The Design of C++11

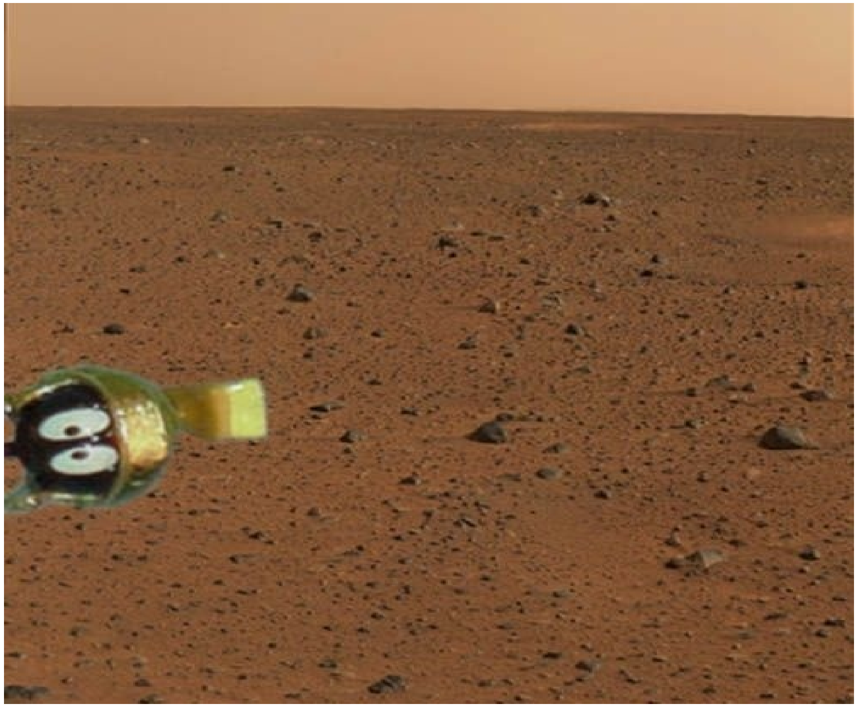
Bjarne Stroustrup
Texas A&M University
http://www.research.att.com/~bs

Overview

- Aims, Ideals, and history
- C++
- Design rules for C++11
  - With examples
- Case study
  - Concurrency

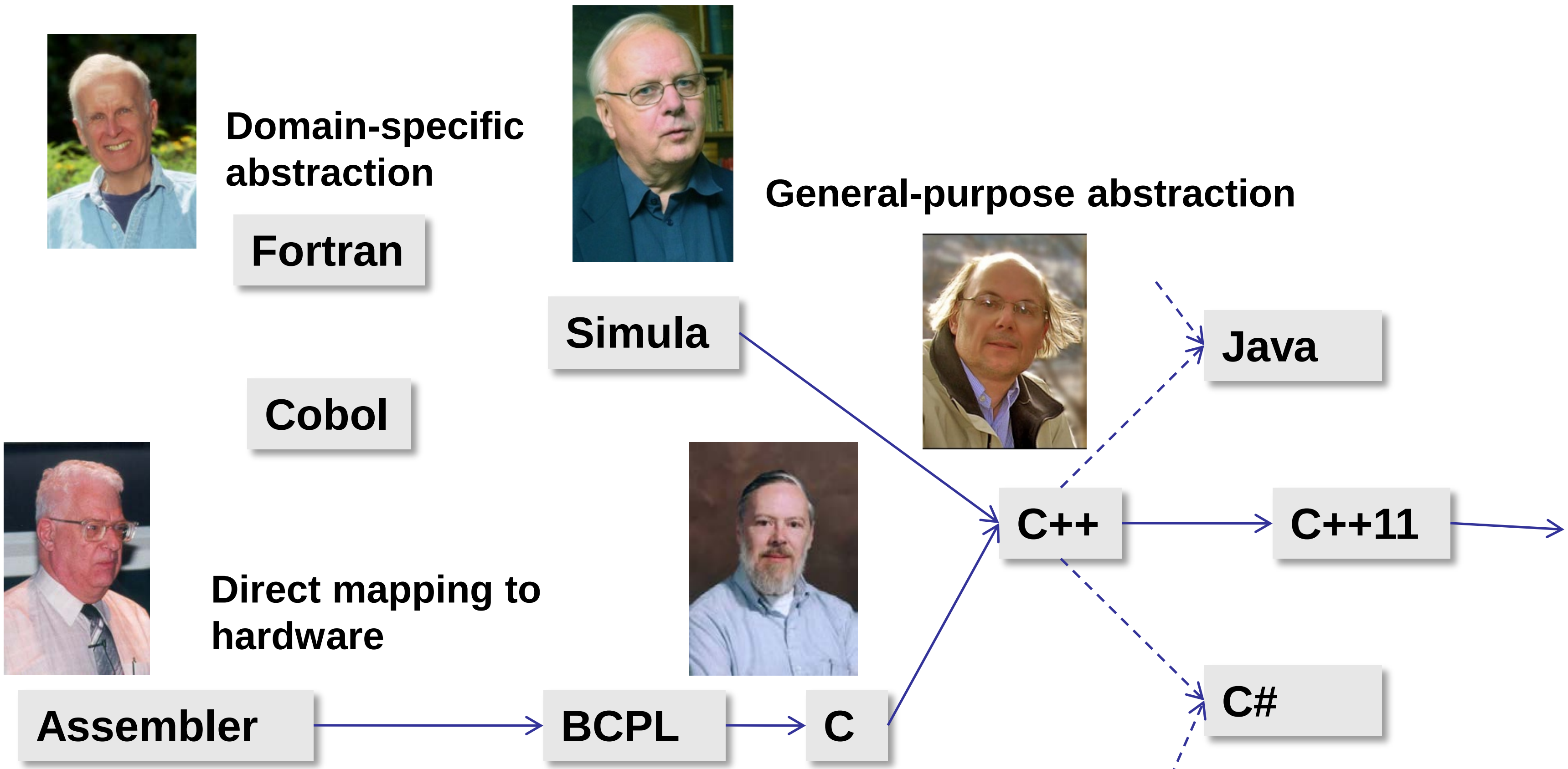
Programming languages

- 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



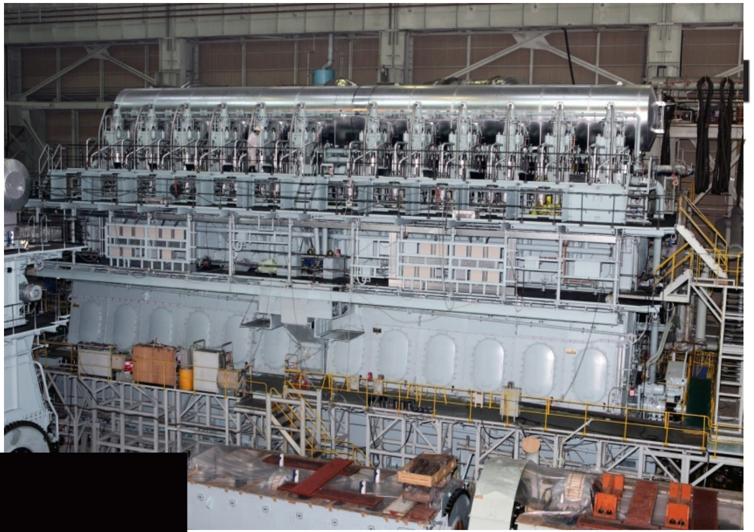
Ideals

- Work at the highest feasible level of abstraction
  - More general, correct, comprehensible, and maintainable code
- Represent
  - concepts directly in code (types, algorithms)
  - 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 with Classes –1980

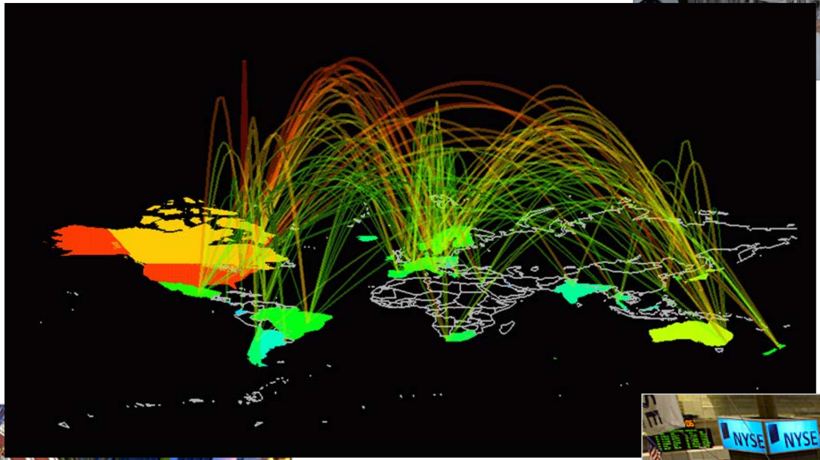
- General abstraction mechanisms to cope with complexity
  - From Simula
- General close-to-hardware machine model for efficiency
  - From C
  - Became C++ in 1984
  - Commercial release 1985
    - Non-commercial source license: $75
  - C++98: ISO standard 1998
  - C++11: 2nd ISO standard 2011

C++ applications



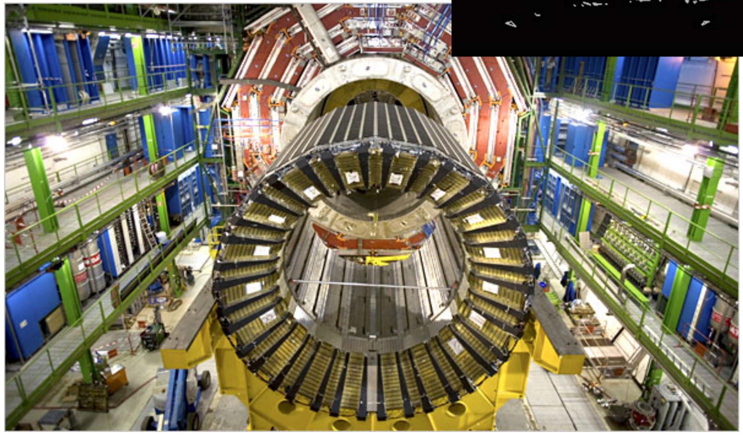
OpenOffice.org

Microsoft .net



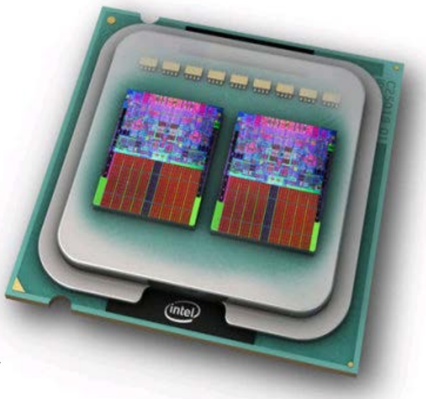
Java

C#



Google

C++ Applications



- www.research.att.com/~bs/applications.html

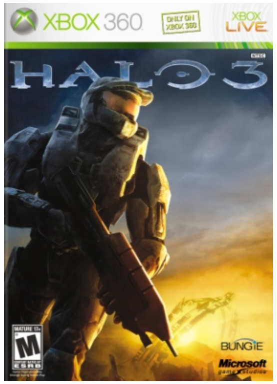
C++ Applications

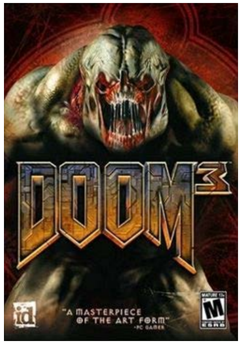
Parasol
Smarter computing.
Texas A&M University

TOMTOM

GARMIN.

MySQL





PayPal

amazon

www.lextrait.com/vincent/implementations.html

aMaDEUS
Your technology partner

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+
    - 200+ members currently
  - 40 to 100 at a meeting
    - 70+ currently
- 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

Design?

- Can a committee design?
  - No (at least not much)
  - Few people consider or care for the whole language
- Is C++11 designed
  - Yes
    - Well, mostly: You can see traces of different personalities in C++11
- Committees
  - Discuss
  - Bring up problems
  - “Polish”
  - Are brakes on innovation

Overall goals for C++11

- 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)

C++11

- C++11 is not science fiction
  – Became an ISO Standard in 2011
  – Every feature is implemented somewhere
    - And shipping, e.g. Microsoft, GCC, Clang, EDG, …
    - E.g. GCC 4.7: Rvalues, Variadic templates, Initializer lists, Static assertions, **auto**, New function declarator syntax, Lambdas, Right angle brackets, Extern templates, Strongly-typed **enum**s, **constexpr**, Delegating constructors (patch), Raw string literals, Defaulted and deleted functions, **noexcept**, Local and unnamed types as template arguments, range-**for**, user-defined literals, …
  – Standard library components are shipping widely
    - E.g. GCC, Microsoft, Boost

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

Maintain stability and compatibility

- “Don’t break my code!”
  - There are billions of lines of code “out there”
  - There are millions of C++ programmers “out there”

- “Absolutely no incompatibilities” leads to ugliness
  - We introduce new keywords as needed: **auto** (recycled), **decltype**, **constexpr**, **thread_local**, **nullptr**
  - Example of incompatibility:

    static_assert(4<=sizeof(int),"error: small ints");

Support both experts and novices

- *Example*: minor syntax cleanup

```
vector<list<int>> v;            // note the "missing space"
```

- *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
```

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)
```

Uniform initialization

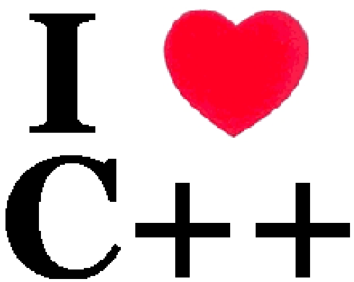
- **{}**-initialization **X{v}** yields the same value of **X** in every context

```
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
```

Uniform initialization

- **{}**-initialization does not narrow

```
int x1 = 7.9;     // x1 becomes 7
int x2 {7.9};     // error: narrowing conversion

Table phone_numbers = {
    { "Donald Duck", 2015551234 },
    { "Mike Doonesbury", 9794566089 },
    { "Kell Dewclaw", 1123581321 }
};
```



Prefer libraries to language extensions

- Libraries deliver more functionality
- Libraries are immediately useful
- *Problem*: Enthusiasts prefer language features
  - see library as 2nd best
- *Example*: New library components
  - **std::thread**, **std::future**, …
    - Threads ABI; not thread built-in type
  - **std::unordered_map**, **std::regex**, …
    - Not built-in associative array

Prefer generality to specialization

- *Example*: Prefer improvements to abstraction mechanisms over separate new features
  - Inherited constructor

```
template<class T> class Vector : std::vector<T> {
    using vector::vector<T>;          // inherit all constructors
    // …
};
```

  - Move semantics supported by rvalue references

```
template<class T>  class vector {
    // …
    void push_back(T&& x);            // move x into vector
                                      // avoid copy if possible
};
```

- *Problem*: people love small isolated features

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*

Move semantics

- Move constructor

```
template<typename T>
class vector {
        // …
        vector(vector&& v)
        {
                elem = v.elem;          // “steal” v’s representation
                sz= v.sz;
                elem = nullptr;         // leave v empty
                sz = 0;
        }
private:
        T* elem;
        int sz;
}
```

Increase type safety

Parasol
Smarter computing.
Texas A&M University

- Approximate the unachievable ideal
  - *Example*: Strongly-typed enumerations

```
enum class Color { red, blue, green };
int x = Color::red;          // error: no Color->int conversion
Color y = 7;                 // error: no int->Color conversion
Color z = red;               // error: red not in scope
Color c = Color::red;        // fine
```

  - *Example*: Support for general resource management
    - **std::unique_ptr** (for ownership)
    - **std::shared_ptr** (for sharing)
    - Garbage collection ABI

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 {              // "literal type" can be used in constant expression
    int x, y;
    constexpr Point(int xx, int yy) : x{xx}, y{yy} { }
};

constexpr Point p1{1,2};        // must be evaluated at compile time: ok
constexpr Point p2{1,abs(x)};  // ok?: is x is a constant expression?
```

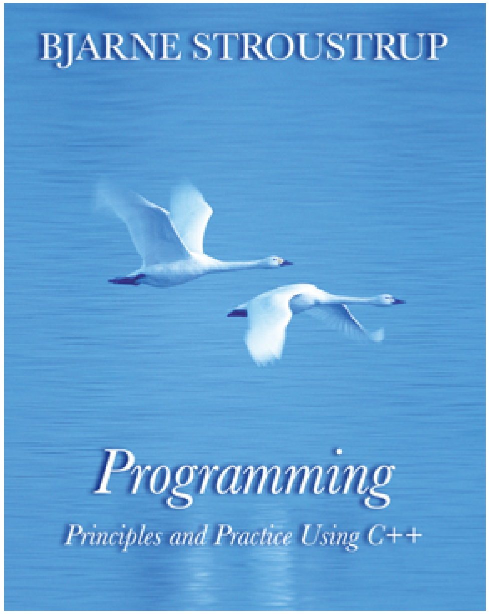
Make only changes that change the way people think

- Think/remember:
  - Object-oriented programming
  - Generic programming
  - Concurrency
  - …
- But, most people prefer to fiddle with details
  - So there are dozens of small improvements
    - All useful somewhere
    - **long long**, **static_assert**, raw literals, **thread_local**, unicode types, …
  - *Example*: A null pointer keyword

```
void f(int);
void f(char*);
f(0);            // call f(int);
f(nullptr);      // call f(char*);
```

Fit into the real world

- *Example*: Existing compilers and tools must evolve
  - Simple complete replacement is impossible
  - Tool chains are huge and expensive
  - There are more tools than you can imagine
  - C++ exists on *many* platforms
    - So the tool chain problems occur N times
      - (for each of M tools)

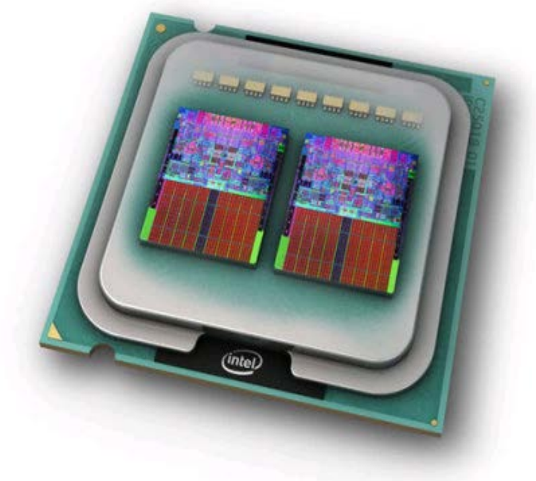


- *Example*: Education
  - Teachers, courses, and textbooks
    - Often mired in 1970s thinking ("C is the perfect language")
    - Often mired in 1980s thinking ("OOP: Rah! Rah!! Rah!!!")
  - "We" haven't completely caught up with C++98!
    - "legacy code breeds more legacy code"

Parasol
Smarter computing.
Texas A&M University

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 **enums**
  - **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**)
  - Intializer-list constructors
  - Emplace operations
  - Scoped allocators
- More algorithms (just a few)
- 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++11

- It *feels* like a new language
  - Compared to C++98
- It's *not* just "object oriented"
  - Many of the key user-defined abstractions are not objects
    - Types
    - Classifications and manipulation of types (types of types)
      - I miss "concepts"
    - Algorithms (generalized versions of computation)
    - Resources and resource lifetimes
- The pieces fit together much better than they used to

C++

Key strength:

Building software infrastructures and resource-constrained applications

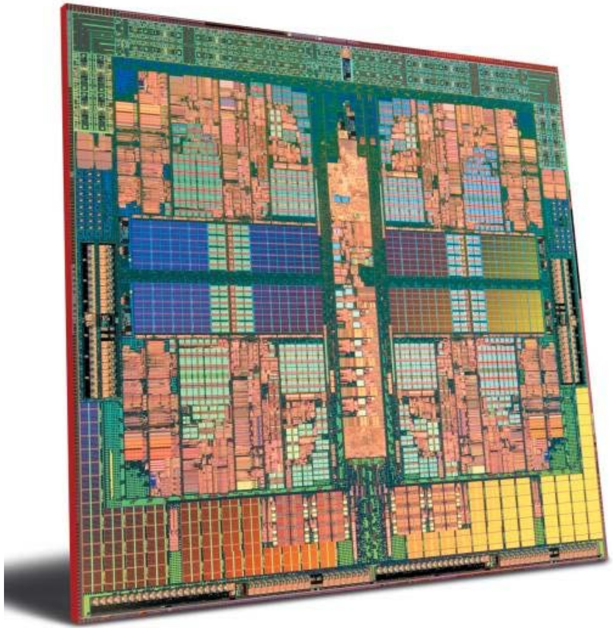
A light-weight abstraction programming language

So, what does "light-weight abstraction" mean?

- The design of programs focused on the design, implementation, and use of abstractions
  - Often abstractions are organized into libraries
    - So this style of development has been called "library-oriented"
- C++ emphasis
  - Flexible static type system
  - Small abstractions
  - Performance (in time and space)
  - Ability to work close to the hardware

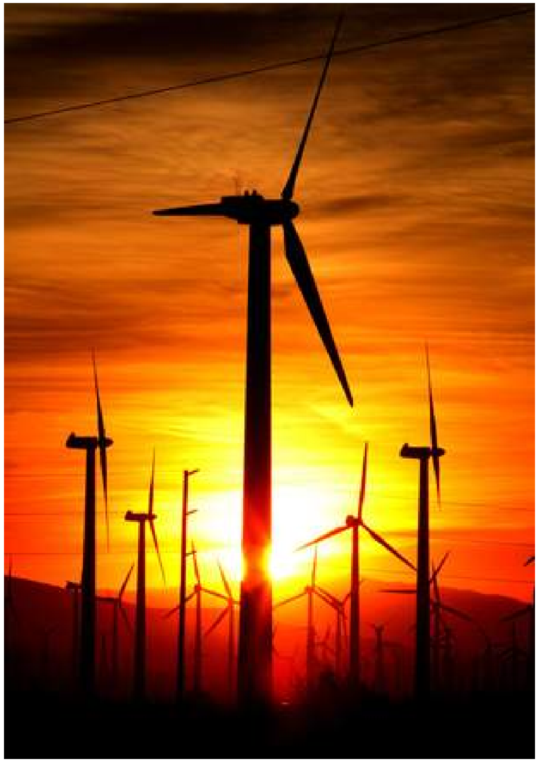
Case study

- Concurrency
  - “driven by necessity”
  - More than ten years of experience



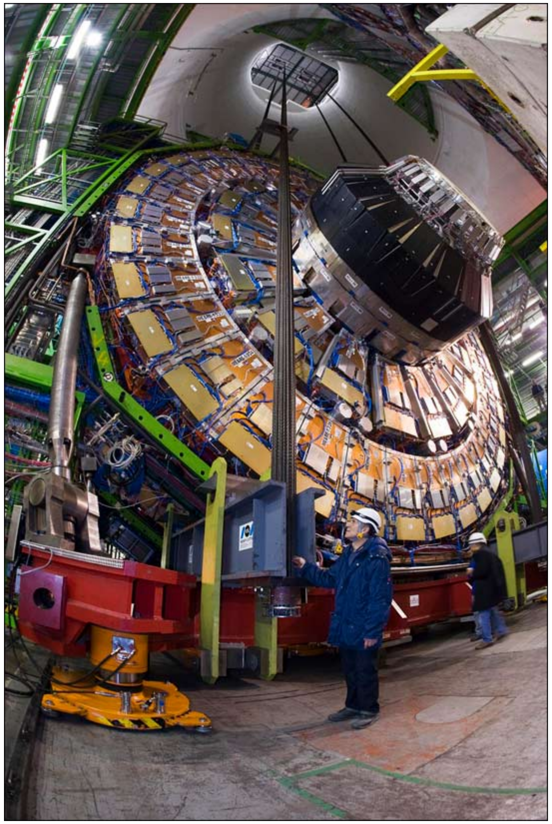
Case study: Concurrency

- 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 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.

```
// 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
};
```

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, …

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 …

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 result (primitive)

```
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 << '\n';
}
```

Thread – pass argument and result

```
double* f(const vector<double>& v);    // read from v return result
double* g(const vector<double>& v);    // read from v return result

void user(const vector<double>& some_vec)        // note: const
{
    double res1, res2;
    thread t1 {[&]{ res1 = f(some_vec); }};    // lambda: leave result in res1
    thread t2 {[&]{ res2 = g(some_vec); }};    // lambda: leave result in res2
    // …
    t1.join();
    t2.join();
    cout << res1 << ' ' << res2 << '\n';
}
```

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 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();
```

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
}
```

Mutex – try_lock_for()

- Don’t wait for too long:

```
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:

```
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

```
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**)

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

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

Potential deadlock

- Unstructured use of multiple locks is hazardous:

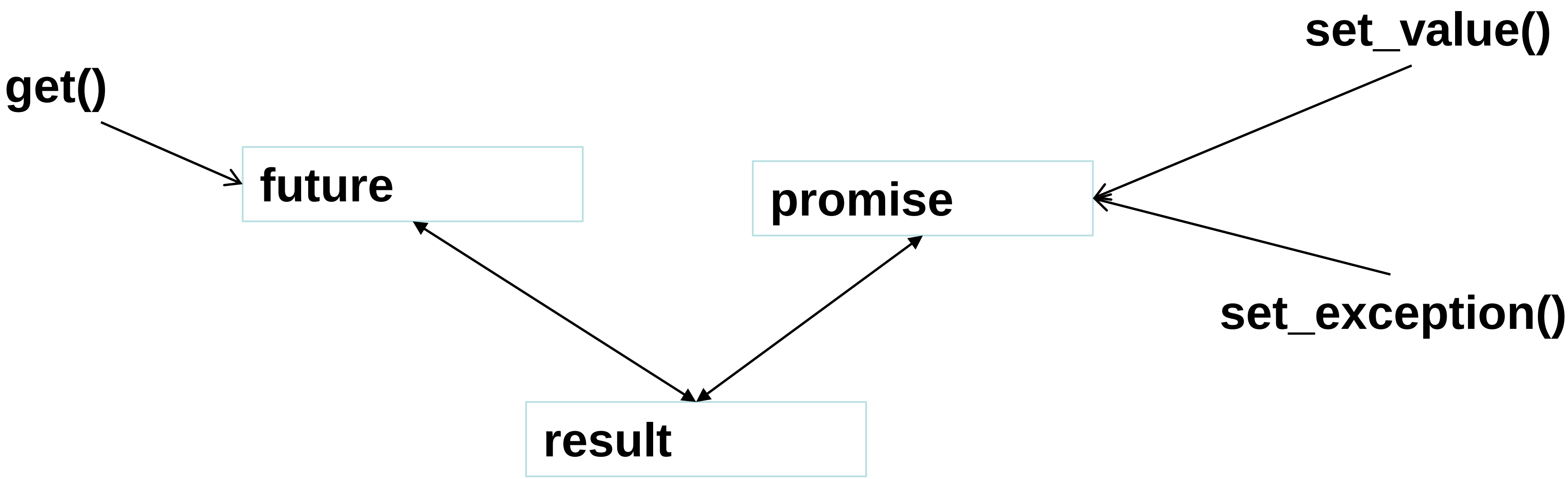

```
mutex m1;
mutex m2;
int sh1; // shared data
int sh2;
// …
void f() {
    // …
    unique_lock<mutex> lck1(m1);
    unique_lock<mutex> lck2(m2);
    // manipulate shared data:
    sh1+=sh2;
}
```

RAII for mutexes: std::lock

- We can safely use several locks

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

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**:

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

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

```
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() – pass argument and return result

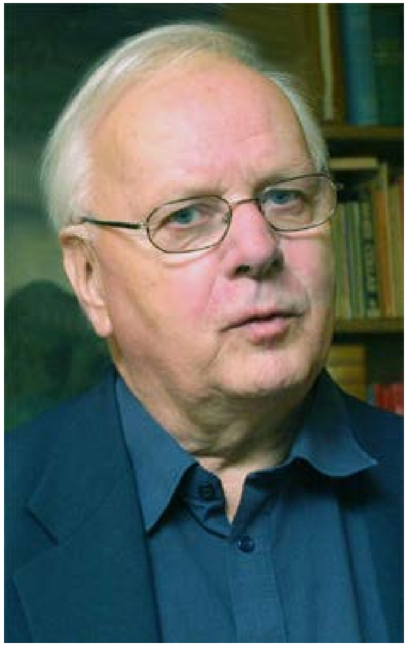
```
double* f(const vector<double>& v);    // read from v return result
double* g(const vector<double>& v);    // read from v return result

void user(const vector<double>& some_vec)        // note: const
{
    auto res1 = async(f,some_vec);
    auto res2 = async(g,some_vec);
    // …
    cout << *res1.get() << ' ' << *res2.get() << '\n';  // futures
}
```

- Much more elegant than the explicit thread version
  - And most often faster

async()

- Simple launcher using the variadic template interface

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

void comp(vector<double>& v)          // spawn many
{
    auto b = &v[0];
    auto sz = v.size();

    auto f0 = async(Accum, b,           b+sz/4,     0.0);
    auto f1 = async(Accum, b+sz/4,     b+sz/2,     0.0);
    auto f2 = async(Accum, b+sz/2,     b+sz*3/4, 0.0);
    auto f3 = async(Accum, b+sz*3/4, b+sz,        0.0);
    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

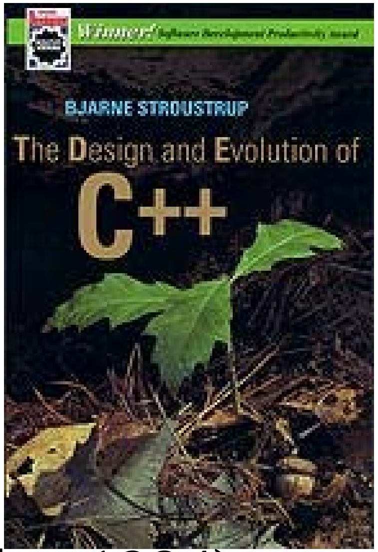


Thanks!

Parasol
Smarter computing.
Texas A&M University

- 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

More information



- My home pages
  - C++11 FAQ
  - Papers, FAQs, libraries, applications, compilers, …
    - Search for “Bjarne” or “Stroustrup”
    - “Software Development for Infrastructure” 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”