The Design of C++0x

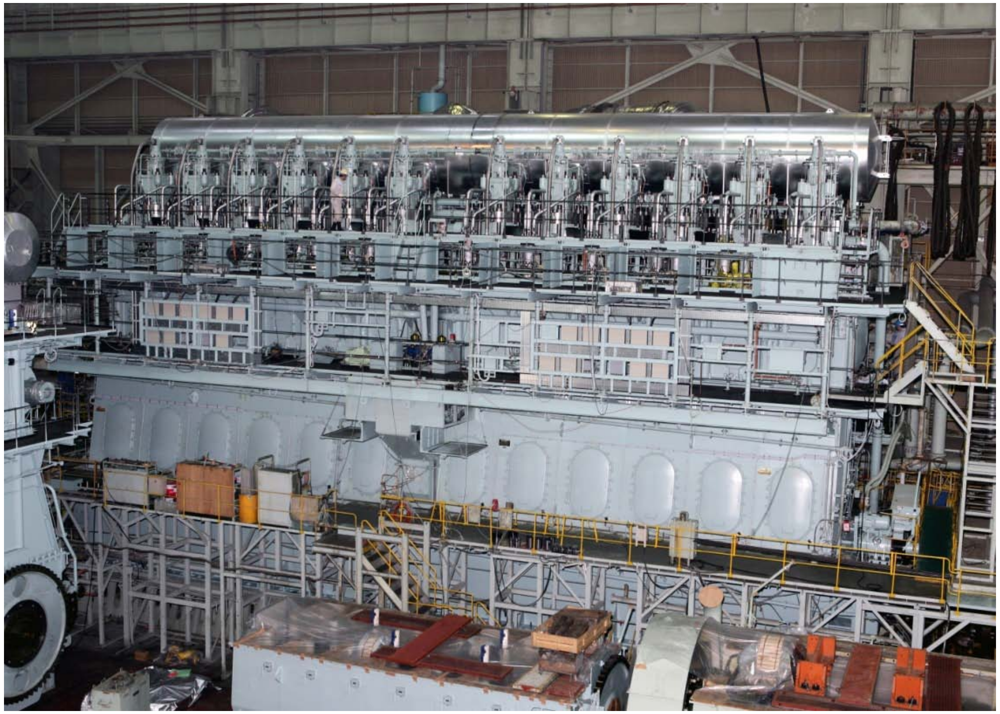
Bjarne Stroustrup
Texas A&M University
http://www.research.att.com/~bs

Caveat

- There are people who want “just the facts, just the technical details”
  - I’m writing the C++0x FAQ for you
  - If you really want all the details, read the draft standard (hard)
  - Technical details in isolation are sterile
- There are people who want “grand theory, fundamental principles, and no distracting details”
  - I don’t do that
  - Theory in isolation is sterile
- This talk
  - Gives a bit of background (history) and some simple design principles illustrated by the simplest code examples I can find

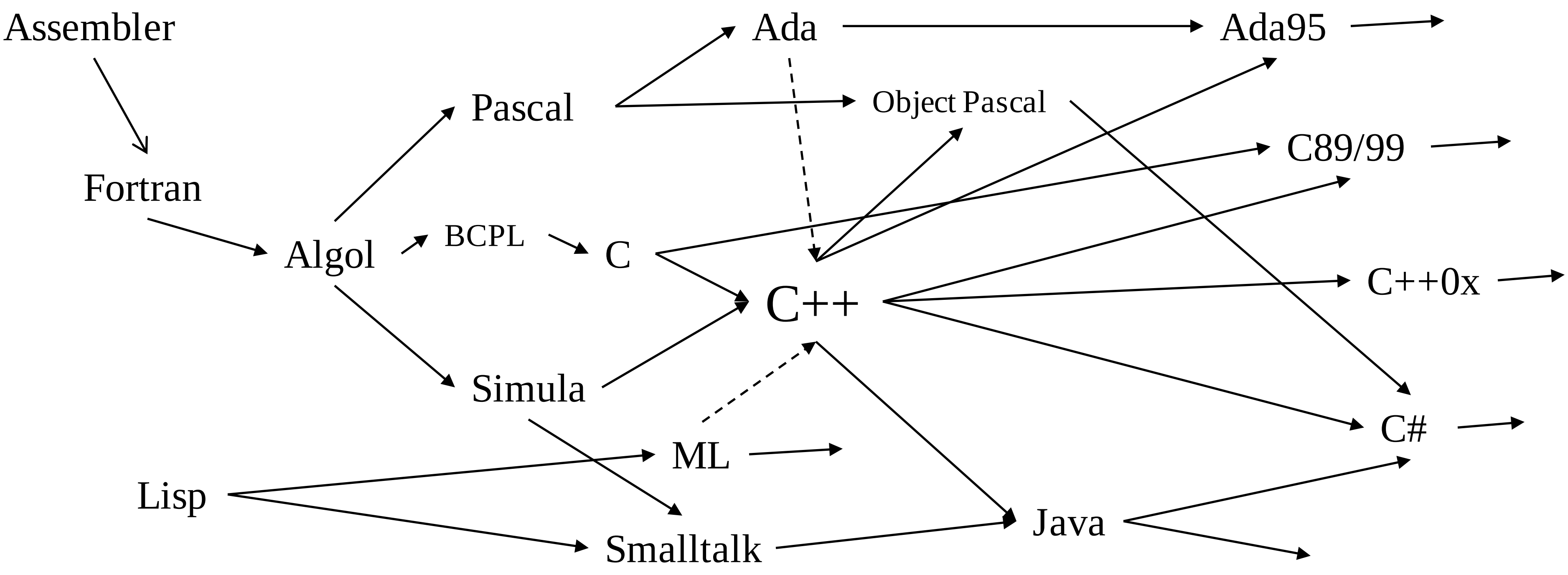
Overview

- Aims, Ideals, and history
- C++
- Design rules for C++0x
  - With examples
- Case studies
  - Initialization
  - Concurrency



8000+ Programming Languages

- C++'s family tree (part of)



- And this is a gross oversimplification!

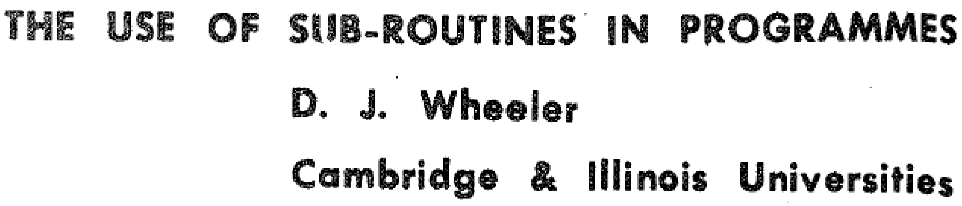
Programming languages

- 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 continues

Assembler –1951

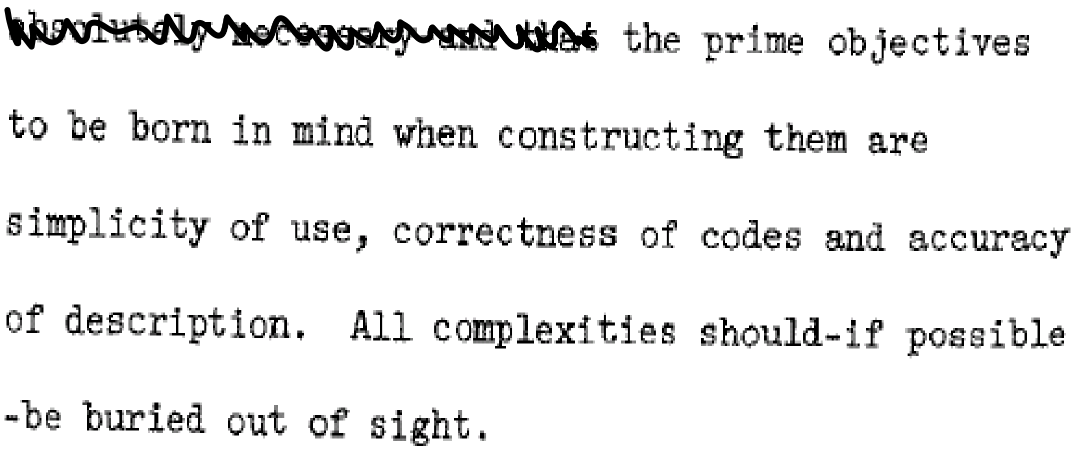
- Machine code to assembler and libraries
  - Abstraction
  - Efficiency
  - Testing
  - documentation



THE USE OF SUB-ROUTINES IN PROGRAMMES

D. J. Wheeler

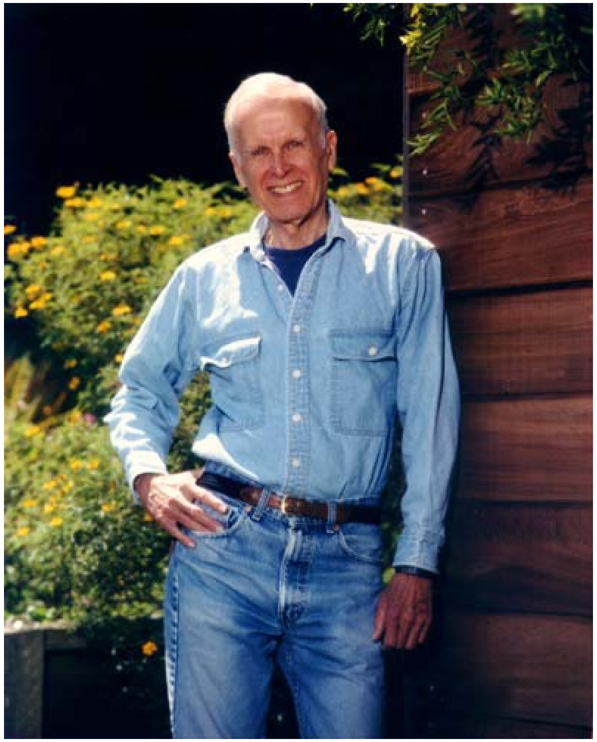
Cambridge & Illinois Universities



~~absolutely necessary and that~~ the prime objectives to be born in mind when constructing them are simplicity of use, correctness of codes and accuracy of description. All complexities should-if possible -be buried out of sight.

Fortran –1956

- A notation fit for humans
  - For a specific application domain
    - **A(I) = B(I)+C*D(I)**
  - Efficiency a premium
  - Portability



Simula –1967

- Organize code to “model the real world”
  - Object-oriented design
- Let the users define their own types (classes)
  - In general: concepts/ideas map to classes
  - “Data abstraction”
- Organize classes into hierarchies
  - Object-oriented programming

C –1974

- An simple and general notation for systems programming
  - Somewhat portable
  - Direct mapping of objects and basic operations to machine
    - Performance becomes somewhat portable

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
  - ISO standard 1998
  - 2nd ISO standard 200x (‘x’ is hex ☹)

ISO Standard C++

- C++ is a general-purpose programming language with a bias towards systems programming that
  - is a better C
  - supports data abstraction
  - supports object-oriented programming
  - supports generic programming

From day 1 (1980)

From mid-1983

From about 1990

- The most effective styles use a combination of techniques

Focus of C++0x work

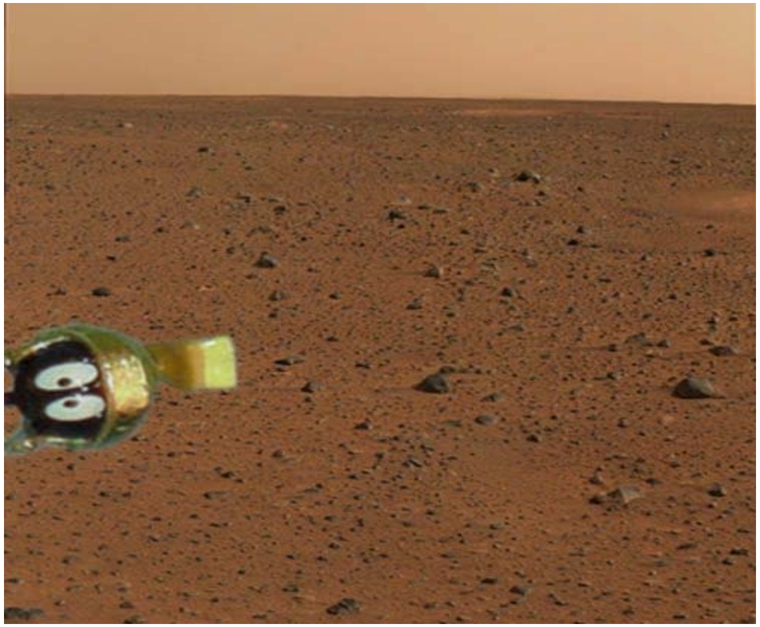
What’s distinctive about C++?

- Stability
  - Essential for real-world software
  - 1985-2008
  - 1978-2008 (C and C with Classes)
- Non-proprietary
  - Yet almost universally supported
  - ISO standard from 1998
- Direct interface to other languages
  - Notably C, assembler, Fortran
- Abstraction + machine model
  - Zero overhead principle
    - For basic operations and abstraction mechanisms
  - User-defined types receive the same support as built-in types
  - Standard library written in the language itself
    - And most non-standard libraries

C++ is everywhere

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

- Telecommunications
- Google
- Microsoft applications and GUIs
- Linux tools and GUIs
- Games
- PhotoShop
- Finance
- …



- Mars Rovers
- Marine diesel engines
- Cell phones
- Human genome project
- Micro electronics design and manufacturing
- …

C++ ISO Standardization

- Slow, bureaucratic, democratic, formal process
- 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

Design?

- Can a committee design?
  - No (at least not much)
  - Few people consider or care for the whole language
- Is C++0x designed
  - Yes
    - Well, mostly
    - You can see traces of different personalities in C++0x
- Committees
  - Discuss
  - Bring up problems
  - “Polish”

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++0x

- It *feels* like a new language
    - Compared to C++98
- How can I categorize/characterize it?
- 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 (language features) fit together much better than they used to

C++

A language for 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
  - Performance (in time and space)
  - Small abstractions

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)

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: **concept**, **auto** (recycled), **decltype**, **constexpr**, **thread_local**, **nullptr**, **axiom**
  - 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>> vl;          // 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
```

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::unique_future**, …
    - Threads ABI; not thread built-in type
  - **std::unordered_map**, **std::regex**, …
    - 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*: 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(const T&& x);    // move x into vector
                                        // avoid copy if possible
};
```

- *Problem*: people love small isolated features

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; }

struct Point {
   int x, y;
   constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};

constexpr Point p1(1,2);          // ok
constexpr Point p2(1,abs(x));  // error unless x is a constant expression
```

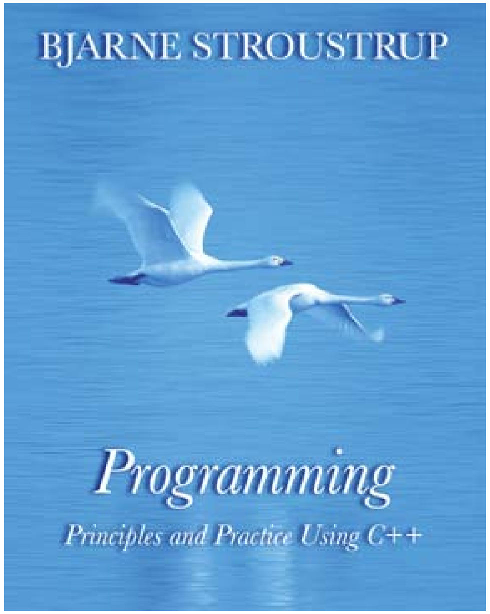
Make only changes that change the way people think

- 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*);
```

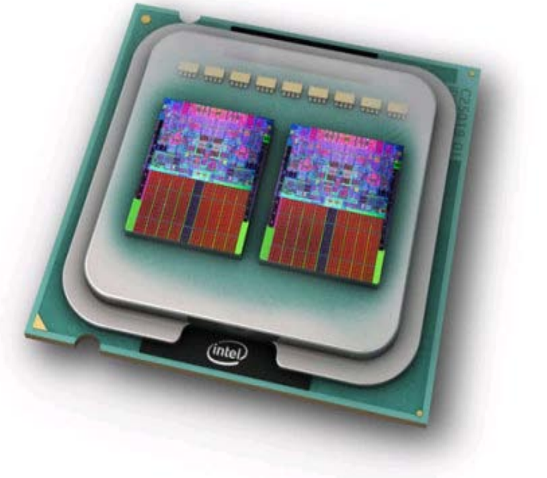
Fit into the real world

- *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)
    - or 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**)
  - Atomic ABI
  - Thread-local storage (**thread_local**)
  - Asynchronous message buffer (**std::future**)



- Support for generic programming
  - (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.
  - …

Case studies

- Concurrency
  - “driven by necessity”
- Initialization
  - “language maintenance”

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

- 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.

// thread 1:	*// thread 2:*
char c;	**char b;**
c = 1;	**b = 1;**
int x = c;	**int y = b;**

x==1 and **y==0** as anyone would expect

(but don't try that for two bitfields of the same word)

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 …
```

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
```

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();
```

Mutual exclusion: std::mutex

- Not all **mutex** uses are simple:

```
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
}
```

RAII for mutexes: std::lock

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

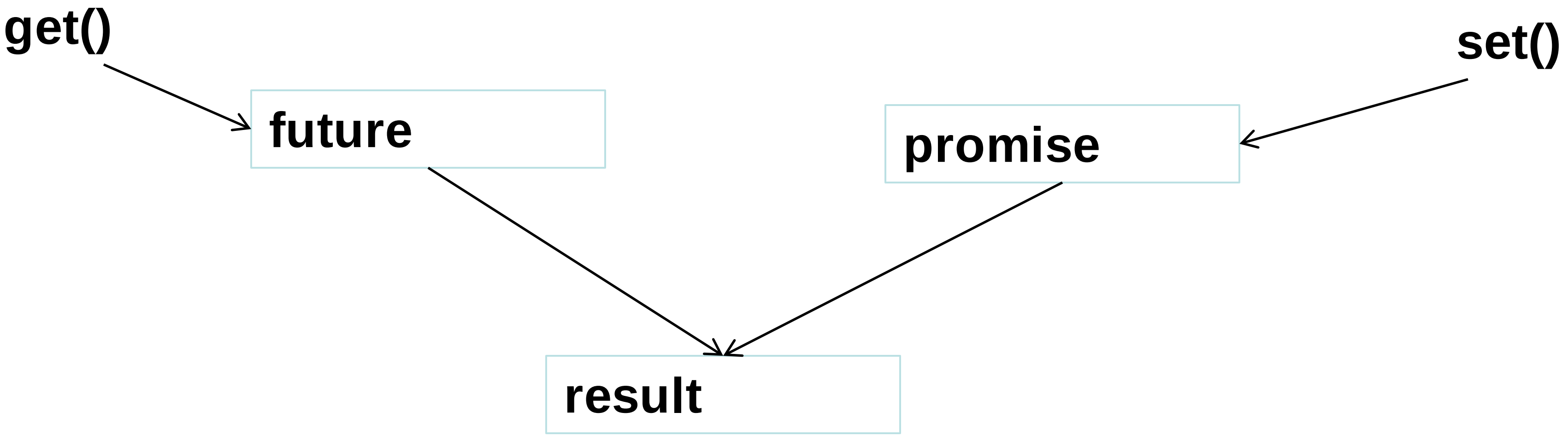
```
std::mutex m;
int sh; // shared data

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

RAII for mutexes: std::lock

- We can safely use several locks

```
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
}
```

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

- Put to a promise:

```
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 (warning: only approved in principle)

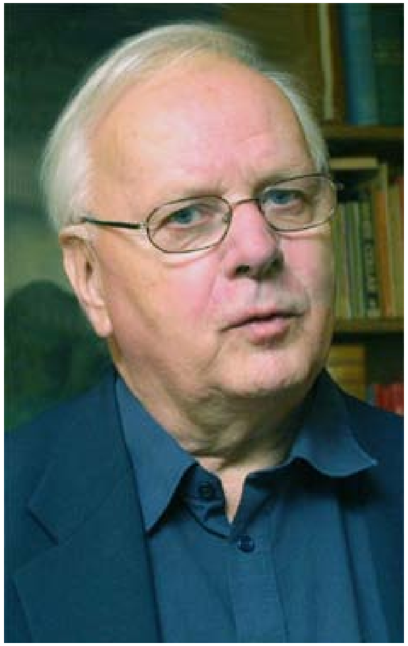
```
template<class T, class V> struct Accum {
    // accumulator function object
};

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();
}
```

Future

- Lots of use
  - C++98, C++0x, C++1x, …
- Is there a future for “the C++ model” beyond C++?
  - Direct map to hardware
  - Zero-overhead abstraction
  - Minimal run-time environment
  - Destructor-based resource management
  - Heavy use of stack
- Challenges
  - Small language (or at least much, much smaller)
  - Complete and enforced type safety
  - Concurrency

yes

I think it can be done

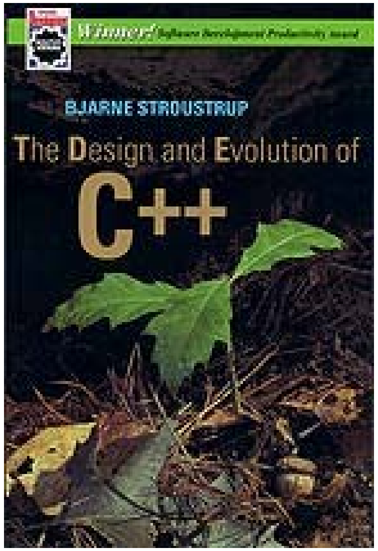


Thanks!

- 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

More information

- My HOPL-II and HOPL-III papers
- The Design and Evolution of C++ (Addison Wesley 1994)
- My home pages
  - Papers, FAQs, libraries, applications, compilers, …
    - Search for “Bjarne” or “Stroustrup”
  - C++0x FAQ
- 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”



C++0x examples

```
// bind a template argument (Currying):
template<class T> using  Vec = std::vector<T,My_alloc<T>>; // an alias
Vec<double> v = { 1, 2.2, 3, 9 };     // Note: general and uniform initialization

sort(v);  // simplicity is the ultimate sophistication (and no spurious overheads)
sort({"Nygaard", "Ritchie", "Richards"}); // error: can sort a constant

for (auto x : v ) cout << x <<'\n';   // simple traversal


// run in parallel:
auto x = asynch([&v]() { return accumulate(v.begin(), v.end(), 0.0); }); // a lambda
// …
double d = x.get();            // if necessary, wait for result
```

Parasol
Smarter computing.
Texas A&M University

C++0x examples

```
struct F {            // function object
    F(const string&);
    double operator()(double, int); // application (function call) operator
    // …
};

auto f = bind(F("Hello"), _1, 42); // _1 is a placeholder
double dd = f(1.23);               // F(“hello”)(1.23,42);
```

Problem #1: irregularity

- There are four notations and none can be used everywhere

```
int a = 2;                  // “assignment style”
int[] aa = { 2, 3 };        // assignment style with list
complex z(1,2);             // “functional style” initialization
x = Ptr(y);                 // “functional style” for conversion/cast/construction
```

- Sometimes, the syntax is inconsistent/confusing

```
int a(1);        // variable definition
int b();         // function declaration
int b(foo);      // variable definition or function declaration
```

- We can’t use initializer lists except in a few cases

```
string a[] = { "foo", " bar" };              // ok: initialize array variable
vector<string> v = { "foo", " bar" };        // error: initialize vector variable
void f(string a[]);
f( { "foo", " bar" } );                      // error: initializer array argument
```

Is irregularity a real problem?

- Yes, a major source of confusion and bugs
- Can it be solved by restriction?
  - No existing syntax can be used in all cases

```
int a [] = { 1,2,3 };                     // can't use () here
complex<double> z(1,2);                   // can't use { } here
struct S { double x,y; } s = {1,2};       // can't use ( ) here
int* p = new int(4);                      // can't use { } or = here
```

  - No existing syntax has the same semantics in all cases

```
typedef char* Pchar;
Pchar p(7);                 // error (good!)
Pchar p = Pchar(7);         // "legal" (ouch!)
```

- Principle violated:
  - Uniform support for types (user-defined and built-in)

Problem #2: list workarounds

- Initialize a vector (using push_back)
  - Clumsy and indirect

```
template<class T> class vector {
    // …
    void push_back(const T&) { /* … */ }
    // …
};

vector<double> v;
v.push_back(1.2);
v.push_back(2.3);
v.push_back(3.4);
```

- Principle violated:
  - Support fundamental notions directly (“state intent”)

Problem #2: list workarounds

- Initialize vector (using general iterator constructor)
  - Awkward, error-prone, and indirect
  - Spurious use of (unsafe) array

```
template<class T> class vector {
    // …
    template <class Iter>
        vector(Iter first, Iter last) { /* … */ }
    // …
};

int a[ ] = { 1.2, 2.3, 3.4 };                        // bug
vector<double> v(a, a+sizeof(a)/sizeof(int));    // hazard
```

- Principle violated:
  - Support user-defined and built-in types equally well

C++0x: initializer lists

- An initializer-list constructor
  - defines the meaning of an initializer list for a type

```
template<class T> class vector {
        // …
        vector(std::initializer_list<T>);    // initializer list constructor
        // …
};

vector<double> v = { 1, 2, 3.4 };

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

C++0x: initializer lists

- Not just for templates and constructors
  - but **std::initializer list** is simple – does just one thing well

```
void f(int, std::initializer_list<int>, int);

f(1, {2,3,4}, 5);
f(42, {1,a,3,b,c,d,x+y,0,g(x+a),0,0,3}, 1066);
```

Uniform initialization syntax

- Every form of initialization can accept the { … } syntax

```
X x1 = X{1,2};
X x2 = {1,2};       // the = is optional
X x3{1,2};
X* p2 = new X{1,2};

struct D : X {
   D(int x, int y) :X{x,y} { /* … */ };
};

struct S {
   int a[3];
   S(int x, int y, int z) :a{x,y,z} { /* … */ };   // solution to old problem
};
```

Uniform initialization semantics

- **X { a }** constructs the same value in every context
  - **{ }** initialization gives the same result in all places where it is legal

    X x{a};

    X* p = new X{a};

    z = X{a}; *// use as cast*

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

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

    …

- **X { … }** is always an initialization
  - **X var{}** *// no operand; default initialization*
    - Not a function definition like **X var();**
  - **X var{a}** *// one operand*
    - Never a function definition like **X var(a);** (if **a** is a type name)

Initialization problem #3: narrowing

- C++98 implicitly truncates

```
int x = 7.3;                        // Ouch!
char c = 2001;                      // Ouch!
int a[] = { 1,2,3.4,5,6 };          // Ouch!

void f1(int);       f1(7.3);                 // Ouch!
void f2(char);      f2(2001);                // Ouch!
void f3(int[]);     f3({ 1,2,3.4,5,6 });     // oh! Another problem
```

- A leftover from before C had casts!
- Principle violated:
  - Type safety
- Solution:
  - C++0x { } initialization doesn’t narrow.
    - all examples above are caught

Uniform Initialization

- Example

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

- What is **Table**?
  - a **map**? An array of **struct**s? A **vector** of **pair**s? My own **class** with a constructor? A **struct** needing aggregate initialization? Something else?
  - We don’t care as long as it can be constructed using a C-style string and an integer.
  - Those numbers cannot get truncated