Homework

• Read my History Of Programming Languages conference papers

• Write a short (minimum 4 pages) essay due next Monday (8 am):
  – Assume you had a time machine and you could take just one trip back in time to help develop C++, to when would you go? And what advise would you give? Remember, you need to back your suggestion with technical arguments that might convince someone at the time (e.g. me or Dennis Ritchie). Beware: you know the present, but you really don’t know the distant past and people then couldn’t imagine the current future.
Supporting material

• Books
  – B. Stroustrup: The Design and Evolution of C++ (Addison Wesley 1994) – History and design
  – B. Stroustrup: The C++ Programming Language (Addison Wesley 2000) – Language features and semi-advanced programming techniques

• The ISO C++ standard committee’s site:
  – All documents from 1994 onwards
    • Search for “WG21”
  – Get the pdf of the “FCD” (most current standard draft)

• My home pages
  – Papers, FAQs, libraries, applications, compilers, …
    • Search for “Bjarne” or “Stroustrup”

• Compilers
  – Get GCC C++ 4.5 or 4.6
  – Get Microsoft VS 2010 C++
Course website

- Is being set up
Rules

• Plagiarism
  – Unfortunately, it has been observed
    • And the response is a 0 for the work (or worse)
  – You can use any source as long as you properly reference it
    • The worst that can happen is that you don’t get much credit

• Collaboration
  – Is strongly encourage
    • Except where explicitly no allowed
  – When you collaborate clearly indicate who did what
    • Or you might find yourself in trouble with the plagiarism rule

• If in doubt, ask me
  – The worst that can happen when you ask is that you get an answer you don’t like
Overview

- 1949-1978: Prehistory – Aims and Ideals
- 1979-1990: The early years – C with Classes and C++
- 1998-2010: Living in the real world – C++0x
Programming languages

• A programming language exists to help people express ideas
  – Programming language features exist to serve design and programming techniques
  – The real measure of value is the number, novelty, and quality of applications
Programming Languages

- Assembler
- Cobol
- Fortran
- Simula
- BCPL
- C
- C++
- Java
- C++0x
- C#

- Domain-specific abstraction
- General-purpose abstraction

- Direct mapping to hardware
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

---

[Handwritten notes] 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.

1949,
May 6th.

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
  – Continuous evolution ever since
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

• The most effective styles use a combination of techniques
  – Note: Inherent complexity of a solution always goes somewhere
    • if not in the language then in the run-time support system, libraries, the applications, etc.
C++ applications
(www.research.att.com/~bs/applications.html)

- Telecommunications
- Google, Amazon, …
- Microsoft applications and GUIs
- Linux tools and GUIs
- Financial
- Games
- PhotoShop
- Most browsers
- …

- Mars Rovers
- Marine diesel engines
- Cell phones
- Human genome project
- High-energy physics
- Micro electronics design and manufacturing
- …
C++ applications
C++ Applications

15 Fall'10 Stroustrup
C++ Applications

www.lextrait.com/vincent/implementations.html
Growth

• 1979-1991: the number of users doubled every 7.5 months
  – I could count
  – 0 to ~900,000 in 12 years

• 1991-2000: the numbers of users grew steadily to about 3 million
  – Reliable estimates are really hard to come by
  – ~900,000 to ~3,100,000 in 8 years

• 2000-2010: the number of users appear to be steady
  – Has the “infrastructure areas” been saturated?
  – There appears to be a shift from “applications” to “infrastructure” and a growth in “embedded systems”

• Billions of lines of C++ code in current use (N*1,000,000,000)
What’s distinctive about C++?

- **Stability**
  - Essential for real-world software
  - 1985-2010
  - 1978-2010 (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 (e.g. memory access) 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
Aims for C++

• Support real-world software developers
  – “better software now”
  – by “better” I mean correct, maintainable, efficient, portable, …

• Change the way people think about software
  – Object-oriented programming
  – Generic programming
  – Resource management
  – Error handling

• Functional, not academic, beauty
  – “even I could have designed a much prettier language”
    • – B.S. 1984 or so
Ideals

• The fundamental ideals for good design
  – Represent ideas directly in code
  – Represent independent ideas independently in code
  – Represent relationships among ideas directly in code
    • Hierarchical
    • Parametric
  – Combine ideas expressed in code freely
    • where and only where combinations make sense

• C++
  – Make these ideals viable for the largest possible range of application areas
    • “viable” includes “affordable” and “on available hardware”
    • “viable” includes “performs as well as the gold standard in a given area”
      – e.g. Fortran for scientific computation and C for systems programming
    • “viable” includes “in the hands of ordinary programmers”
Feedback loops

• We don’t know what would be the ideal programming language
  – “good” depends on
    • Application
    • Constraints (time, space, time-to-market, reliability, …)
    • Development context (tools, developer education/background management ,…)
    • …

• Proper experiments are almost impossible to conduct
  – Scale matters (#people, #installations, #lines of code, …)

• Emphasize diversity of applications
  – What worked where? Why? Why not?

• Develop design “rules of thumb”
Feedback loops

• Being best at just one or two things is not good enough
  – Nothing in a tool chain is irrelevant to a real-world user
  – Don’t ignore “irrelevant details”

• Listen to lots of users
  – They almost always focus on minor improvements
    • And ignore the major problems
  – They always have a point
  – Don’t listen too much to non-users
    • Incl. “visionaries” and academics

• Education is immensely important
  – You can’t (successfully) use what you don’t understand
  – Traditional education / self study / professional training
  – Listen to students
Feedback loops

• Build on what is known to work
  – In the relevant real-world application areas

• Don’t be satisfied with status quo
  – Idealism and risk-taking are necessary for progress

• Explore and experiment
  – Don’t trust small experiments with select developers in controlled environments

• Generalize
  – You can’t imagine all the applications and implications of your work

• Articulate design decisions
  – Rationales
  – Tutorials
Language features – 1979-1990

• C with Classes (1979-84)
  – Function argument declarations and checking
  – `const` (also in constant expressions)
  – Classes
  – Derived classes
  – Constructors, destructors
  – `new` and `delete`
  – Inline functions

• C++ (in 1983-86)
  – Overloading (incl. `=`, `[]`, and `()`)
  – `virtual` functions
  – Type-safe linkage

• C++ (1988-90)
  – Templates
  – Exceptions

Not in C until much later

Huge impact

Rather late
Basic resource management

- A resource can be memory, file handle, lock, socket, etc.

```cpp
class vector {
    vector(int s); // constructor: validate arguments, acquire resources
    ~vector();     // destructor: release resources
    // ...
};

void f(int s)
{
    vector v(s);   // construct v
    // ...
} // (implicitly) destroy v
```
Object-oriented programming

- Class hierarchies, dynamic lookup, and static interfaces

```cpp
class Shape {
    Point c; // common implementation detail: often a dumb idea
    Color col;
    public: // common user interface
        virtual void draw();
        virtual void move(Point p) { c=p; }
        virtual void rotate(int deg);
        // ...
};

class Circle : public Shape {
    Circle(Point cc, Color co);
    void rotate(int) {} // nice optimal algorithm
    // ...
};
```

Fall'10 Stroustrup
C++ ISO Standardization – Membership

• About 22 nations (8 to 12 at a meeting)
  – ISO WG21 meetings
  – Other technical meetings
• Membership have varied
  – 100 to 200+
    • 200+ members currently
  – 40 to 100+ at a meeting
    • ~60 currently
• Most members work in industry
• Most are volunteers
  – Even many of the company representatives
• Most major platform, compiler, and library vendors are represented
  – E.g., Apple, IBM, Intel, Microsoft, Sun
• End users are underrepresented
• Academia/Education is underrepresented
C++ ISO Standardization – Process

Formal, slow, bureaucratic, and democratic
- “the worst way, except for all the rest”
  (apologies to W. Churchill)

Most technical work happens
- in “working groups”
- In more specialized meetings
- electronically between meetings
- electronically during meetings
For C++, the ISO standards process is central

• Standard support needed for mainstream use
  – Huge potential for improvement of application code
  – For (far too) many “if it isn’t in the standard it doesn’t exist”
• Significant defense against vendor lock-in
• C++ has no rich owner
  – who can dictate changes, pay for design, implementation, marketing, etc.
• The C++ standards committee is the central forum of the C++ community
  – Endless discussions among people who would never meet otherwise
• The committee receives feedback from a broad section of the community
  – Much of it industrial
• The committee is somewhat proactive
  – Adds features not previously available in the C++ world
C++ ISO Standardization – Results

1998  ISO standard
   – 22-0 vote

2003  Technical Corrigenda
   – “bug fix release”; no new features

2008  Registration draft for C++0x
   – Feature freeze (in theory)

2010  Final Committee Draft
   – C++2011? (C++0B)

• Technical reports
  – Decimal floating point (2008)
  – Library2
  – Modularity (?)

1992  Covariant return types
1993  Run-time type identification (RTTI: `dynamic_cast`, `typeid`, and `type_info`)
      Declarations in conditions
      Overloading based on enumerations
      `namespaces`
      `mutable`
      New casts (`static_cast`, `reinterpret_cast`, and `const_cast`)
      A Boolean type (`bool`)
      Explicit template instantiation
      Explicit template argument specification in function template calls
1994  Member templates (“nested templates”)
      Class templates as template arguments
1996  In-class member initializers
      Separate compilation of templates (`export`)
      Template partial specialization
      Partial ordering of overloaded function templates

The sum is far more significant than the parts.
C++98 example: Resource management

- Standard library containers
  - with exception-safety guarantees (e.g., `vector`)
  - the techniques can be used by every user

- No resources are leaked
  - E.g. `vector` elements and file handles (handled by `ifstream`)
  - Destructors do cleanup
    - guaranteed, implicitly
  - Based on a simple and systematic view of resource management
    - Resources: e.g. locks, sockets, memory, thread handles, file handles
    - Exception safety guarantees
    - RAII

```cpp
void f(string s)
{
    vector<int> v;
    ifstream is(s);
    // ...
    int x;
    while (is>>x) {
      if (x<=0) throw Bad_value(x);
      v.push_back(x);
    }
    // ...
}
```
The STL

• Ideal: The most general and most efficient expression of an algorithm
  – Focus on algorithms
  – Separate algorithms from data
    • Using iterators
  – Go from the concrete to the abstract
    • Not the other way
  – Use compile-time resolution to eliminate overheads
    • Inlining and overloading
  – Where needed, parameterize with policies
    • E.g. sorting criteria
STL example: find_if

- Definition

```cpp
template<class Iter, class Pred>
Iter find_if(Iter first, Iter last, Pred p)
{
    while (first!=last && !p(*first)) // while not at end and predicate not met
        ++first; // advance to next element
    return first; // return the element reached
}
```

```cpp
pi = find_if(v.begin(), v.end(), Less_than<int>(42));
if (pi!=v.end()) {
    // found it!
}
```
C++0x: 2002-2010

• Overall goals
  – Make C++ a better language
    • for systems programming
    • for library building
  – Make C++ easier to teach and learn
    • generalization
    • better libraries

• Massive pressure for
  – More language features
  – Stability / compatibility
    • Incl. C compatibility

• Insufficient pressure for
  – More standard libraries
    • The committee doesn’t have the resources required for massive library development
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
    • Microsoft C++2010: Lambdas, auto
  – Standard library components are shipping widely
    • E.g. GCC, Microsoft, Boost
  – The last design points have been settled
    • We are processing 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
C++0x: Areas of change

- Machine model and concurrency
  - Memory model
  - Threads library, asynchronous return
  - Atomic API
  - ...

- Support for generic programming
  - `auto`, template aliases, Rvalue references, ...
  - General and uniform initialization
  - ...

- Etc.
  - improved `enums`
  - `long long`, C99 character types, etc.
  - ...

- Libraries
  - Resource management pointers
  - Regular expressions
  - Hashed containers
  - ...
C++0x features

- **Language (+27%)**
  - Uniform initializer syntax and generalized initializer lists
  - Move semantics (rvalue references)
  - Generalized constant expressions (**constexpr**)
  - Variadic templates
  - **auto** and **decltype** — type deduction from expressions
  - A for-statement for ranges
  - Scoped and strongly typed enumerations (**class enum**)
  - **nullptr** — Null pointer constant
  - Lambda expressions
  - ...

- **Library (+100%)**
  - Threads (**thread**)
  - Regular expression matching (**regex**)
  - Hash tables (e.g. **unordered_map**)
  - Resource management (e.g. **unique_ptr** and **shared_ptr**)
  - Garbage collection ABI
  - ...

The whole is much more than its parts
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, 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 (and 4.6 is on its way)
  – Standard library components are shipping widely
    • E.g. GCC, Microsoft, Boost
  – The last design points have been settled
    • barring formal requests from National Standards Bodies
      – And they tend to be very conservative
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:
    ```cpp
    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
  ```cpp
  vector<list<int>> vl; // note the “missing space”
  ```

- *Example:* simplified iteration
  ```cpp
  for (auto x : v) cout << x <<'
';
  ```

- *Note:* Experts don’t easily appreciate the needs of novices
  - Example of what we couldn’t get just now
    ```cpp
    string s = "12.3";
    double x = lexical_cast<double>(s); // extract value from string
    ```
Support both experts and novices

• *Example*: minor syntax cleanup
  
  ```cpp
  vector<list<int>> vl;  // note the “missing space”
  ```

• *Example*: simplified iteration
  
  ```cpp
  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
    
    ```cpp
    string s = "12.3";
    double x = lexical_cast<double>(s);  // extract value from string
    ```
Uniform initialization

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

```cpp
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
  
  ```c
  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 }
  };

Fall'10 Stroustrup
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)
  ```
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

• **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>`

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

Fall'10 Stroustrup
Prefer generality to specialization

• *Example:* Prefer improvements to abstraction mechanisms over separate new features
  – Inherited constructor
    
    ```cpp
template<class T> class Vector : std::vector<T> {
        using vector::vector<T>; // inherit all constructors
        // …
    };
    ```
  – Move semantics supported by rvalue references
    ```cpp
template<class T> class vector {
    // …
    void push_back(T&& x); // move x into vector
    // avoid copy if possible
    };
    ```

• *Problem:* people love small isolated features
Move semantics

- Often we don’t want two copies, we just want to move a value
  
  ```cpp
  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
  }
  ```

  ```cpp
  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 in the standard library

- Prefer unique ownership to shared ownership
  void f(shared_ptr<X>);
  void g(unique_ptr<X>);

  f(shared_ptr<X>(new X(1,2,3)));  // manipulate (shared) use count
                           // 0->1->2->1
  g(unique_ptr<X>(new X(1,2,3)));  // zero overhead (no use count)
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

```cpp
struct Point {
    int x, y;
    constexpr Point(int xx, int yy) : x{xx}, y{yy} { } // “literal type”
};
```

`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?
Increase type safety

• Approximate the unachievable ideal
  – *Example*: Strongly-typed enumerations
    ```cpp
    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
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”
Concurrence support

• Memory model
  – To guarantee our usual assumptions

• Support for concurrent systems programming
  – Atomic types for implementing concurrency support features
    • “Here be dragons”
    • Lock-free programming
  – Thread, mutex, lock, …
    • RAII for locking
    • Type safe

• A single higher-level model
  – async() and futures
async() and futures

- 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
{
    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();
}
```
Why did C++ succeed?

• Reasons
  – Low-level access plus abstraction mechanisms
    • Performance
    • Direct access to real hardware
    • Very general zero-overhead abstraction
  – C compatibility
  – A useful tool (from day #1)
  – Timing (“in the right decade”)
  – Non-proprietary – ISO standard
  – Stable
  – Evolving

“Being *best* at one or two things is not enough, you must be *good enough* at everything someone consider important”
Why did C++ succeed?

• Popular non-reasons
  – Just luck
    • For 25 years!
  – AT&T’s marketing might
    • Must be a joke 😊
  – It was first
    • Except for Ada, CommonLoops, Smalltalk, Eiffel, Objective C, Modula-2, C, Fortran, ML, …
  – Just C compatibility
    • Never 100%
  – It was cheapest
    • Not for most of its lifetime (incl. all the early years)
What is C++?

A multi-paradigm programming language

A hybrid language

A random collection of features

Embedded systems programming language

It’s C!

Low level!

Supports generic programming

An object-oriented programming language

Buffer overflows

Too big!

Template meta-programming!

Fall’10 Stroustrup
C++0x

• It *feels* like a new language  
  – Compared to C++98  
  – Just like C++98 did relative to early C++
• 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++

Key strength:
building software infrastructures and resource-constrained applications

A light-weight-abstraction programming language
Thanks!

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