

# The Design of C++0x

Bjarne Stroustrup

Texas A&M University

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



# Overview

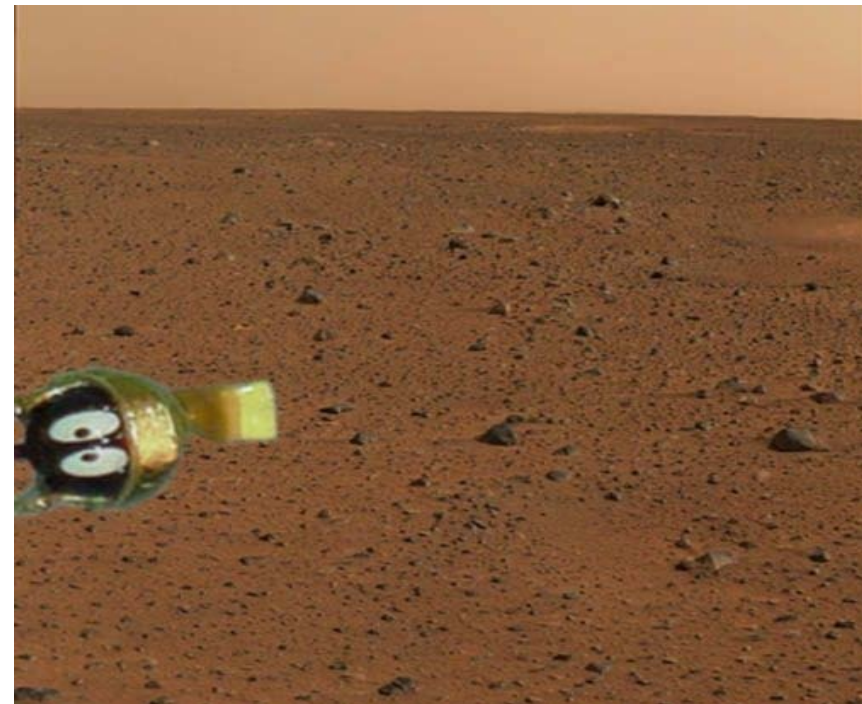
- Aims, ideals, and history
- C++
- Design rules for C++0x
  - With tiny examples



# Programming languages

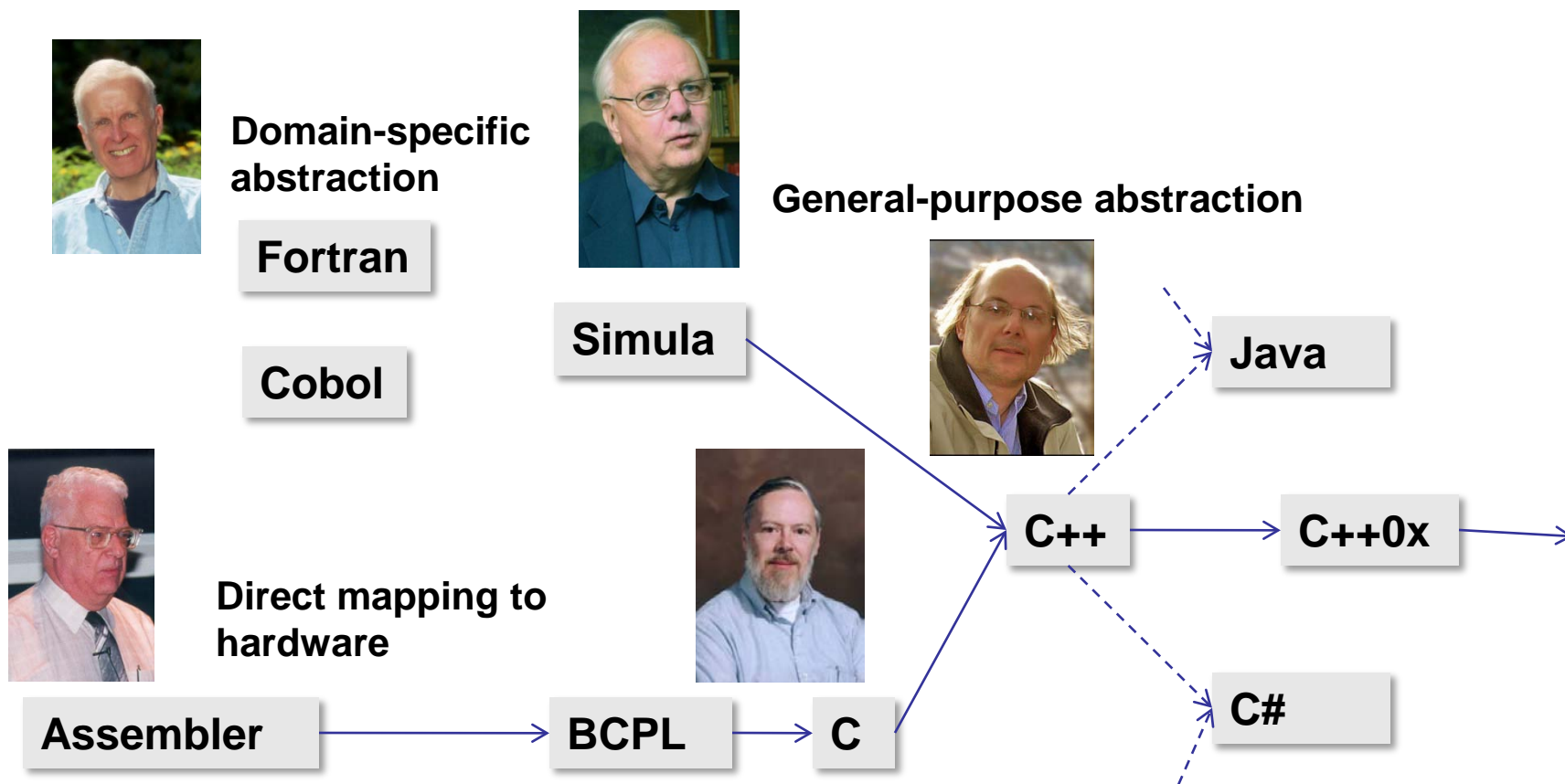
A programming language exists to help people express ideas

- 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

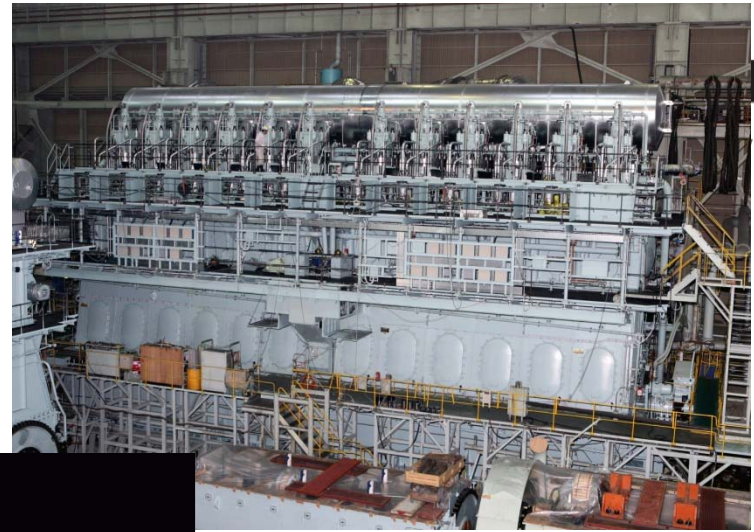
- Work at the highest feasible level of abstraction
  - More general, correct, comprehensible, and maintainable code
- Represent
  - concepts directly in code
  - independent concepts independently in code
- Represent relationships among concepts directly
  - For example
    - Hierarchical relationships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

# C with Classes –1980

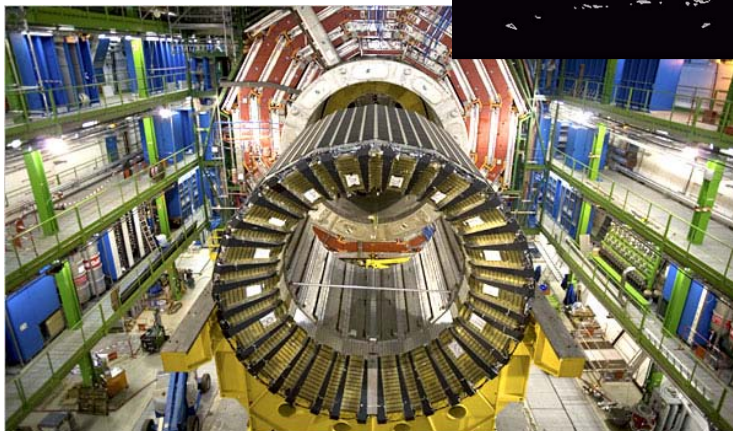
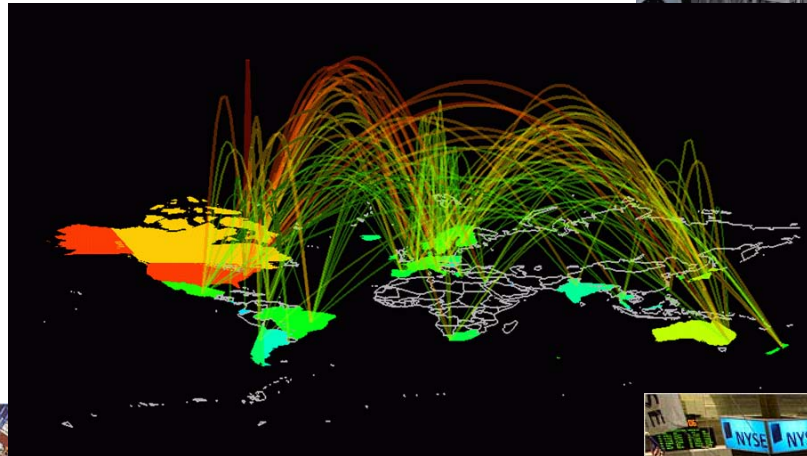
- General abstraction mechanisms to cope with complexity
  - From Simula
- General close-to-hardware machine model for efficiency
  - From C
- Became C++ in 1984
  - Commercial release 1985
    - Non-commercial source license: \$75
  - ISO standard 1998
  - C++0x Final Draft Standard 2010
    - 2<sup>nd</sup> ISO standard 200x ('x' is hex ☹)



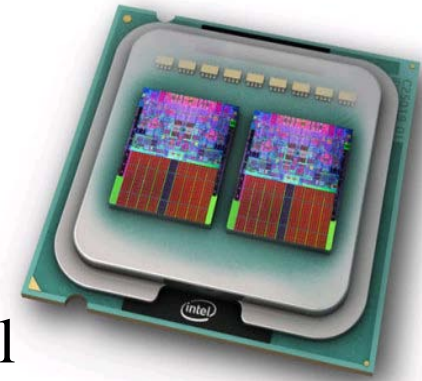
# C++ applications



Microsoft  
.net



# C++ Applications



- [www.research.att.com/~bs/applications.html](http://www.research.att.com/~bs/applications.html)

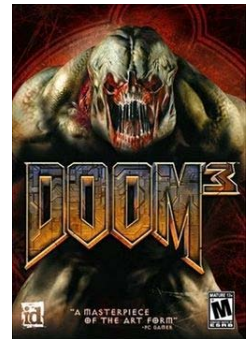


Stroustrup - Rapperswil - 2010

# C++ Applications



S117E08041



[www.lextrait.com/vincent/implementations.html](http://www.lextrait.com/vincent/implementations.html)



Stroustrup - Rapperswil - 2010

# 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
    - ~60 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



# 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)

# C++0x

- ‘x’ may be hex, but C++0x is not science fiction
  - Every feature is implemented somewhere
    - E.g. GCC 4.5: 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
  - 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

# 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");`
- “Absolutely no incompatibilities” leads to absurdities  
`_Bool` *// C99 boolean type*  
`typedef _Bool bool;` *// C99 standard library typedef*

# 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
```

# 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
```

# Prefer libraries to language extensions

- Libraries deliver more functionality
- Libraries are immediately useful
- *Problem:* Enthusiasts prefer language features
  - see library as 2<sup>nd</sup> best
- *Example:* New library components
  - `std::thread`, `std::future`, ...
  - `std::unordered_map`, `std::regex`, ...
    - Threads ABI; not thread built-in type, not built-in associative array, ...
- *Example:* Mixed language/library extension
  - The new **for** works for every type with `std::begin()` and `std::end()`
  - The new initializer lists are based on `std::initializer_list<T>`

```
vector<string> v = { "Nygaard ", "Ritchie" };
for (auto& x : {y,z,ae,ao,aa}) cout << x << '\n';
```



# Prefer generality to specialization

- *Example:* Improvements to abstraction mechanisms

- 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*

# Increase type safety

- 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 {`

`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?`

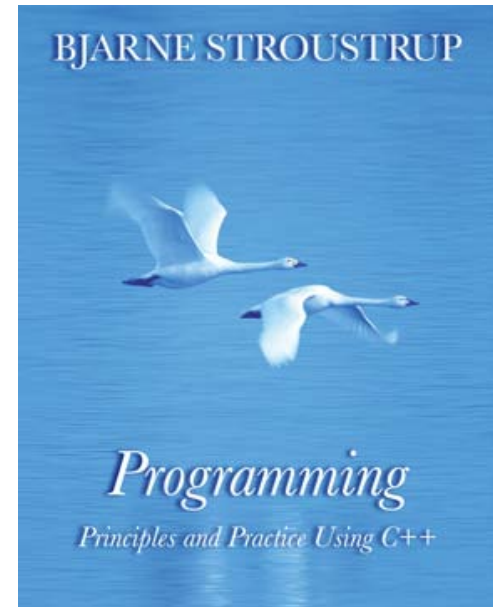
# Make only changes that change the way people think

- 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*);
```

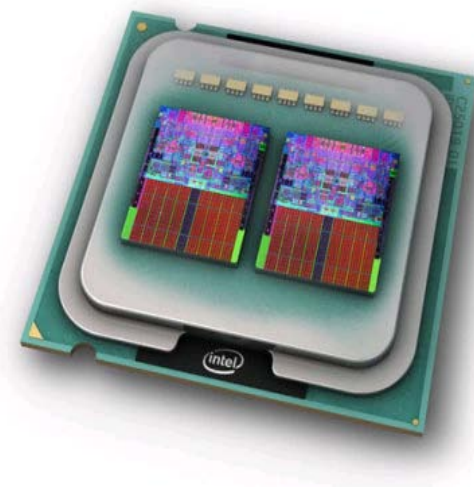
# Fit into the real world

- *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”



# 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.
  - ...



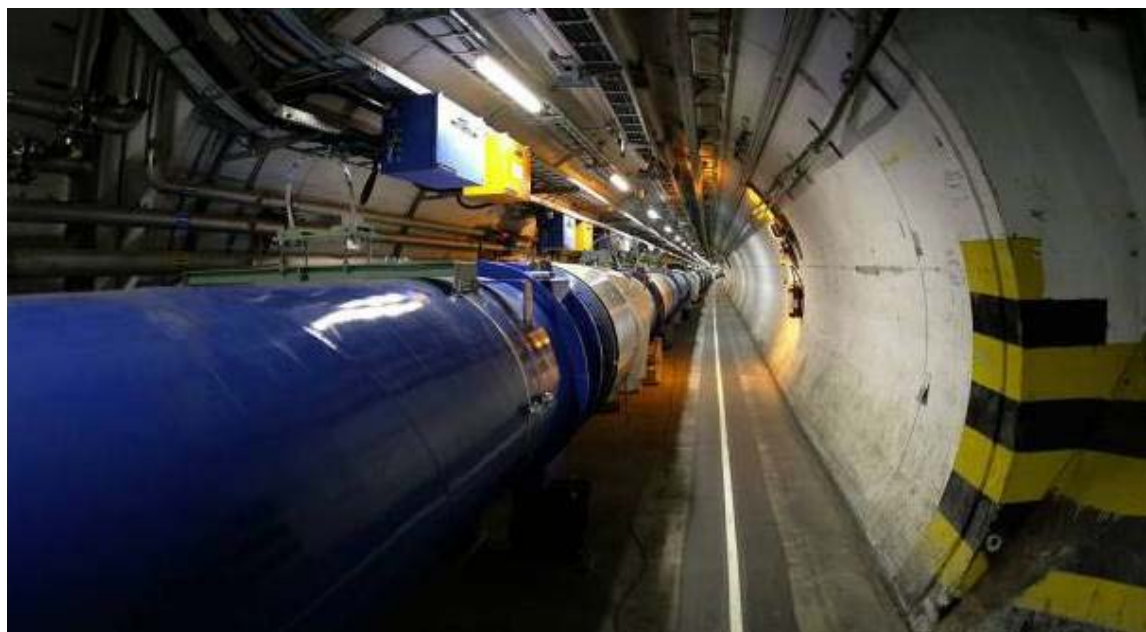
# Standard Library Improvements

- New containers
  - Hash Tables (**unordered\_map**, etc.)
  - Singly-linked list (**forward\_list**)
  - Fixed-sized array (**array**)
- Container improvements
  - Move semantics (e.g. **push\_back**)
  - \_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**



**Too big!**

**It's C!**

**Embedded systems  
programming language**

**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**

# 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





# Thanks!

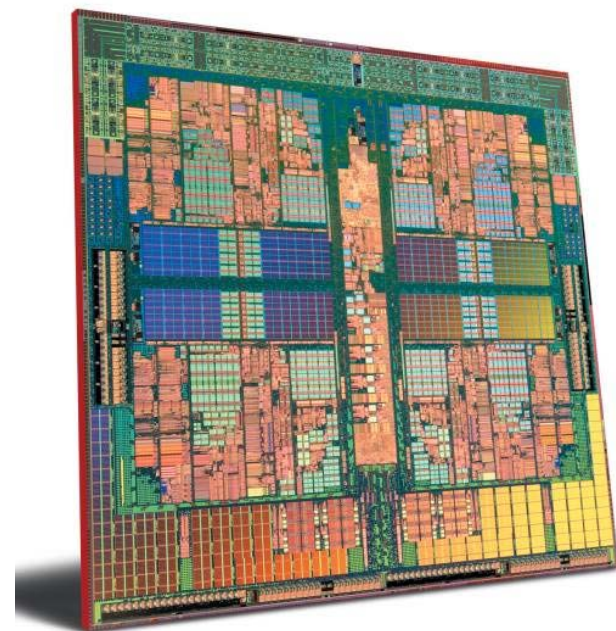
- C and Simula
  - Brian Kernighan
  - Doug McIlroy
  - Kristen Nygaard
  - Dennis Ritchie
  - ...
- ISO C++ standards committee
  - Steve Clamage
  - Francis Glassborow
  - Andrew Koenig
  - Tom Plum
  - Herb Sutter
  - ...
- C++ compiler, tools, and library builders
  - Beman Dawes
  - David Vandevoorde
  - ...
- Application builders
  - Stroustrup - Rapperswil - 2010





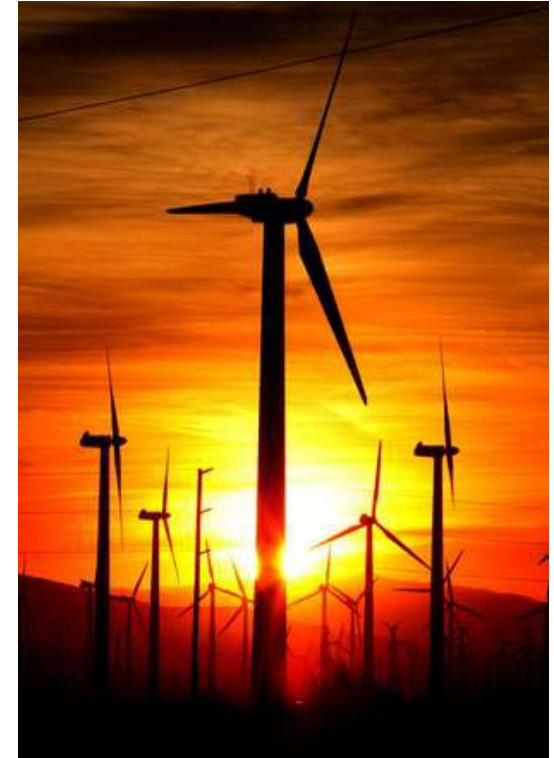
# Case study

- 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>&);
```

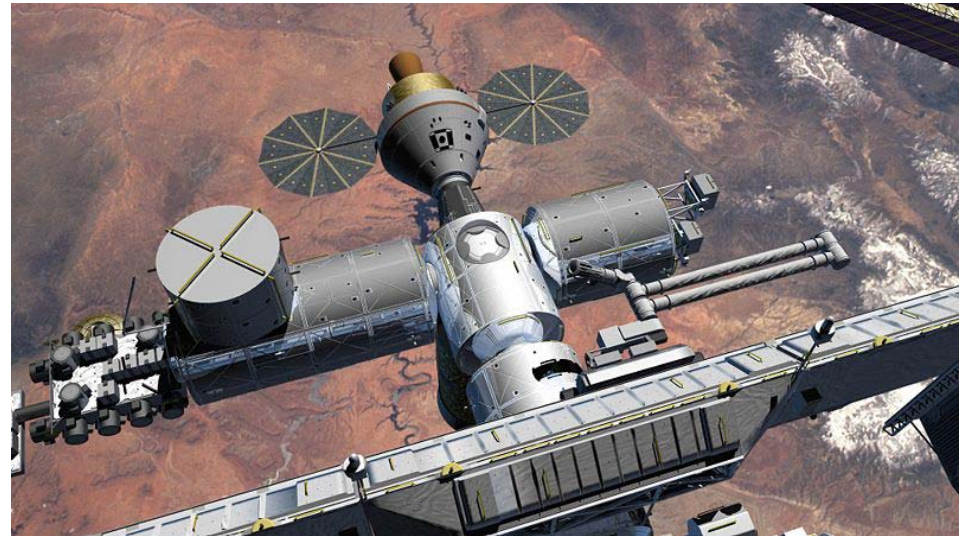
```
struct F {  
    vector<double>& v;  
    F(vector<double>& vv) :v{vv} { }  
    void operator();  
};
```

```
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();  
}
```

# 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  
}
```

# 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
```

# 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;  
}
```

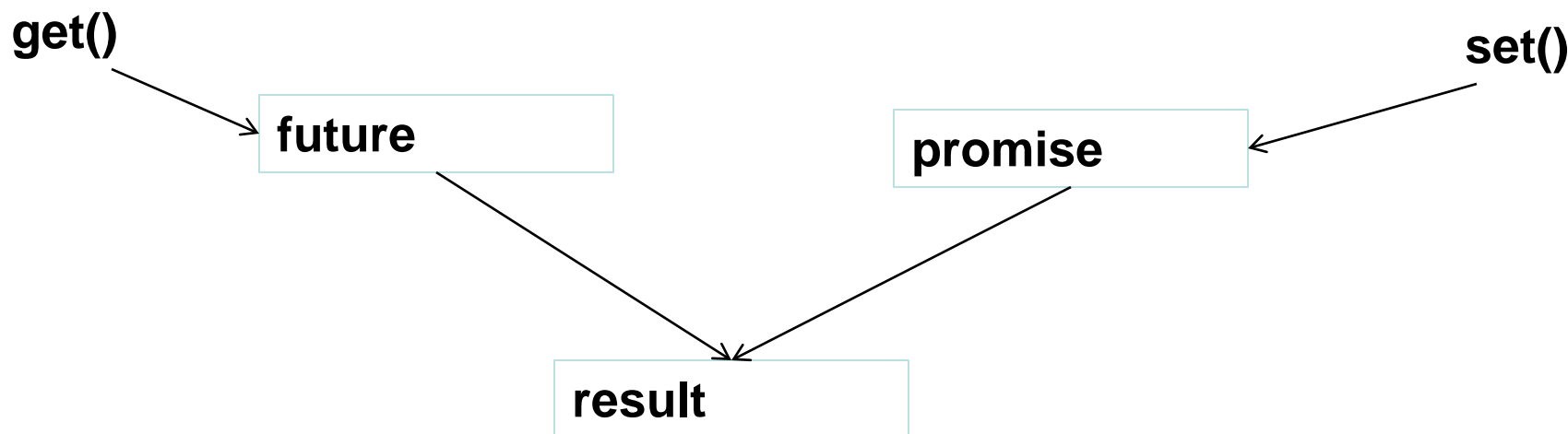


# RAII for mutexes: `std::lock`

- We can safely use several locks

```
void f() {  
    // ...  
    std::unique_lock<mtx> lck1(m1, std::defer_lock); // don't yet acquire  
    std::unique_lock<mtx> lck2(m2, std::defer_lock);  
    std::unique_lock<mtx> lck3(m3, std::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 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/](http://www.softwarepreservation.org/projects/c_plus_plus/)
      - Search for “C++ Historical Sources Archive”

