Two C++0x case studies

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

Concurrency support
“Concepts”
What we want

• Ease of programming
  – Writing correct concurrent code is hard
  – Modern hardware is concurrent in more way than you imagine

• Uncompromising performance
  – But for what?

• Portability
  – Preferably portable performance

• System level interoperability
  – C++ shares threads with other languages and with the OSs
We can’t get everything

• No one concurrency model is best for everything

• We can’t get all that much
  – C++0x is not a research project
  – WG21 has very few resources (time, people)
  – “Don’t break my code!”

• “C++ is a systems programming language”
  – (among other things) implies serious constraints
Concurrency

• Not
  – Massively parallel (scientific) computing
  – Web services
  – Simple high-level abstract model
  – System of real-time guarantees

• Instead
  – A systems-level foundation for all
Concurrency overview

• Foundation
  – Memory model
  – atomics

• Concurrency library components
  – std::thread
  – std::mutex (several)
  – std::lock (several)
  – std::condition (several)
  – std::future, std::promise, std::packaged_task
  – std::async()

• Resource management
  – std::unique_ptr, std::shared_ptr
  – GC ABI
Memory model

- A memory model is an agreement between the machine architects and the compiler writers to ensure that most programmers do not have to think about the details of modern computer hardware.

```c
// thread 1:
char c;
c = 1;
int x = c;

// thread 2:
char b;
b = 1;
int y = b;
```

`x==1` and `y==1` as anyone would expect
(but don’t try that for two bitfields of the same word)
Memory model

• Two threads of execution can update and access separate memory locations without interfering with each other.

• But what is a “memory location?”
  – A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width.
  – For example, here S has exactly four separate memory locations:

        struct S {
            char a;        // location #1
            int b:5;      // location #2
            unsigned c:11;
            unsigned :0;  // note: :0 is "special"
            unsigned d:8;    // location #3
            struct {int ee:8;} e;   // location #4
        };

Stroustrup - Oslo
Atomics (“here be dragons!”)

- Components for fine-grained atomic access
  - provided via operations on atomic objects (in `<cstdatomic>`)
  - Low-level, messy, and shared with C (making the notation messy)
  - what you need for lock-free programming
  - what you need to implement `std::thread`, `std::mutex`, etc.
  - Several synchronization models, CAS, fences, …

```cpp
enum memory_order { // regular (non-atomic) memory synchronization order
    memory_order_relaxed, memory_order_consume, memory_order_acquire,
    memory_order_release, memory_order_acq_rel, memory_order_seq_cst
};

C atomic_load_explicit(const volatile A* object, memory_order);
void atomic_store_explicit(volatile A *object, C desired, memory_order order);
bool atomic_compare_exchange_weak_explicit(volatile A* object, C *expected, C desired, memory_order success, memory_order failure);
```

// ... lots more ...

Stroustrup - Tandberg 2009
Threading

- You can
  - wait for a thread for a **specified time**
  - control access to some data by **mutual exclusion**
  - control access to some data using **locks**
  - wait for an action of another task using a **condition variable**
  - return a value from a thread through a **future**
Concurrency: std::thread

#include<thread>

void f() { std::cout << "Hello "; }

struct F {
    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
} // spot the bugs
Concurrency: std::thread

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
}

// and another bug: don’t write to cout without synchronization
Thread – pass arguments

- Use function object or bind()

```cpp
void f(vector<double>&);
struct F {
  vector<double>& v;
  F(vector<double>& vv) : v{vv} {}  // F(some_vec)
  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();
}
```
Thread – pass arguments

- Use `bind()` or variadic constructor

```cpp
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 - Oslo
Thread – pass result (primitive)

```cpp
void f(vector<double>&, double* res); // place result in res
struct F {
    vector& v; double* res;
    F(vector<double>& vv, double* p) :v{vv}, res{p} { }
    void operator()(); // place result in res
};

int main()
{
    double res1; double res2;
    std::thread t1{f,some_vec,&res1}; // f(some_vec,&res1)
    std::thread t2{F,some_vec,&res2}; // F(some_vec,&res2)()
    t1.join(); t2.join();
    std::cout << res1 << ' ' << res2 << '
';
}
```
No cancellation/interruption

• When a **thread** goes out of scope the program is **terminate()**d unless its task has completed. That's obviously to be avoided.
• There is no way to request a **thread** to terminate (i.e. request that it exit as a soon as possible and as gracefully as possible) or to force a thread to terminate (i.e. kill it). We are left with the options of
  • designing our own cooperative ``interruption mechanism'' (with a piece of shared data that a caller thread can set for a called thread to check (and quickly and gracefully exit when it is set)),
  • ``going native'' (using **thread::native_handle()** to gain access to the operating system's notion of a thread),
  • kill the process (**std::quick_exit()**),
  • kill the program (**std::terminate()**).
Mutual exclusion: std::mutex

- A **mutex** is a primitive object used 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();
  ```
Mutex – try_lock()

• Don’t wait unnecessarily

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

• Don’t wait for too long:

```cpp
std::timed_mutex m;
int sh; // shared data
// ...
if (m.try_lock_for(std::chrono::seconds(10))) { // Note: time
    // manipulate shared data:
    sh+=1;
    m.unlock();
} else {
    // we didn't get the mutex; do something else
}
```
Mutex – try_lock_until()

• We can wait until a fixed time in the future:

```cpp
std::timed_mutex m;
int sh; // shared data

// ...
if (m.try_lock_until(midnight)) { // manipulate shared data:
    sh+=1;
    m.unlock();
} else {
    // we didn't get the mutex; do something else
}
```
Recursive mutex

- In some important use cases it is hard to avoid recursion

```cpp
std::recursive_mutex m;
int sh; // shared data
// ...

void f(int i)
{
    // ...
    m.lock();
    // manipulate shared data:
    sh+=1;
    if (--i>0) f(i);
    m.unlock();
    // ...
}
```
RAII for mutexes: std::lock

- A lock represents local ownership of a non-local resource (the mutex)
  
  ```cpp
  std::mutex m;
  int sh; // shared data
  
  void f()
  {
    // ...
    std::unique_lock lck(m); // grab (acquire) the mutex
    // manipulate shared data:
    sh+=1;
  } // implicitly release the mutex
  ```
Potential deadlock

- Unstructured use of multiple locks is hazardous:

```cpp
std::mutex m1;
std::mutex m2;
int sh1; // shared data
int sh2;
// ...
void f() {
    // ...
    std::unique_lock lck1(m1);
    std::unique_lock lck2(m2);
    // manipulate shared data:
    sh1+=sh2;
}
```
RAII for mutexes: std::lock

• We can safely use several locks
  
  ```cpp
  void f() {
      // ...
      std::unique_lock lck1(m1,std::defer_lock);  // make locks but don't yet
      // try to acquire the mutexes

      std::unique_lock lck2(m2,std::defer_lock);
      std::unique_lock lck3(m3,std::defer_lock);
      // ...
      lock(lck1,lck2,lck3);
      // manipulate shared data
  }  // implicitly release the mutexes
  ```

Stroustrup - Tandberg 2009
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 from a `future<X>` called `f`:
  ```cpp
  X v = f.get(); // if necessary wait for the value to get
  ```

- Put to a `promise<X>` called `p` (attached to `f`):
  ```cpp
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

```cpp
template<class T, class V> struct Accum {
    // accumulator function object
};

void comp(vector<double>& v) // spawn many tasks if v is large enough
{
    if (v.size()<10000) return std::accumulate(v.begin(),v.end(),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[v.size()],0.0});
    return f0.get()+f1.get()+f2.get()+f3.get();
}
```
async()

- Simple launcher using the variadic template interface

```cpp
template<class T, class V> struct Accum {
    // accumulator function object
};

void comp(vector<double>& v) // spawn many tasks if v is large enough
{
    if (v.size()<10000) return std::accumulate(v.begin(),v.end(),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[v.size()],0.0);
    return f0.get()+f1.get()+f2.get()+f3.get();
}
```
Generic programming: The language is straining

- Fundamental cause
  - The compiler doesn’t know what template argument types are supposed to do and not do
    - We don’t tell it
    - Much interface specification is in the documentation/comments
- Use requires too many clever tricks and workarounds
  - Works beautifully for correct code
    - Uncompromising performance is usually achieved
      - After much effort
  - Users are often totally baffled by simple errors
    - Poor error messages
      - Amazingly so!
    - Late checking
      - At template instantiation time
- The notation can be very verbose
  - Pages of definitions for things that’s logically simple
Example of a problem

// standard library algorithm fill():
// assign value to every element of a sequence

template<class For, class V>
void fill(For first, For last, const V& v)
{
    while (first!=last) {
        *first = v;
        first=first+1;
    }
}

fill(a,a+N,7);  // works for an array
fill(v.begin(), v.end(),8);  // works for a vector

fill(0,10,8);    // fails spectacularly for a pair of ints
fill(lst.begin(),lst.end(),9);  // fails spectacularly for a list!
What’s right in C++98?

• Parameterization doesn’t require hierarchy
  – Less foresight required
    • Handles separately developed code
    – Handles built-in types beautifully
• Parameterization with non-types
  – Notably integers
• Uncompromised efficiency
  – Near-perfect inlining
• Compile-time evaluation
  – Template instantiation is Turing complete

We try to strengthen and enhance what works well
C++0x: Concepts

• “a type system for C++ types”
  – and for relationships among types
  – and for integers, operations, etc.

• Based on
  – Search for solutions from 1985 onwards
    • Stroustrup (see D&E)
    • Lobbying and ideas for language support by Alex Stepanov
  – Analysis of design alternatives
    • 2003 papers (Stroustrup, Dos Reis)
  – Designs by Dos Reis, Gregor, Siek, Stroustrup, …
    • Many WG21 documents
  – Academic papers:
    • POPL 2006 paper, OOPSLA 2006 papers
  – Experimental implementations (Gregor, Dos Reis)
  – Experimental versions of libraries (Gregor, Siek, …)

• Removed from C++0x (July 2009)
  – Stroustrup papers
Concept aims

• Direct expression of intent
  – Separate checking of template definitions and template uses
    • Implying radically better error messages
    • We can almost achieve perfection
  – Increase expressiveness overloading
  – Simple tasks are expressed simply
    • close to a logical minimum
  – Simplify all major current template programming techniques

• No performance degradation compared to current code
  – Non-trivial and important
    • Design and implementation

• Relatively easy implementation within current compilers
  – For some definition of “relatively easy”

• Current template code remains valid
Checking of uses

• The checking of use happens immediately at the call site and uses only the declaration

```cpp
template<Forward_iterator For, class V>
    requires Assignable<For::value_type,V>
void fill(For first, For last, const V& v);    // <<< just a declaration, not definition

int i = 0;
int j = 9;
fill(i, j, 99);    // error: int is not a Forward_iterator (int has no prefix *)

int* p = &v[0];
int* q = &v[9];
fill(p, q, 99);    // ok: int* is a Forward_iterator
```
Checking of definitions

- Checking at the point of definition happens immediately at the definition site and involves only the definition

```cpp
template<Forward_iterator For, class V>
    requires Assignable<For::value_type,V>
void fill(For first, For last, const V& v)
{
    while (first!=last) {
        *first = v;
        first=first+1;   // error: + not defined for Forward_iterator
                         // (instead: use ++first)
    }
}
```
Concept maps

• We need to map types defined “elsewhere” into our type system
  – e.g. a pointer (invented 1974) into an Forward Iterator (invented 1992)

```cpp
template<Value_type T>
concept_map Forward_iterator<T*> {    // T*’s value_type is T
    typedef T value_type;
};
```

// “when we consider T* a Forward_Iterator, the value_type of T* is T
// value_type is an associated type of Forward_iterator

• “Concept maps” is a general mechanism for non-intrusive mapping of types to requirements
Expressiveness

• Simplify notation through overloading:

```cpp
void f(vector<int>& vi, list<int>& lst, Fct f)
{
    sort(vi); // sort container (vector) using <
    sort(vi, f); // sort container (vector) using f
    sort(lst); // sort container (list) using <
    sort(lst, f); // sort container (list) using f
    sort(vi.begin(), vi.end()); // sort sequence using <
    sort(vi.begin(), vi.end(), f); // sort sequence using f
}
```

• Currently, this requires a mess of helper functions and traits
  – For this example, some of the traits must be explicit (user visible)
Concepts as predicates

- A concept can be seen as a predicate:
  - `Forward_iterator<T>`: Is type T a `Forward_iterator`?
  - `Assignable<T::value_type,V>`: can we assign a V to T’s `value_type`?
- So we can do overload resolution based on simple sets of concepts:

- Intersection: ambiguous
- Disjoint: independent (ok)
- Subset: specialization (ok, pick the most specialized)
Expressiveness

// iterator-based standard sort (with concepts):

template<Random_access_iterator Iter>
    requires Comparable<Iter::value_type>
void sort(Iter first, Iter last);  // the usual implementation


template<Random_access_iterator Iter, Compare Comp>
    requires Callable<Comp, Iter::value_type>
void sort(Iter first, Iter last, Comp comp);  // the usual implementation
Expressiveness

// container-based sort:

template<Container Cont>
    requires Comparable<Cont::value_type>
void sort(Cont& c)
{
    sort(c.begin(),c.end());
}

template<Container Cont, Compare Comp>
    requires Callable<Comp, Cont::value_type>
void sort(Cont& c, Comp comp)
{
    sort(c.begin(),c.end(),comp);
}
Defining concepts

concept Forward_iterator<typename Iter> // Iter is a Forward_iterator
    : Input_iterator<Iter> // a Forward_iterator is an Input_iterator
    && Output_iterator<Iter> // a Forward_iterator is an Output_iterator
requires Default_constructible<Iter> && Assignable<Iter>
{
    // Input_iterator defines the associated type value_type

    // associated functions:
    Iter& operator=(const Iter&); // assignment yields lvalue;
    Iter& operator++(Iter&); // pre-increment yields lvalue
    const Iter& operator++(Iter&, int); // post-increment yields rvalue
    Iter::value_type operator*(Iter); // the result of * can be
    // assigned to Iter’s value_type
};
Concepts: semantic properties

- An axiom is an assertion that a compiler or a tool may take for granted
  - For external tools
  - Maybe later for resolution
  - We can use predicate logic (only)

```cpp
concept Number<class N> {
    // ...
    axiom commutative(N a, N b) {
        a+0 <=> a;
        0+a <=> a;
        a+b <=> b+a;
        // ...
    }
}
```
2009 Reflections

• Concepts for C++0x were defeated by
  – Fear of the new, fear of the unknown and unknowable
  – Fear of compilation overheads
  – Inflexibility relative to “status quo” definition/implementation of concepts
  – Fear that the specification was unstable
  – Fear of complexity (specifications and use)
  – Fear that the “status quo” design would not scale (program size and usability)

• “Schedule” was IMO only a small part of the problem

• We have to reevaluate and redesign from scratch
  – Of course we now have much more experience
  – We must work harder on simplicity (teachability and scalability)
  – We must work harder on validation
  – Implementation is no substitute for design (and vise versa)
2009 Reflections

• We were obsessed with the STL and monoids
  – Fundamental programming ideas must be validated in several application domains
  – Grad students are not a good substitute for “Real programmers”
Scale

• Composition must be easy
  – How modular?
  – Don’t damage run-time performance

• Think in terms of million-line programs
  – Incremental adoption is essential

• Think in terms of millions of programmers

• Compilation, linking, and run-time speed
Simple to use

• Most C++ programmers don’t know type theory
  – For “type theory” think ‘anything written with Greek letters’
  – And will not learn it

• Many C++ programmers dislike type theory
  – Even if they don’t know it
  – Even if they do know it

• What the end user have to write must not increase significantly
  – Real Programmers™ *hate* writing what they consider redundant
  – It must be easy to give a reason for anything a user must write

• Beware of cleverness and novelty
  – needed to publish academic papers
  – I suspect we need a lot of refinement of ideas (engineering)
Heard in Bergen

• Mix concepts code and “old template” code
• Tool support is essential
  – Debugging
  – Intellisence
• Concepts as predicates?
  – Predicates on function arguments in requires specification?
  – How about exception is concept predicates?
  – How to define equivalence for concept predicates?
• Interface to theorem prover?
• The compiler know a lot about types that we can’t get to
• We need to reduce the amount of template hacking
Heard in Bergen

• The main purpose of “concepts” is to help programmers
  – not to get academic publications!
  – Me, of course

• C++’s image is a huge problem
  – Financial, high-performance, mobile

• Incremental adoption is essential
  – Concept and old-template libraries working together
  – Must be able to start small, e.g. individual class or algorithm
References (links on my home pages)