class A> class vector_base {    // manage space
public:
    A alloc;        // allocator
    T* v;           // start of allocated space
    T* space;       // end of element sequence, start of free space
    T* last;        // end of allocated space

    vector_base(const A&a, typename A::size_type n)
        :alloc(a),  v(a.allocate(n)), space(v+n), last(v+n) { }
    ~vector_base() { alloc.deallocate(v,last-v); }
};

// works best if a.allocate(0)==0
// we have assumed a stored allocator for convenience
```


A vector is something that provides access to memory

```
template<class T, class A = allocator<T> >
class vector : private vector_base {
    void destroy_elements() { for(T* p = v; p!=space; ++p) p->~T(); }
public:
    // …
    explicit vector(size_type n, const T& val =T(), const A& a =std::allocator());
    vector(const vector&);                      // copy constructor
    vector& operator=(const vector&);           // copy assignment
    ~vector() { destroy_elements(); }           // destructor
    void push_back(const T&);                   // add element at end
    size_type size() const { return space-v; }  // calculated, not stored
    size_type capacity() const { return last-v; }
    // …
};
```

Exception safety: vector

- Given **vector_base** we can write simple **vector** constructors that don't leak

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
    : vector_base(a,n)                    // allocate space for n elements
{
    uninitialized_fill(v,v+n,val);        // initialize
}
```


Exception safety: vector

- Given **vector_base** we can write simple **vector** constructors that don't leak

```
template<class T, class A>
vector<T,A>::vector(const vector& a)       // copy constructor
    : vector_base(a.get_allocator(),a.size())  // allocate space for a.size() elements
{
    uninitialized_copy(a.begin(),a.end(),v);    // initialize
}
```


But how do you handle errors?

- Where do you catch?
  - Keep it simple => multi-level
- Did you remember to catch?
  - Static vs. dynamic vs. no checking

reserve() is key

- That's where most of the tricky memory management reside

```
template<class T, class A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=capacity()) return;              // never decrease allocation
    vector_base<T,A> b(alloc,newalloc);            // allocate new space
    for (int i=0; i<sz; ++i) alloc.construct(&b.elem[i],elem[i]);     // copy
    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i],space); // destroy old
    swap< vector_base<T,A> >(*this,b);             // swap representations
}
```


push_back() is (now) easy

```
const int first_capacity = 4;

template<class T, class A>
void vector<T,A>::push_back(const T& val)
{
    if (sz==space) reserve(space ? 2*space : first_capacity); // get more space
    alloc.construct(&elem[sz],d);          // add d at end
    ++sz;                                  // increase the size
}
```

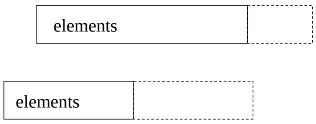

resize()

- Similarly, **vector<T,A>::resize()** is not too difficult:

```
template<class T, class A>
void vector<T,A>::resize(int newsize, T val = T())
{
    reserve(newsize);
    for (int i = sz; i<newsize; ++i) alloc.construct(&elem[i],val); // construct
    for (int i = newsize; i<sz; ++i) alloc.destroy(&elem[i]); // destroy
    sz = newsize;
}
```

Vector assignment

- To assign, we must consider two objects
  - **a= b;**
  - After **a=b;**
    - we must have **a==b**
    - **a**'s old elements have been destroyed



Exception safety: vector

- Naïve assignment (old-fashioned, unsafe and inefficient)

```
template<class T, class A >
vector<T,A>& vector<T,A>::operator=(const vector& a)
{
    destroy_elements();                          // destroy old elements
    alloc.deallocate(v);                         // free old allocation
    alloc = a.get_allocator();                   // copy allocator
    v = alloc.allocate(a.size());                // allocate
    for (int i = 0; i<a.size(); i++) v[i] = a.v[i];   // copy elements
    space = last = v+a.size();
    return *this;
}
```


Assignment with strong guarantee

```
template<class T, class A >
vector<T,A>& vector<T,A>::operator=(const vector& a)
{
    vector temp(a);                              // copy vector
    swap< vector_base<T,A> >(*this,temp);        // swap representations
    return *this;
}
```

- Note:
  - The algorithm is very simple
  - The algorithm is not optimal
    - What if the new value fits in the old allocation?
  - The implementation is optimal
  - The "naïve" assignment simply duplicated code from other parts of the vector implementation

Optimized assignment (1)

- If there is space, just copy the elements
  - (and avoid memory management)

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)
{
    if (capacity() < a.size()) {        // we must make new vector representation
        vector temp(a);                 // copy vector
        swap< vector_base<T,A> >(*this,temp);
        return *this;
    }
    if (this == &a) return *this;       // redundant self assignment check
    //  copy into existing space
    return *this;
}
```


Optimized assignment (2)

```
template<class T, class A >
Vector<T,A>& Vector<T,A>::operator=(const vector& a)
{
    // …
    size_type sz = size();
    size_type asz = a.size();
    alloc = a.get_allocator();
    if (asz<=sz) {
        copy(a.begin(),a.begin()+asz,v);
        for (T* p =v+asz; p!=space; ++p) p->~T();       // destroy surplus elements
    }
    else {
        copy(a.begin(),a.begin()+sz,v);
        uninitialized_copy(a.begin()+sz,a.end(),space); // construct extra elements
    }
    space = v+asz;
    return *this;
}
```


Optimized assignment (3)

- The optimized assignment
  - 19 lines of code
    - 3 lines for the unoptimized version
  - offers the basic guarantee
    - not the strong guarantee
  - can be an order of magnitude faster than the unoptimized version
    - depends on usage and on free store manager
  - is what the standard library offers
    - I.e. only the basic guarantee is offered
    - But your implementation may differ and provide a stronger guarantee

Exception safety

- Rules of thumb:
  - Decide which level of fault tolerance you need
    - Not every individual piece of code needs to be exception safe
  - Aim at providing the strong guarantee
    - Keep a good state (usually the old state) until you have constructed a new state; then update "atomically"
  - Always provide the basic guarantee if you can't afford the strong guarantee
  - Define "good state" (invariant) carefully
    - Establish the invariant in constructors (*not* in "**init()** functions")
  - Minimize explicit try blocks
  - Represent resources directly
    - Prefer "resource acquisition is initialization" over code where possible
    - Avoid "free standing" **new**s and **delete**s
  - Keep code highly structured ("stylized")
    - "random code" easily hides exception problems

RAII: What is the alternative?

- Commonly suggested alternative:
  - Let the constructor initialize to a default state
    - Such a constructor never fails*
  - Acquire resources later
    - If and when needed**
  - Suggested/assumed benefit
    - The constructor can't throw an exception***

* often wishful thinking
** why make an object if you don't need it yet
*** but constructers are supposed to throw when they can't establish the invariant

Part 2 overview

- Generic programming
  - Motivation
  - Lifting
  - The STL
- Classes and Class hierarchies
  - Memory management
  - Struct vs. class
  - Object-oriented programming
  - OOP vs. GP

Abstraction

- Simple user-defined types ("concrete types")
  - classes
    - Amazingly flexible
    - Zero overhead (time and space)
- Hierarchical organization ("abstract types")
  - Class hierarchies, virtual functions
    - Object-oriented programming
    - Fixed minimal overhead
- Parameterized abstractions ("generic types and functions")
  - Templates
    - Generic programming
    - Amazingly flexible
    - Zero overhead (time and space)

A class – defined

C++0x

```
class vector {        // simple vector of double
public:    // interface:
    // a constructor establishes the class invariant (acquiring resources as needed):
    vector();                              // constructor: empty vector
    vector(initializer_list<double>);      // constructor: initialize from a list
    ~vector();                             // destructor for cleanup

    double& operator[](int i);             // range checked access
    const double& operator[](int i) const; // access to immutable vector
    int size() const;

    // copy operations
private:   // representation (simplified):
    int sz;
    double* p;
};
```

Think of vector as a resource handle

A generic class – used

- "Our" vector is just an ordinary type used like any other type

```
vector  v1;                       // global variables
vector s2 = { 1, 2, 3, 4 };

void f(const vector& v)           // arguments and local variables
{
    for (int i = 0; i<v.size(); ++i) cout << v[i] << '\n';
    vector s3 = { 1, 2, 3, 5, 8, 13 };
    // …
}

struct S {
    vector s1;                    // class members
    vector s2;
};
```

No explicit resource management

A class - implemented

```
class vector {        // simple vector of double
public:
    vector() :sz(0), elem(0) { }
    vector(initializer_list<double> il) :sz(il.size()), elem(new double[sz])
          { uninitialized_copy(il.begin(), il.end(), elem); }
    ~vector() { delete[] elem; }
    double& operator[](int i)
          { if (i<0||sz<=i) throw out_of_range(); return elem[i]; }
    const double& operator[](int i) const;        // access to immutable vector
    int size() const { return sz; }
    // copy operations
private:  // representation (simplified):
    int sz;
    double* elem;
};
```

No run-time support system “magic”

A class – made generic

```
template<class T> class vector {        // simple vector of T
public:
    vector() :sz(0), elem(0) { }
    vector(initializer_list<double> il) :sz(il.size()), elem(new T[sz])
          { uninitialized_copy(il.begin(), il.end(), elem); }
    ~vector() { delete[] elem; }
    T& operator[](int i)
          { if (i<0||sz<=i) throw out_of_range(); return elem[i]; }
    const T& operator[](int i) const;   // access to immutable vector
    int size() const { return sz; }
    // copy operations
private:  // representation (simplified):
    int sz;
    T* elem;
};
```

No overheads compared to the non-generic version

A generic class – used

- “Our” vector is used just like any other type, taking its element type as an argument
  - No fancy runtime system
  - No overheads (time or space) compare to hand coding

```
vector<int> vi;
vector<double> vd = { 1.0, 2, 3.14 }; // exactly like the non-parameterized version
vector<string> vs = {"Hello", "New", "World" };
vector<vector<Coord>> vvc = {
    { {1,2,3}, {4,5,6} },
    {},
    { {2,3,4}, {3,4,5}, {4,5,6}, {5,6,7} }
};
```

C++0x

In real-world code

- We use the standard-library vector
  - Fundamentally similar to “our” vector
    - same mapping to hardware
  - More refined that “our” vector
  - As efficient (same map to hardware)
    - or better
- Or we use an industry, corporation, project “standard” container
  - Designed to cater for special needs
- Build our own
  - Using the same facilities and techniques used for the standard library
- There are tens of thousands of libraries “out there”
  - But no really good way of finding them

Generic programming

- First: parameterize containers
  - **vector<int> v;** *// vector<T> where T is int*
- Then: parameterize operations on those containers
  - **sort(v);** *// sort(vector<T>) where T is int*
- Then: parameterize those operations
  - **sort(v,abs);** *// sort(vector<T>) where T is int for absolute values*
- Then: provide specialized implementations for “special cases”
  - **sort(vector<char*>&);** *// don’t use the default sort for C-style strings*
- Then: note that templates provide a complete (Turing complete) compile-time programming language
  - Try to figure out what makes sense and what doesn’t
  - “Just because you can, doesn’t mean that you have to”

Generic programming

- A. Stepanov:
  “Aim: The most general, most efficient, most flexible representation of concepts”
  - Represent separate concepts separately in code
  - Combine concepts freely wherever meaningful

- Don’t abstract for the sake of abstraction
- Generalize from concrete examples
- Maintain (optimal) performance

Background

- Templates are great
  - Flexible
  - General
  - Great performance in time and space
  - The language base for modern generic programming in C++
  - The language base for most current high-performance work in C++
  - The language base for template meta-programming in C++
- But
  - Brittle: spectacularly bad error messages
  - Poor overloading – leading to verbosity
  - Much undisciplined hacking
  - Much spectacularly obscure code

Generic programming

- Start with a concrete algorithm
  - Or better yet: a set of related uses
- Generalize it until it makes the minimal assumptions needed
  - Without losing performance
- That's sometimes called "lifting an algorithm"
  - We go from the concrete to the more abstract
    - The other way most often leads to bloat
  - We are concerned with performance
    - Slow code will eventually be thrown away
  - Our the aim (for the end user) is
    - Greater range of uses (re-use)
    - More correctness
      - Through better specification

Lifting example (concerte algorithms)

```
double sum(double* array, int n)                    // one concrete algorithm
{                                                   // on array of doubles
    double s = 0;
    for (int i = 0; i < n; ++i ) s = s + array[i];
    return s;
}

struct Node { Node* next; int data; };

int sum(Node* first, Node* last)                    // another concrete algorithm
{                                                   // on list of ints
    int s = 0;
    while (first != last) {
        s += first->data;
        first = first->next;
    }
    return s;
}
```

Lifting example (abstract the data type)

// abstract pseudo-code for a more general version of both algorithms

```
T sum(data)          // Somehow parameterize by the value type
{
   T s = 0;
   while (not at end) {
     s = s + get value;
     get next data element;
   }
   return s;
}
```

- The data structure needs three operations:
  - not at end
  - get value
  - get next data element
- The value type needs three operations:
  - Initialize to zero
  - Add
  - Return the result

Lifting example (STL version 1)

// Concrete STL-style code for a more general version of both algorithms

```
template<class Iter>               // Iter should be an Input_iterator
Iter::value_type sum(Iter first, Iter last)
{
   Iter::value_type s = 0;        // how do we know that value_type
                                  // has initialization by 0?
   while (first!=last) {
         s = s + *first;          // why plus?
         ++first;
   }
   return s;
}
```

- The data structure is represented by a pair of iterators
  - * accesses the value
  - ++ gets next element
  - != checks if we are at the end

Lifting example (STL version 2)

// Concrete STL-style code for a more general version of both algorithms

```
template<class Iter, class T>     // Iter should be an Input_iterator
                                  // T should be something we can + and =
T sum(Iter first, Iter last, T s) // T is the "accumulator type"
{
   while (first!=last) {
         s = s + *first;          // why plus?
         ++first;
   }
   return s;
}
```

- The user initializes the accumulator

```
float a[10];
// …
double d = 0;
d = sum(a,a+10,d);
```

Lifting example (Abstract operation)

// Concrete STL-style code for a more general version of both algorithms

```
template<class Iter, class T, Class Oper>   // Iter should be an Input_iterator
                                            // T should be something we can Oper and =
T sum(Iter first, Iter last, T s, Oper op)  // T is the "accumulator type"
{
    while (first!=last) {
        s = op(s,*first);
        ++first;
    }
    return s;
}
```

- The user initializes the accumulator and supplies the operation

```
float a[10];
// …
double d = 1;  // note: 1 (rather than 0)
d = sum(a,a+10,d, Multiply<float>());
```


Lifting example

- Almost the standard library **accumulate()**
  - I simplified a bit for terseness
- Works for
  - arrays
  - **vector**s
  - **list**s
  - **istream**s
  - …
- Runs as fast as "hand-crafted" code
  - Given decent inlining
- The code's requirements on its data has become explicit
  - We understand the code better

STL

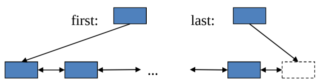
- The most prominent example of generic programming
  - Alex Stepanov and friends
  - Initially developed in 1993 +- a couple of years
    - Not Alex's first attempt (See my HOPL-3 paper)
    - The presentations of ideas have evolved a fair bit over the years
- It's widely copied
  - MTL, GIL, …
- I has articulated principles
- It is reasonably realized in C++
  - relying heavily on templates
  - Note: a *decent* match but not a *perfect* match on ideals
- It provides
  - a very useful set of ideas for how to structure code
  - Some very useful examples illustrating those ideas

Other notions of GP

- Uses of **void*** (in C and C++)
- Reliance on abstract classes as interfaces from generic algorithms
  - Textbooks, Eiffel, Java, C#
- Ada
  - E.g. Stepanov & Musser book
  - Ultimately not successful
- Essentially all run-time typed code could be deemed generic
  - The code works for all arguments for which it works
    - You get run-time errors where the C++ equivalent wouldn't compile
  - This kind of use tends to be unarticulated and ad hoc
  - Stepanov & Musser tried (unsuccessfully)to use Scheme
- Data-generic programming
  - Functional programming research

STL iterators

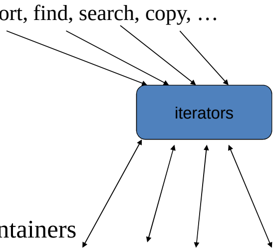
- An iterator denotes (points to, refers to) an element of a sequence
- A sequence is defined by a pair of iterators
  - A sequence is half open [first:last)
  - An empty sequence has first==last
- There are many different iterator types
  - A vector<int> iterator is not a list<int> iterator
  - There is no iterator class that is common to all iterators
  - Every iterator operation have the same semantics for every iterator
- Not all iterators provide the same set of operations



Basic iterator model

- Algorithms

  sort, find, search, copy, …



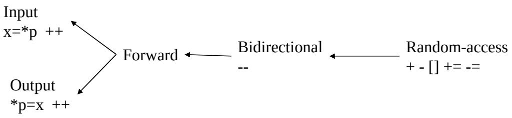
- Containers

  vector, list, map, hash_map, …

- Separation of concerns
  - Algorithms manipulate data, but don't know about containers
  - Containers store data, but don't know about algorithms
  - Algorithms and containers interact through iterators
    - Each container has its own iterator types

STL iterator categories

- Iterator categories
  - Note: not a class hierarchy



- Important
  - there are just five categories
  - Imagine the mess/complexity if there had been 17

STL iterator categories

- Most important use: “overload on iterator category”

```
// First try:

template<class Forward_iterator > void advance(Iter p, int n)
{
    while(n--) ++p;     // slow; often very slow
}

template<class Random_access_iterator > void advance(Iter p, int n)
{
    p+=n;     // fast
}
```

- Obviously(?) doesn’t work

STL iterator categories

- Use helper functions based on traits (compile-time resolution!)

```
template<class Iter> void advance(Iter p, int n)     // STL function
{
    advance_helper(p,n,iterator_category<p>);
}

// “implementation details”:
template<class Iter> void advance_helper(Iter p, int n, forward_iterator)
{
    while(n--) ++p;     // slow; often very slow
}

template<class Iter> void advance_helper(Iter p, int n, random_access_iterator)
{
    p+=n;     // fast
}
```

Traits and categories

- Indirect use of advance():

```
template<class ForwadIterator>
void algo(ForwardIterator b, ForwardIterator e)
{
   // …
   algo(b,b+advance(b.size()/2);
   // …
}

vector<int> v(100000);
// …
algo(v.begin(),v.end());
```

- If **algo()** used the **forward_iterator** version of **advance()** we'd have an N^2 algorithm rather than an N log(N) one

Operations

- Functions

```
bool  greater(int a, int b) { return a>b; }

qsort(&v.begin(),v.size(),sizeof(int),greater);   // indirect function call
sort(v.begin(),v.end(),greater);                  // direct function call
```

- Function objects

```
struct Greater {
   bool operator()(int a, int b) const { return a>b; }
};

sort(v.begin(), v.end(),Greater());       // inlined function
```


Generic Programming: Operations

- But it is useful?

```
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());
```

Generic Programming: Operations

```
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);
    }
};
```

80

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)

Generic Programming: function objects

- 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 standard library
- Key to emulating functional programming techniques

Specialization and overloading

- Some types are "odd" (i.e., their semantics is not what it appears to be)
  - Especially pointers and arrays
    - **if (s>s2)** does not compare C-style strings **s** and **s2**
    - **p=a;** does not copy the built-in array **a** into the array pointed to by **p**
- Provide "ad hoc" common interfaces
  - Overload function templates
    - **template<class T> void sort(vector<T&>);**
    - **void sort(vector<const char*>&);**
  - Specialize class templates
    - **template<class T> class vector { /* … */ };**
    - **template<> class vector<char*> { /* … */ };**

Classes and class hierarchies

- Struct vs. class
- Object-oriented programming
- OOP vs. GP

Struct and class

- Use **struct** if you can't define an invariant

```
struct Address {        // "Plain Old Data" (POD)
                        // the variations of names and addresses worldwide
                        // defeats attempts to validate
    string name;
    string address;
};
```

- Define an invariant for every class (that's not a **struct**)
  - Establish invariant in constructor
    - Acquire any needed resources
    - Throw exception if you cannot establish invariant
  - Provide a way of checking or enforcing the invariant

Classes

- Why bother with the public/private distinction?
- Why not make everything public?
  - To provide a clean interface
    - Data and messy functions can be made private
  - To maintain an invariant
    - Only a fixed set of functions access the data
    - May lead to get and set functions (avoid if you can)
  - To ease debugging
    - Only a fixed set of functions access the data
    - (known as the "round up the usual suspects" technique)
  - To allow a change of representation
    - You need only to change a fixed set of functions
    - You don't really know who is using a public member

APT tutorial - Stroustrup 86

Classes

- What makes a good interface?
  - Minimal
    - As small as possible
  - Complete
    - And no smaller
  - Type safe
    - Beware of confusing argument orders
  - Const correct
    - Immutable is the ideal
- What operations need direct access to data?
  - Logical necessity
  - Performance requirement
    - how often used?
    - Is there a check for each call?
    - Needs to be inlined?

APT tutorial - Stroustrup 87

Inheritance

- Benefits
- Problems
- Abstract classes
- Protected
- OOP vs. GP
  - No: object-oriented programming plus generic programming

APT tutorial - Stroustrup 88

Benefits of inheritance

- Interface inheritance
  - A function expecting a shape (a **Shape&**) can accept any object of a class derived from **Shape**.
    - Simplifies use (sometimes dramatically)
  - We can add derived classes to a program without rewriting user code
    - Adding without touching old code is one of the "holy grails" of programming
- Implementation inheritance
  - Simplifies implementation of derived classes
    - Common functionality can be provided in one place
    - Changes can be done in one place and have universal effect
      - Another "holy grail"

Problems with inheritance

- Anyone can provide a derived class that overrides a virtual function
  - "insanity is hereditary; you can get it from your kids"
- Anyone can provide a derived class with a larger object size
  - Arrays + inheritance == trouble
- You cannot change a base class after deploying it
  - Unless you can get all users to recompile
  - Well, maybe you can add virtual functions (but that's cheating)
  - PIMPL idiom
- Manipulate a class from a hierarchy though a pointer or reference
  - Abstract classes is key
  - Do remember to provide a virtual destructor (if you have any virtual function)
  - Note performance implications:
    - Kills inlining
    - Ensures large footprint
    - Ensures per-object memory overhead

Use abstract classes as interfaces to users

- Abstract (interface inheritance)

```
struct Shape {
    // no data no constructors
    virtual void draw() = 0;
    // …
    virtual void ~Shape() = 0;
};
```

  - The most abstract and least brittle
    - within the bounds of OOP

Use implementation inheritance for implementation

- Concrete (implementation inheritance)

```
class Shape {
    Point center;
    // …
public:
    virtual void draw();
    // …
    virtual void ~Shape();
};
```

  - Can lead to maintenance problems
    - You can't update the data part without complete recompilation of all users
  - Ideal where you control the set of users
    - So use it for your implementation and give users a pure interface

You can separate implementation and interface hierarchies

- Forwarding (e.g. Pimpl)

```
class Shape {
    class Shape_impl* p;  // points to a concrete/simple/implementation hierarchy
public:
    void draw() ;
    // …
    void ~Shape() ;
};

// elsewhere:
    class Shape_impl { /* … */ };
    void draw() { p->draw(); }
    void ~Shape() { delete p; }
```

  - Well, you could give a complete talk about the details of doing this
    - Should Shape have constructors? (yes)
    - Is defining these forwarding functions in-class a big mistake? (yes)
    - Should the forwarding functions be virtual and Shape a base class? (maybe)
    - …

Protected

- Basic idea:
  - Make members accessible to derived class members but not to "the general public"
- Too crude
  - People who write derived classes ***are*** "the general public"
    - They mess with protected data in incautious ways, causing maintenance problems
    - The "brittle base class problem" reemerges
  - Protected member functions and protected inheritance
    - Seem not to cause the problems of protected data
    - Seem essential for ***lots*** of OO techniques

Hierarchy vs. parameterization

- OOP
  - Run time
    - Resolution implies run-time error handling
  - Ad hoc
  - Often a focus on data presented by classes
- GP
  - Compile time (link time)
    - Resolution implies much more attention to type system
  - Often a focus on algorithms
- I don't see a *fundamental* tension
  - We need data *and* algorithms
  - We need ad hoc code *and* (more formal) algorithms
  - lots of difficult tradeoffs, though

Hierarchy ***and*** parameterization

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

template<class C> void draw_all(C& c)       // for all 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));
}
```


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
- Soon, I'll find a proper name for "Multiparadigm programming"

Memory management

- Ad hoc (who would do that in 2010? ☹)
  - For a large program, “naked” **new** and **delete** leads to
    - Memory leaks
    - Memory corruption (write to freed memory)
- Library supported discipline
    - Containers
    - Scope-based techniques (scoped roots)
- Smart pointers – though not a panacea
    - Cost
    - Race conditions
- GC – though not a panacea
    - Sometimes, the other techniques get messy
    - Sometimes, you need to live with code written by people who think “ad hoc” is cool

Part 3 C++0x

- ISO Standardization
- Aims
- Design rules and examples
- What is C++?
- Case study: Concurrency

C++ ISO Standardization

- Slow, bureaucratic, democratic, formal process
  - “the worst way, except for all the rest”
    - (apologies to W. Churchill)
- About 22 nations
  - (5 to 12 at a meeting)
- Membership have varied
  - 100 to 200+ active
  - 40 to 100 at a meeting
- ISO
  - Started work 1990
  - First standard in 1998
  - C++0x “Final Draft” 2010
    - C++ 0x will be C++11
- Most members work in industry
- Most members are volunteers
  - Even many of the company representatives
- Most major platform, compiler, and library vendors are represented
  - E.g., IBM, Intel, Microsoft, Sun
- End users are underrepresented

Overall goals for C++0x

- 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-style application development)
  - Build directly on C++'s contributions to systems programming

- 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)

Stroustrup - Rapperswil - 2010 101

C++0x

- 'x' may be hex, but C++0x is not science fiction
  - Every feature is implemented somewhere. E.g.:
    - GCC 4.6: Rvalues, Variadic templates, Initializer lists, Static assertions, auto-typed variables, New function declarator syntax, Lambdas, Right angle brackets, Extern templates, Strongly-typed enums, Delegating constructors (patch), Raw string literals, Defaulted and deleted functions, Inline namespaces, Local and unnamed types as template arguments, new for statement, …
    - Microsoft: lambdas, concurrency
  - Standard library components are shipping widely
    - E.g. GCC, Microsoft, Boost
  - The last design points have been settled
    - We are now processing formal requests from National Standards Bodies

Stroustrup - Rapperswil - 2010 102

Rules of thumb / Ideals

- Integrating features to work in combination is the key
  - And the most work
  - The whole is much more than the simple sum of its part

- Maintain stability and compatibility
- Prefer libraries to language extensions
- Prefer generality to specialization
- Support both experts and novices
- Increase type safety
- Improve performance and ability to work directly with hardware
- Make only changes that change the way people think
- Fit into the real world

Stroustrup - Rapperswil - 2010 103

Support both experts and novices

- *Example*: minor syntax cleanup
  vector<list<int>> v; *// note the "missing space"*
- *Example: deduced type:*
  auto x = v.begin(); *// x becomes a vector<list<int>>::iterator*
- *Example*: simplified iteration
  for (auto x : v) cout << x <<'\n';
- *Note*: Experts don't easily appreciate the needs of novices
  - Example of what we couldn't get just now
    string s = "12.3";
    double x = lexical_cast<double>(s); *// extract value from string*

Stroustrup - Rapperswil - 2010 104

Uniform initialization

- You can use **{}**-initialization for all types in all contexts

```
int a[] = { 1,2,3 };
vector<int> v { 1,2,3};

vector<string> geek_heros = {
   "Dahl", "Kernighan", "McIlroy", "Nygaard ", "Ritchie", "Stepanov"
};

thread t{};    // default initialization
               // remember "thread t();" is a function declaration

complex<double> z{1,2};         // invokes constructor
struct S { double x, y; } s {1,2};  // no constructor (just initialize members)
```

Stroustrup - Rapperswil - 2010 105

Uniform initialization

- **{}**-initialization **X{v}** yields the same value of **X** in every context

```
X x{a};
X* p = new X{a};
z = X{a};       // use as cast

void f(X);
f({a});         // function argument (of type X)

X g()
{
   // …
   return {a}; // function return value (function returning X)
}

Y::Y(a) : X{a} { /* … */ };       // base class initializer
```

Stroustrup - Rapperswil - 2010 106

Not a reference

Move semantics

- Often we don't want two copies, we just want to move a value

```
vector<int> make_test_sequence(int n)
{
    vector<int> res;
    for (int i=0; i<n; ++i) res.push_back(rand_int());
    return res; // move, not copy
}

vector<int> seq = make_test_sequence(1000000);        // no copies
```

- New idiom for arithmetic operations:
  - **Matrix operator+(const Matrix&, const Matrix&);**
  - **a = b+c+d+e;** // *no copies*

Improve performance and the ability to work directly with hardware

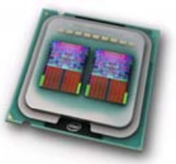
- Embedded systems programming is very important
  - *Example*: address array/pointer problems
    - **array<int,7> s;** // *fixed-sized array*
  - *Example*: Generalized constant expressions (think ROM)

```
constexpr int abs(int i) { return (0<=i) ? i : -i; } // can be constant expression

struct Point {
    int x, y;
    constexpr Point(int xx, int yy) : x{xx}, y{yy} { }    // "literal type"
};

constexpr Point p{1,2};              // must be evaluated at compile time: ok
constexpr Point p2{p.x,abs(x)};   // ok?: is x is a constant expression?
```


Areas of language change



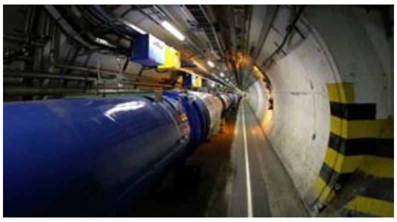
- Machine model and concurrency Model
  - Threads library (**std::thread**)
  - Atomics ABI
  - Thread-local storage (**thread_local**)
  - Asynchronous message buffer (**std::future**)
- Support for generic programming
  - (no concepts ☹)
  - uniform initialization
  - **auto**, **decltype**, lambdas, template aliases, move semantics, variadic templates, range-**for**, …
- Etc.
  - **static_assert**
  - improved **enum**s
  - **long long**, C99 character types, etc.
  - …

Standard Library Improvements

- New containers
  - Hash Tables (**unordered_map**, etc.)
  - Singly-linked list (**forward_list**)
  - Fixed-sized array (**array**)
- Container improvements
  - Move semantics (e.g. **push_back**)
  - Initializer-list constructors
  - Emplace operations
  - Scoped allocators
- More algorithms (just a few)
- More and better utilities
  - **bind()**, **function**, …
- Concurrency support
  - **thread**, **mutex**, **lock**, …
  - **future**, **async**, …
  - Atomic types
- Garbage collection ABI

Standard Library Improvements

- Regular Expressions (**regex**)
- General-purpose Smart Pointers (**unique_ptr**, **shared_ptr**, …)
- Extensible Random Number Facility
- Enhanced Binder and function wrapper (**bind** and **function**)
- Mathematical Special Functions
- Tuple Types (**tuple**)
- Type Traits (lots)



What is C++?

Template meta-programming!

A hybrid language

A multi-paradigm programming language

Buffer overflows

It's C!

Embedded systems programming language

Too big!

Supports generic programming

Low level!

An object-oriented programming language

A random collection of features

C++

Key strength:
Building software infrastructures and resource-constrained applications

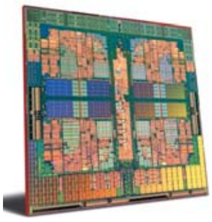
A light-weight abstraction programming language

Thanks!

- C and Simula
  - Brian Kernighan
  - Doug McIlroy
  - Kristen Nygaard
  - Dennis Ritchie
  - …
- ISO C++ standards committee
  - Steve Clamage
  - Francis Glassborow
  - Andrew Koenig
  - Tom Plum
  - Herb Sutter
  - …
- C++ compiler, tools, and library builders
  - Beman Dawes
  - David Vandevoorde
  - …
- Application builders

Case study

- Concurrency
  - "driven by necessity"
  - More than ten years of experience



Case study: Concurrency

- What we want
  - Ease of programming
    - Writing correct concurrent code is hard
  - Portability
  - Uncompromising performance
  - System level interoperability
- We can't get everything
  - No one concurrency model is best for everything
  - De facto: we can't get all that much
  - "C++ is a systems programming language"
    - (among other things) implies serious constraints

Concurrency: std::thread

```
#include<thread>
void f() { std::cout << "Hello "; } // function
struct F {          // function object
    void operator()() { std::cout << "parallel world "; }
};
int main()
{
    std::thread t1{f};      // f() executes in separate thread
    std::thread t2{F()};    // F()() executes in separate thread

    t1.join();      // wait for t1
    t2.join();      // wait for t2
} // spot the bug
```

Thread – pass arguments

- Use bind() or variadic constructor

```
void f(vector<double>&);

struct F {
    vector<double>& v;
    F(vector<double>& vv) :v{vv} { }
    void operator()();
};

int main()
{
    std::thread t1{std::bind(f,some_vec)};        // f(some_vec)
    std::thread t2{f,some_vec};          // f(some_vec)
    t1.join(); t2.join();
}
```

Stroustrup - Rapperswil - 2010 119

Mutual exclusion: std::mutex

- A **mutex** is a primitive object use for controlling access in a multi-threaded system.
- A **mutex** is a shared object (a resource)
- Simplest use:

```
std::mutex m;
int sh; // shared data
// ...
m.lock();
    // manipulate shared data:
    sh+=1;
m.unlock();
```

Stroustrup - Rapperswil - 2010 120

Mutex – try_lock()

- Don't wait unnecessarily

```
std::mutex m;
int sh; // shared data
// ...
if (m.try_lock()) { // manipulate shared data:
    sh+=1;
    m.unlock();
else {
    // maybe do something else
}
```

Stroustrup - Rapperswil - 2010 121

RAII for mutexes: std::lock

- A lock represents local ownership of a resource (the **mutex**)

```
std::mutex m;
int sh; // shared data

void f()
{
    // ...
    std::unique_lock<mutex> lck(m);   // grab (acquire) the mutex
    // manipulate shared data:
    sh+=1;
}   // implicitly release the mutex
```

Stroustrup - Rapperswil - 2010 122

Potential deadlock

- Unstructured use of multiple locks is hazardous:

```
std::mutex m1;
std::mutex m2;
int sh1; // shared data
int sh2;
// ...
void f() {
    // ...
    std::unique_lock<mutex> lck1(m1);
    std::unique_lock<mutex> lck2(m2);
    // manipulate shared data:
    sh1+=sh2;
}
```

Stroustrup - Rapperswil - 2010 123

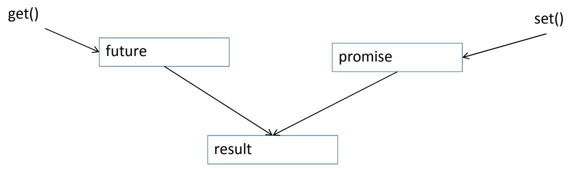
RAII for mutexes: std::lock

- We can safely use several locks

```
void f() {
    // ...
    std::unique_lock<mtex> lck1(m1,std::defer_lock);   // don't yet acquire
    std::unique_lock<mutex> lck2(m2,std::defer_lock);
    std::unique_lock<mutex> lck3(m3,std::defer_lock);
    // …
    lock(lck1,lck2,lck3);
    // manipulate shared data
}   // implicitly release the mutexes
```

Stroustrup - Rapperswil - 2010 124

Future and promise



- future+promise provides a simple way of passing a value from one thread to another
  - No explicit synchronization
  - Exceptions can be transmitted between threads

Future and promise

- Get an X from a **future<X>**:

  X v = f.get();// *if necessary wait for the value to get*

- Put an X to a **promise<X>**:

```
try {
    X res;
    // compute a value for res
    p.set_value(res);
} catch (...) {
    // oops: couldn't compute res
    p.set_exception(std::current_exception());
}
```

async()

- Simple launcher using the variadic template interface

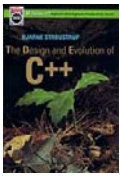
```
double accum(double* b, double* e, double init);

double comp(vector<double>& v) // spawn many tasks if v is large enough
{
    if (v.size()<10000) return accum(&v[0], &v[0]+v.size(), 0.0);

    auto f0 = async(accum, &v[0], &v[v.size()/4], 0.0);
    auto f1 = async(accum, &v[v.size()/4], &v[v.size()/2], 0.0);
    auto f2 = async(accum, &v[v.size()/2], &v[v.size()*3/4], 0.0);
    auto f3 = async(accum, &v[v.size()*3/4], &v[0]+v.size(), 0.0);

    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

More information



- My home pages
  - C++0x FAQ
  - Papers, FAQs, libraries, applications, compilers, …
    - Search for “Bjarne” or “Stroustrup”
    - “What is C++0x ?” paper
- My HOPL-II and HOPL-III papers
- The Design and Evolution of C++ (Addison Wesley 1994)
- The ISO C++ standard committee’s site:
  - All documents from 1994 onwards
    - Search for “WG21”
- The Computer History Museum
  - Software preservation project’s C++ pages
    - Early compilers and documentation, etc.
      - http://www.softwarepreservation.org/projects/c_plus_plus/
      - Search for “C++ Historical Sources Archive”

Stroustrup - Rapperswil - 2010 128