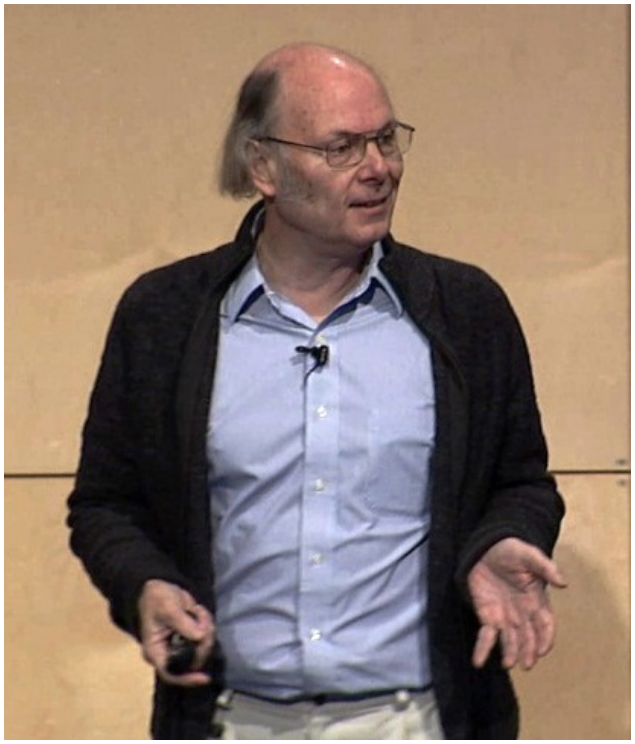
# C++14 and Concepts

Bjarne Stroustrup
Texas A&M University
www.stroustrup.com

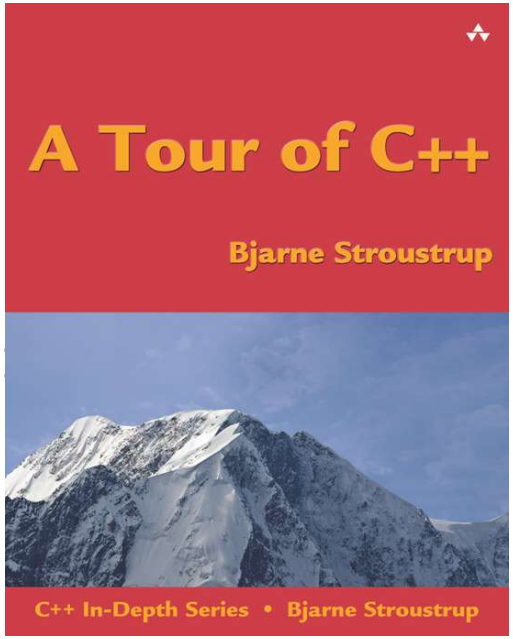
# Videos

- "Broad spectrum" C++ talks

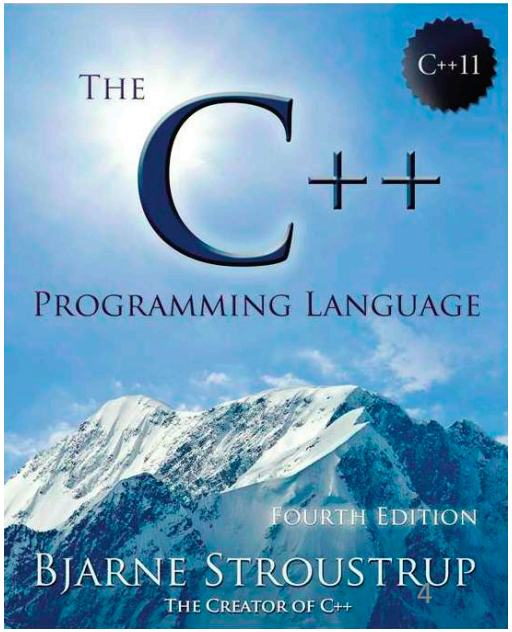


  - **The Essence of C++: With Examples in C++84, C++98, C++11, and C++14:.** http://channel9.msdn.com/Events/GoingNative/2013/Opening-Keynote-Bjarne-Stroustrup
  - **The Future is Here: C++ 11.** http://www.youtube.com/watch?v=jPruO9ZS-yY
  - **C++11 Style – a Touch of Class** http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style

# Books




- Available in paper and electronic forms
  - A Tour of C++.
    Addison Wesley.
    ISBN 978-0321958310. September 2013.
  - The C++ Programming Language (4th Edition)
    Addison-Wesley.
    ISBN 978-0321563842. May 2013.

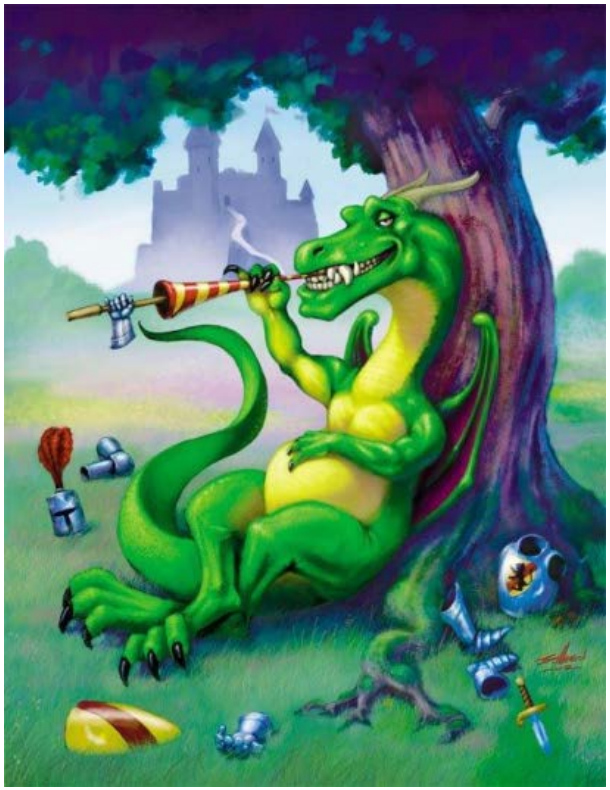


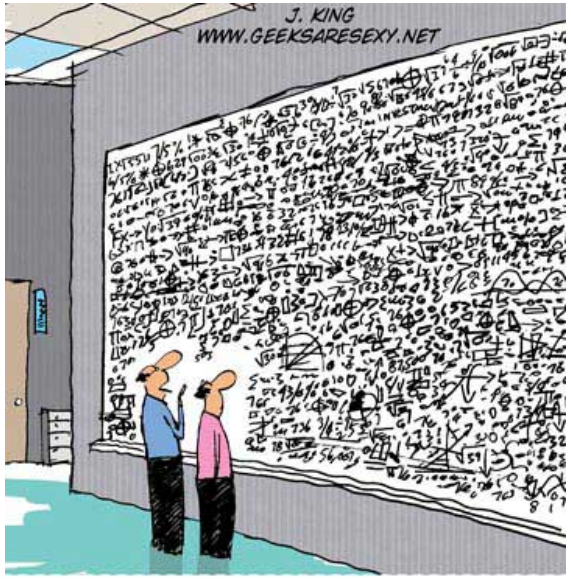
# C++11 – a major revision

- Making simple things simple
  - Uniform and universal initialization
  - Auto: deduce type from initializer
  - Range-for
  - Constexpr: generalized constant expressions
  - …
- Resource Management
  - Move semantics
  - Smart pointers
  - …
- Generic programming support
  - Lambdas
  - Variadic templates
  - Template aliases
  - …
- Concurrency
- Regular expressions, random numbers, …

# The real problems

- Help people to write better programs
  - Easier to write
  - Easier to maintain
  - Easier to achieve acceptable resource usage






*"...And that, in simple terms, is what's wrong with your software design."*

- The primary value of a programming language is in the applications written in it

# C++11

- Is a better approximation of my ideals for support of good programming
  - Significantly better than C++98
- Has tons of distracting “old stuff”
  - Going back to C in 1972
- We must focus on the essentials
  - And the “good stuff”
  - “Elegance ***and*** efficiency”
- C++11 is not the end, we can do much better still
  - Anyone who says *I have a perfect language* is a fool or a salesman
- Stability/compatibility is an important feature in itself
  - And not free

# Range for and move

- As ever, what matters is how features work in combination

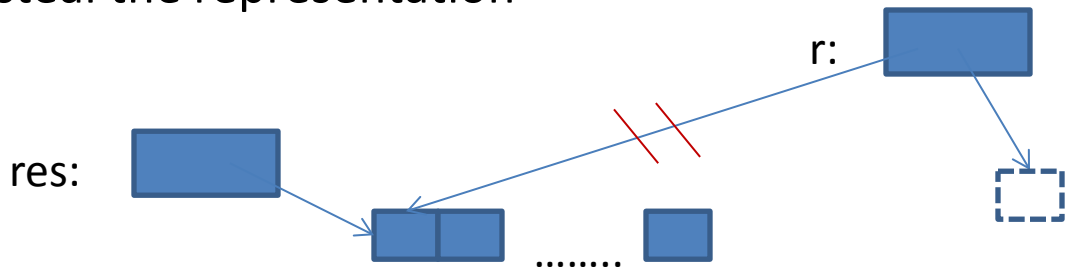
```
template<typename C, typename V>
vector<Value_type<C>*> find_all(C& c, V v)  // find all occurrences of v in c
{
        vector<Value_type<C>*> res;
        for (auto& x : c)
                if (x==v)
                        res.push_back(&x);
        return res;
}

string m {"Mary had a little lamb"};
for (const auto p : find_all(m,'a'))   // p is a char*
        if (*p!='a')
                cerr << "string bug!\n";
```

# Move semantics

- Return a **Matrix**

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
        Matrix r;
        // copy a[i]+b[i] into r[i] for each i
        return r;
}
Matrix res = a+b;
```

- Define move a constructor for **Matrix**
  - don't copy; "steal the representation"



# C++14

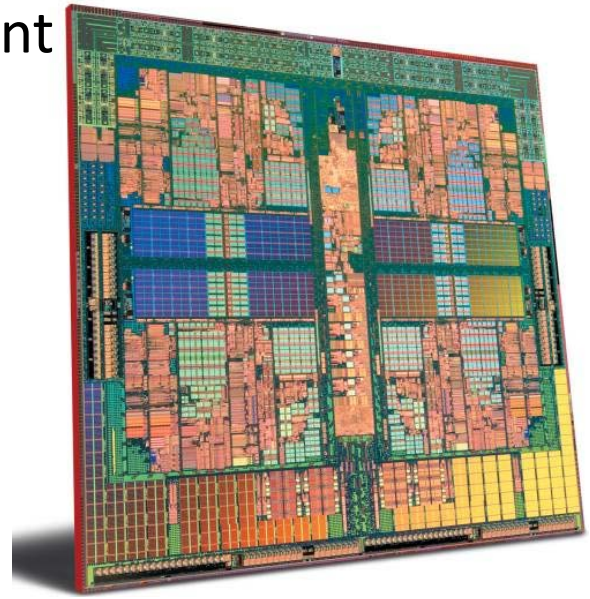
- Completing C++11
  - Cleaner
  - Simpler
  - Faster
  - No new major techniques

# C++14

- Language
  - More general constexpr functions
  - Polymorphic lambdas
  - ….
- Library
- Technical Specifications
  - File System
  - Concepts
  - Networking

# Tiny C++14 Features

- In the committee, nothing is trivial ☹
  - "Standardization is long periods of mind-numbing boredom interrupted by moments of sheer terror"
- To the user, the trivial can be most important
  - We must care about small things



- Brace-copy-initialization
  - **double d0 {27};**
  - **complex<double> d1 {d0};** // OK
  - **complex<double> d2 {d1};** // will become OK

Uniform and universal initialization is very important

# More literals

- Binary literal (language feature)
  - **0b101010**
- User-defined literals for standard library (in **std::literals**)
  - **"Hello, World"s** // a **std::string**
  - **2h + 15min + 125ms + 3000us + 23ns** // **std::duration**s
- Too controversial!
  - **2.5+1.2i** // a **std::complex** (also **il** and **i_f** )
  - maybe

# Polymorphic lambdas

# Templates

- Primary aim: to support efficient generic programming
  - Uncompromised generality
  - Uncompromised performance
  - Good interfaces
    - Well, two out of three ain't bad ☹
- Provides compile-time duck typing

# Duck Typing is Insufficient

- There are no proper interfaces
- Leaves error detection far too late
  - Compile- and link-time in C++
- Encourages a focus on implementation details
  - Entangles users with implementation
- Leads to over-general interfaces and data structures
  - As programmers rely on exposed implementation “details”
- Does not integrate well with other parts of the language
  - Teaching and maintenance problems
- We must think of generic code in ways similar to other code
  - Relying on well-specified interfaces (like OO, etc.)

# Generic Programming is just Programming

- *Traditional code*

```
double sqrt(double d);      // C++84: accept any d that is a double
double d = 7;
double d2 = sqrt(d);        // fine: d is a double
double d3 = sqrt(&d);       // error: &d is not a double
```

- Generic code

```
void sort(Container& c);    // C++14: accept any c that is a Container
vector<string> vs { "Hello", "new", "World" };
sort(vs);                   // fine: vs is a Container
sort(&vs);                  // error: &vs is not a Container
```

# C++14: Constraints aka “Concepts lite”

- How do we specify requirements on template arguments?
  - state intent
    - Explicitly states requirements on argument types
  - provide point-of-use checking
    - No checking of template definitions
  - use constexpr functions
- Voted as C++14 Technical Specification
- Design by B. Stroustrup, G. Dos Reis, and A. Sutton
- Implemented by Andrew Sutton in GCC
- There are no C++0x concept complexities
  - No concept maps
  - No new syntax for defining concepts
  - No new scope and lookup issues

# What is a Concept?

- Concepts are fundamental
  - They represent fundamental concepts of an application area
  - Concepts are come in “clusters” describing an application area
- We have always had concepts
  - C++: Integral, arithmetic
  - STL: forward iterator, predicate
  - Informally: Container, Sequence
  - Algebra: Group, Ring, …
- A concept has semantics (meaning)
  - Not just syntax
  - “**Subtractable**” is not a concept

# What is a Concept?

- Don’t expect to find a new fundamental concept every year
- A concept is ***not*** the minimal requirements for an implementation
  - An implementation does not define the requirements
  - Requirements should be stable
- Concepts support interoperability
  - There are relatively few concepts
  - We can remember a concept

# C++14 Concepts (Constraints)

- A concept is a predicate on one or more arguments
  - E.g. **Sequence<T>()** // *is T a Sequence?*

- Template declaration

```
template <typename S, typename T>
  requires Sequence<S>()
                && Equality_comparable<Value_type<S>, T>()
Iterator_of<S> find(S& seq, const T& value);
```

- Template use

```
void use(vector<string>& vs)
{
  auto p = find(vs,"Jabberwocky");
  // …
}
```

# C++14 Concepts: Error handling

- Error handling is simple (and fast)

```
template<Sortable Cont>
        void sort(Cont& container);

vector<double> vec {1.2, 4.5, 0.5, -1.2};
list<int> lst {1, 3, 5, 4, 6, 8,2};

sort(vec);      // OK: a vector is Sortable
sort(lst);      // Error at (this) point of use: Sortable requires random access
```

- Actual error message

  error: ‘list<int>’ does not satisfy the constraint ‘Sortable’

# C++14 Concepts: “Shorthand Notation”

- Shorthand notation

```
template <Sequence S, Equality_comparable<Value_type<S>> T>
        Iterator_of<C> find(S& seq, const T& value);
```

- We can handle essentially all of the Palo Alto TR
  - (STL algorithms) and more
    - Except for the axiom parts
  - We see no problems checking template definitions in isolation
    - But proposing that would be premature (needs work, experience)
  - We don’t need explicit **requires** much (the shorthand is usually fine)

# C++14 Concepts: Overloading

- Overloading is easy

```
template <Sequence S, Equality_comparable<Value_type<S>> T>
        Iterator_of<S> find(S& seq, const T& value);

template<Associative_container C>
        Iterator_type<C> find(C& assoc, const Key_type<C>& key);

vector<int> v { /* … */ };
multiset<int> s { /* … */ };
auto vi = find(v, 42);                  // calls 1st overload:
                                        // a vector is a Sequence
auto si = find(s, 12-12-12);            // calls 2nd overload:
                                        // a multiset is an Associative_container
```

# C++14 Concepts: Overloading

- Overloading based on predicates
  - specialization based on subset
  - Far easier than writing lots of tests

```
template<Input_iterator Iter>
        void advance(Iter& p, Difference_type<Iter> n) { while (n--) ++p; }

template<Bidirectional_iterator Iter>
      void advance(Iter& i, Difference_type<Iter> n)
      { if (n > 0) while (n--) ++p; if (n < 0) while (n++) --ip}

template<Random_access_iterator Iter>
        void advance(Iter& p, Difference_type<Iter> n) { p += n; }
```

- We don't say

  **Input_iterator < Bidirectional_iterator < Random_access_iterator**

  we compute it

# C++14 Concepts: Definition

- How do you write constraints?
  - Any **bool** expression
    - Including type traits and constexpr function
  - a **requires(expr)** expression
    - **requires()** is a compile time intrinsic function
    - **true** if **expr** is a valid expression
- To recognize a concept syntactically, we can declare it **concept**
  - Rather than just **constexpr**, e.g.

```
concept bool Ordered()
{
        return Regular<T>() && Less_than_comparable<T>();
}
```

# C++14 Concepts: “Terse Notation”

- We can use a concept name as the name of a type than satisfy the concept

```
void sort(Container& c);            // terse notation
```

  - means

```
template<Container __Cont>          // shorthand notation
    void sort(__Cont& c);
```

  - means

```
template<typename __Cont>           // explicit use of predicate
    requires Container<__Cont>()
        void sort(__Cont)& c;
```

  - Accepts any type that is a Container

```
vector<string> vs;
sort(vs);
```

# C++14 Concepts: “Terse Notation”

- We have reached the conventional notation
  - with the conventional meaning
- *Traditional code*

```
double sqrt(double d);      // C++84: accept any d that is a double
double d = 7;
double d2 = sqrt(d);        // fine: d is a double
double d3 = sqrt(&d);       // error: &d is not a double
```

- Generic code

```
void sort(Container& c);    // C++14: accept any c that is a Container
vector<string> vs { "Hello", "new", "World" };
sort(vs);                   // fine: vs is a Container
sort(&vs);                  // error: &vs is not a Container
```

# C++14 Concepts: “Terse Notation”

- Consider **std::merge**
- Explicit use of predicates:

```
template<typename For,
         typename For2,
         typename Out>
   requires Forward_iterator<For>()
   && Forward_iterator<For2>()
   && Output_iterator<Out>()
   && Assignable<Value_type<For>,Value_type<Out>>()
   && Assignable<Value_type<For2,Value_type<Out>>()
   && Comparable<Value_type<For>,Value_type<For2>>()
void merge(For p, For  q, For2 p2, For2 q2, Out p);
```

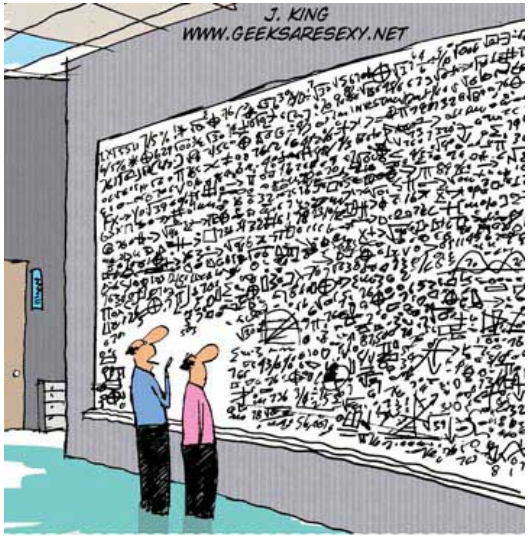
- Headache inducing, and **accumulate()** is worse

# C++14 Concepts: “Terse Notation”

- Better, use the shorthand notation

```
template<Forward_iterator For,
                Forward_iterator For2,
                Output_iterator Out>
                        requires Mergeable<For,For2,Out>()
void merge(For p, For  q, For2 p2, For2 q2, Out p);
```

- Quite readable




*“...And that, in simple terms, is what’s wrong with your software design.”*

# C++14 Concepts: “Terse Notation”

- Better still, use the “terse notation”:

```
Mergeable{For,For2,Out}    // Mergeable is a concept requiring three types
void merge(For p, For  q, For2 p2, For2 q2, Out p);
```

- The

  *concept-name { identifier-list }*

  notation introduces constrained names

- Make simple things simple!

# C++14 Concepts: “Terse Notation”

- Now we just need to define **Mergeable**:

```
template<typename For, typename For2, typename Out>
concept bool Mergeable()
{
        return Forward_iterator<For>()
          && Forward_iterator<For2>()
          && Output_iterator<Out>()
          && Assignable<Value_type<For>,Value_type<Out>>()
          && Assignable<Value_type<For2,Value_type<Out>>()
          && Comparable<Value_type<For>,Value_type<For2>>();
}
```

- It’s just a predicate

# “Paradigms”

- Much of the distinction between object-oriented programming, generic programming, and “conventional programming” is an illusion
  - based on a focus on language features
  - incomplete support for a synthesis of techniques
  - The distinction does harm
    - by limiting programmers, forcing workarounds

```
void draw_all(Container& c)     // is this OOP, GP, or conventional?
        requires Same_type<Value_type<Container>,Shape*>
{
    for_each(c, [](Shape* p) { p->draw(); } );
}
```

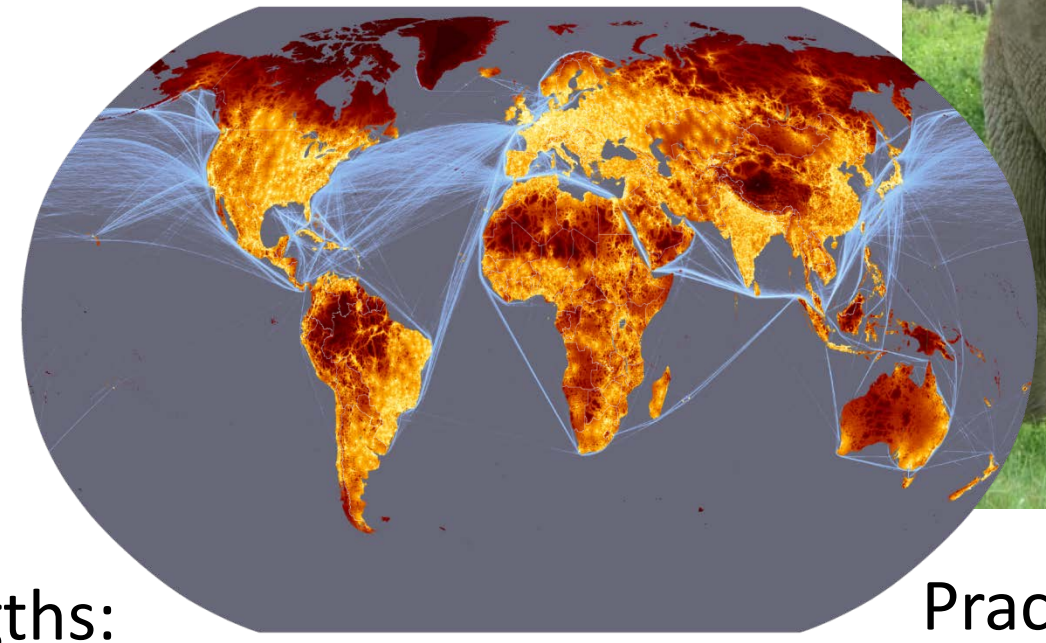
# Challenges

# Challenges

- Obviously, C++ is not perfect
  - How can we make programmers prefer modern styles over low-level code
    - which is far more error-prone and harder to maintain, yet no more efficient?
  - How can we make C++ a better language given the Draconian constraints of C and C++ compatibility?
  - How can we improve and complete the techniques and models (incompletely and imperfectly) embodied in C++?
- Solutions that eliminate major C++ strengths are not acceptable
  - Compatibility
    - link, source code
  - Performance
    - uncompromising
  - Portability
  - Range of application areas
    - Preferably increasing the range

# Questions?

C++: A light-weight abstraction programming language



Key strengths:

- software infrastructure
- resource-constrained applications

Practice type-rich programming