Parasol
Smarter computing.
Texas A&M University

C++14 and early thoughts about C++17

Bjarne Stroustrup

Texas A&M University

www.stroustrup.com

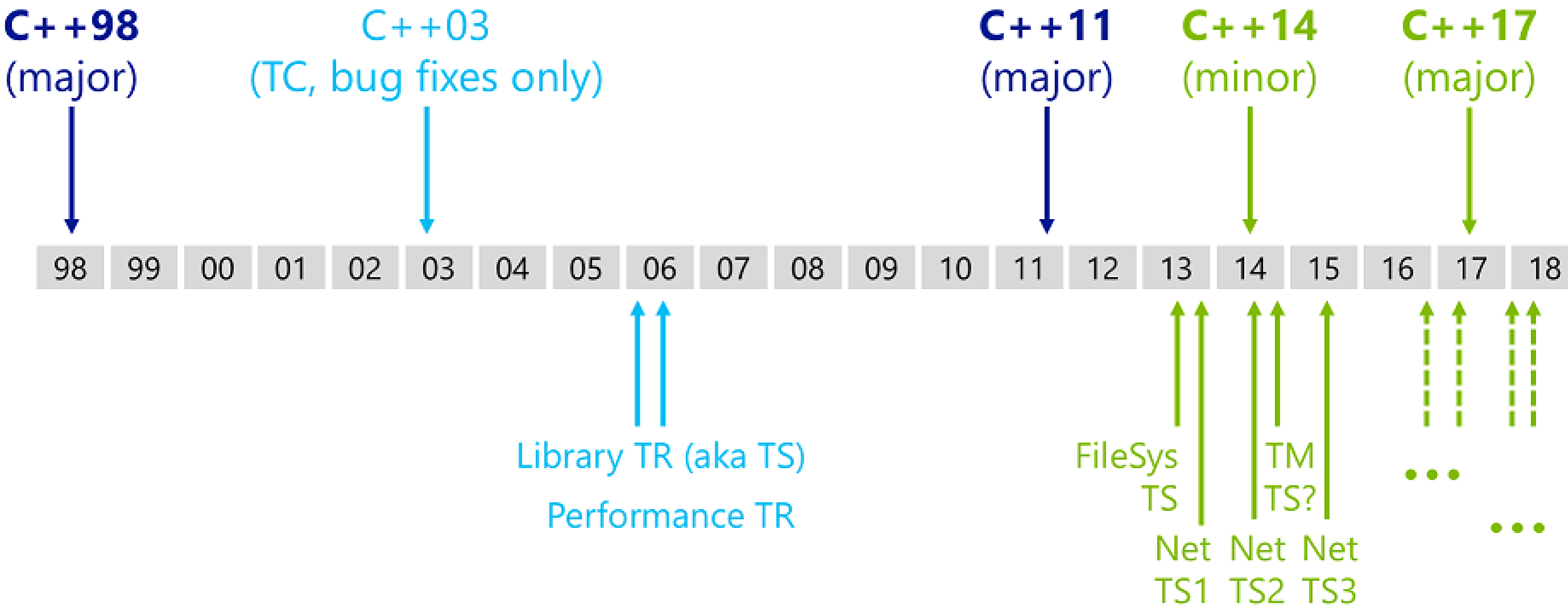
- … there is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things. For the reformer makes enemies of all those who profit by the old order, and only lukewarm defenders in all those who would profit by the new order …

The best
is the enemy of
the good

Overview

- The plan
- C++14
- C++17
- Dreams
- Q&A
  - Ask anything

ISO C++ “Releases”



- From Herb Sutter’s WG21 Oct’12 presentation

C++17 Groups

• **Core**, aka CWG	Mike Miller (EDG)
• **Library**, aka LWG	Alisdair Meredith (Bloomberg)
• **Evolution**, aka EWG	Bjarne Stroustrup (TAMU)
• **Library Evolution**	Beman Dawes
• **SG1, Concurrency**	Hans Boehm (HP)
• **SG2, Modules**	Doug Gregor (Apple)
• **SG3, File System**	Beman Dawes (Boost)
• **SG4, Networking**	Kyle Kloepper (Riverbed)
• **SG5, Transactional Memory**	Michael Wong (IBM)
• **SG6, Numerics**	Lawrence Crowl (Google)
• **SG7, Reflection**	Chandler Carruth (Google)
• **SG8, Concepts**	Matt Austern (Google)
• **SG9, Ranges**	Marshall Clow (Qualcomm)
• **SG10, Feature Test**	Clark Nelson (Intel)

- http://isocpp.org/std/meetings-and-participation

C++14

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

Possible C++14 Language Features

- Tiny
  - Brace-copy-initialization
  - Return type deduction for normal functions
  - Binary literals (e.g., **0b101010**)
  - Remove deprecated **++** for **bool**
  - Sized deallocation
  - **[[deprecated]]** attribute
  - Digit separators (e.g. _)
    - The bike shed
  - Several technical modifications and clarifications

Possible C++14 Language Features

- Minor
  - Constraints aka "concepts lite"
  - Generic (Polymorphic) Lambda Expressions
    - And "Terse templates"
  - Runtime-sized arrays with automatic storage duration
    - and **dynarray (**CWG, LWG)
  - Relaxing syntactic constraints on constexpr function definitions
    - In particular, I'd like an **if** and a ***simple*** **for** in a constexpr function
  - Constexpr variable templates
  - Allowing arbitrary literal types for non-type template parameters.
    - a stretch

- Could have major effects on programming style

Possible C++14 Library Features

- Literal suffixes N3531
- **dynarray** N3532
- **split()** N3593
- Filesystem N3505
- sort(v.begin(), v.end(), greater<>()); // N3421

How do we stay sane?

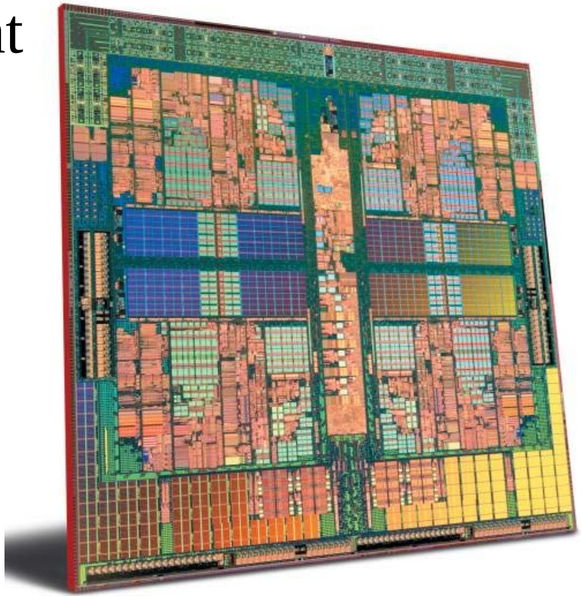
- How do we get good proposals accepted?
- How do we get bad proposals rejected?
- How do we get half-baked proposals baked?
- How do we tell the difference?
- How do we ensure that new proposals?
  - Integrate with the rest of the language
  - Don't simply duplicate existing features
  - Don't wreck our programming models

How do we stay sane?

- Rules of thumb
  - Don’t leave room for a lower level language below C++
    - Except machine code
  - Don’t pay for what you don’t use
    - Zero-overhead principle
  - Go from the concrete/specific to the abstract/general
    - Don’t abstract just for the sake of abstraction
    - Don’t add features just for theoretical completeness
    - Don’t add features just to follow fashion
  - Don’t wait for perfection
    - When you have something definitely beneficial to the community
    - Progress in stages

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

[[deprecated]]

- Deprecated attribute:
  - **int a [[deprecated]];**
  - **int a [[deprecated("message")]];**
  - The presence of the attribute has no normative effect. Implementations may use the deprecated attribute to produce a diagnostic message in case a name or entity is used by the program after the first declaration that specifies the attribute.
  - Put it on part of a library that you want people to stop using
  - Possibly the smallest "new feature" ever

- Competitor in the "smallest new feature" stakes
  - Ban the deprecated ++ for **bool**

Return type deduction

- Return type deduction for normal functions
  - **auto sq1 = [](double d) { return d*d; }** // OK
  - **auto sq2(double d) { return d*d; }** // will become OK

- About time too

- What about multiple return statements?
  - Accept if their types are identical

Digit separators (“the bike shed”)

- There are many alternatives, including
  - **123_456_789**
  - **123’456’789**
  - **123 456 789**
  - **123`456`789**
  - **123~456~789**
- A problem
  - Assume that _ (underscore) is the digit separator
    - You can construct alternative problems for alternative separators
  - Where do the user-defined literal suffix start?
  - **1001_1110_0001_1010_10** // _10 suffix for binary?
  - **0xDEAD_BEEF_code** // _code or ode suffix?
  - **0xDead__x** // x or _x or __x suffix?

Other literals

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

Sized Deallocation

- Allow an operator delete() with a size:

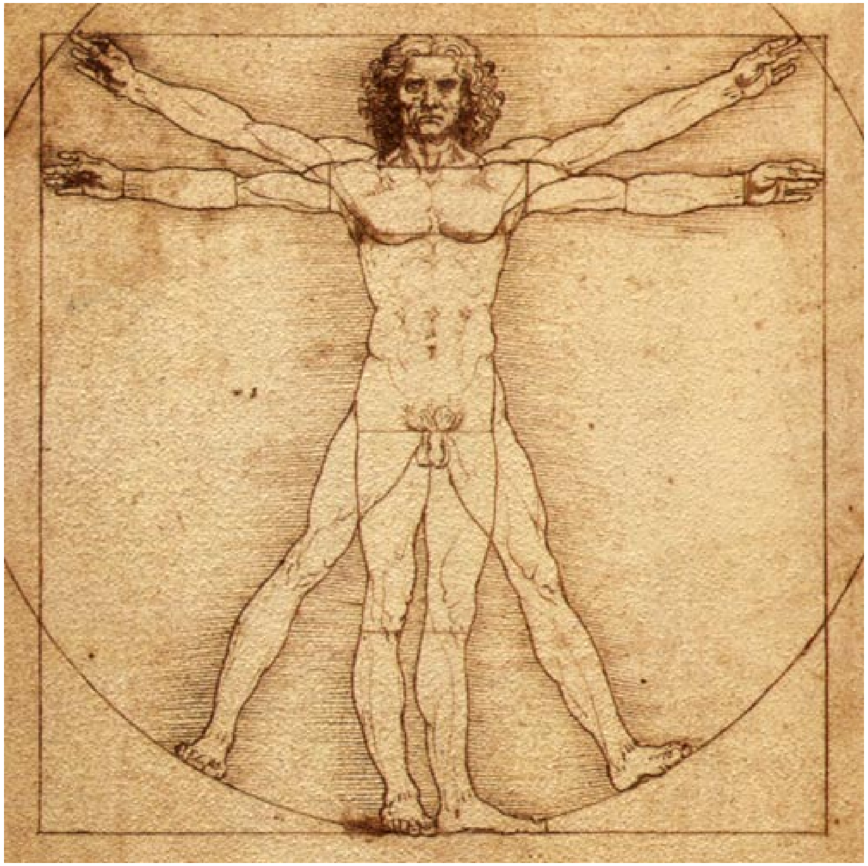
```
// as ever:
void* operator new(std::size_t) throw(std::bad_alloc);
void* operator new[](std::size_t) throw(std::bad_alloc);
void operator delete(void*) noexcept;
void operator delete[](void*) noexcept;

// new:
void operator delete(void*, std::size_t) noexcept;
void operator delete[](void*, std::size_t) noexcept;
```

- A **delete p** uses the version with size if that is declared
  - Otherwise, the version without a size

Small C++14 Features

- Some would call them medium
- Some are important



Runtime-sized arrays

- C99 VLAs is an abomination
  - Just look at a “pointer to VLA”
  - VLAs are optional in C11 (thanks WG14!)
- Runtime-sized arrays with automatic storage duration

```
void f(int n)
{
        double a1[n];
        double a2[100];
        double* p = a1;
        p = a2;                  // a1 and a2 are of  the same type
}
```

- **dynarray** to complement/complete run-time sized arrays.

```
dynarray<double> a3(n);          // like array<double,7>
```

  - no implicit conversion to **double***
  - **a3.size()**

Generalizing constexpr functions

- Why all of that pure functional programming?
  - Few people like it
  - Many people hate it
  - Can lead to inferior algorithms (in the run-time evaluated case)
- Most of all, I want
  - A simple for loop

Generalizing constexpr functions

- Roughly:
  - a constexpr function can contain anything that does not have side effects outside the function
  - For example
    - Local variables are ok
    - Loops are ok

Generalizing constexpr functions

- We have to be careful
  - Not to require a full C++ Interpreter at compile time
  - Not to leave details up to compiler writers (leading to incompatibilities)
  - For example, not
    - Lambdas affecting linkage
    - Exceptions (not impossible, but possibly hard, and what are the use cases?)
    - Variadic templates
    - New/delete
    - Undefined behavior (e.g., uninitialized variables)
- Writing constexpr functions will become much easier
  - but still constrained
  - "being able to do something is not sufficient reason for doing it."

Constexpr variable templates

- We have always had workarounds

```
template<typename T>
        struct numeric_limits {
                static constexpr bool is_modulo = false;        // value definition
};

// …

template<typename T>
        constexpr bool numeric_limits<T>::is_modulo;        // object definition

// …

auto m = numeric_limits<double>::is_modulo;        // use
```

Constexpr variable templates

- But we can do much better

```
template<typename T>
        constexpr bool is_modulo = false;       // value definition

auto m = is_modulo<double>;                      // use
```

- Notation matters

Constexpr variable templates

- We need to do better because literal types are getting popular
  - Naturally: type-rich compile-time programming
- We can do much better

```
namespace Pauli {
    template<typename T> using spin = matrix<T, 2>;
    template<typename T> constexpr spin<T> sigma1 = { { 0, 1 }, { 1, 0 } };
    template<typename T> constexpr spin<T> sigma2 = { { 0, -1i }, { 1i, 0 } };
    template<typename T> constexpr spin<T> sigma3 = { { 1, 0 }, { 0, -1 } };
}

auto s = Pauli::sigma1<double>;
```

Constraints aka “Concepts lite”

- How do we specify requirements on template arguments?
- Constraints
  - state intent
    - Explicitly states requirements on argument types
  - provides point-of-use checking
    - No checking of template definitions
  - are constexpr functions
- There are no C++0x concept complexities
  - No concept maps
  - No new syntax for defining concepts
  - No new scope and lookup issues
  - No semantic specification (axioms)
- Implemented by Andrew Sutton in GCC

Constraints aka “Concepts lite”

- 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

```
auto p = find(vs,"Jabberwocky");
```

Constraints aka “concepts lite”

- 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)

Constraints aka “Concepts lite”

- 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
sort(lst);      // Error at (this) point of use
```

- Actual error message

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

Constraints aka “Concepts lite”

- 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
auto si = find(s, 12-12-12);          // calls 2nd overload
```

Constraints aka “concepts lite”

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

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

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

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

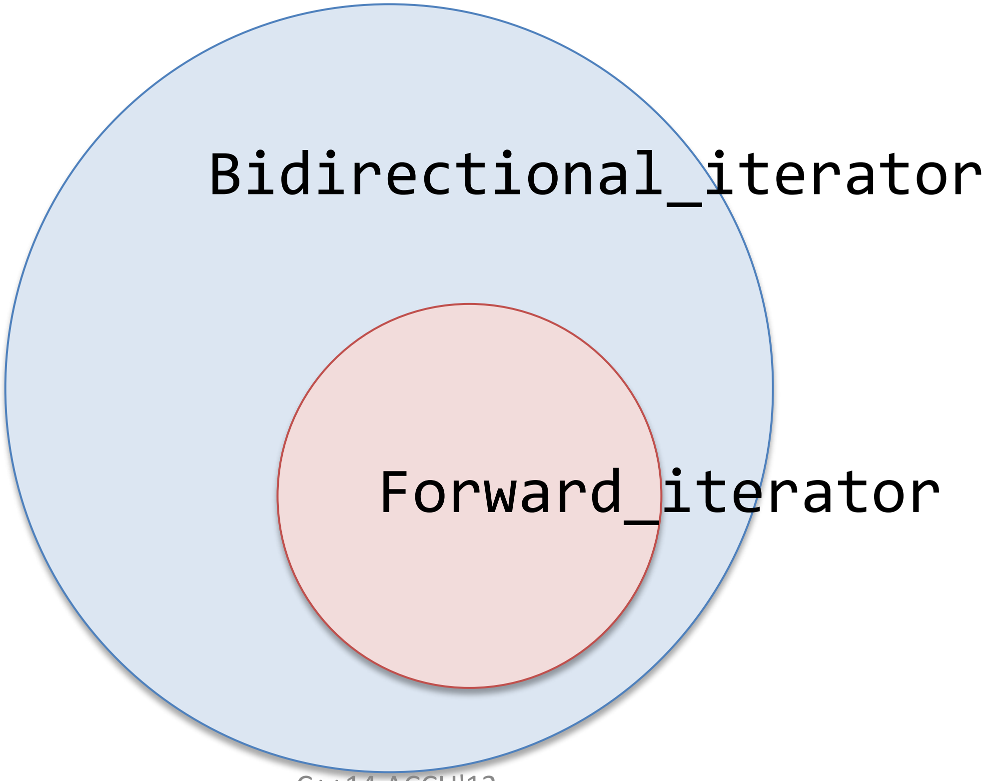
- We don’t say

```
Input_iterator < Bidirectional_iterator < Random_access_iterator
```

we compute it

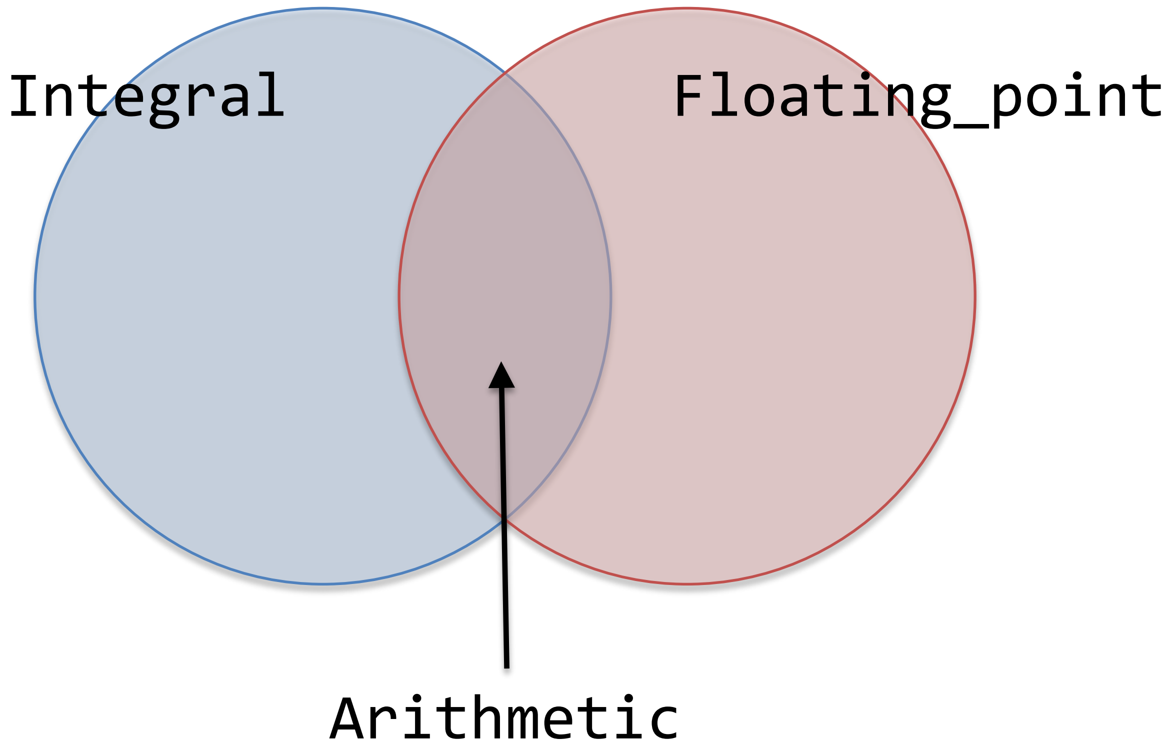
Conjunction and Refinement

Constraints that subsume others are *refinements*



Disjunction and Overlap

The disjunction of overlapping constraints



Constraints aka “Concepts lite”

- How do you write constraints?
  - Type traits and constexpr function will do
  - But we can do better with a standard mechanism for expressing type inquiries
  - SG8 asked us to devise one

- So we have
  - a **requires(e)** function that checks if **e** is a valid expression
    - Called **__is_valid_expr()** in the posted paper (N3580)

Constraints aka "Concepts lite"

- How do you write constraints?

```
template<typename T>                    // pseudo definition of
concept bool requires(T expr)           // intrinsic function
{
        // return true if expr is a valid expression
}
```

- We need a way to express convertibility

```
template<typename T>
concept bool Equality_comparable()
{
  return requires (T a, T b) {
    {a == b} -> bool;          // a can be compared to b using ==
                               // and returns something convertible to bool
    {a != b} -> bool;
  };
}
```

Generic (Polymorphic) Lambdas

- Problem: We must specify the type of a lambda argument
  - **for_each(begin(v), end(v),**
    [](decltype(*begin(v)) x){ std::cout << x; });
  - **auto get_size =**
    [](std::unordered_multimap<std::wstring,std::list<std::string>> const& m)
    { return m.size(); };
- In other contexts, we use **auto** to simplify notation
- We would rather write
  - **for_each(begin(v), end(v), [](auto& x){ std::cout << x; });**
  - **auto get_size = [](auto& m){ return m.size(); };**

“Terse Templates”

- How do we constrain a generic lambda?

```
vector<double>v;
// …
sort(v.begin(),v.end(),
          [](double x, double y) { return x%100 < y%100; });  // error
sort(v.begin(),v.end(),
          [](auto x, auto y) { return x%100 < y%100; }) ;       // error
```

- How do we get less verbose error messages?
- How do we get less verbose template definitions?

“Terse Templates”

- Consider a declaration

```
void sort(Cont& c);
```

- It means

```
template<Container Cont>          // Container is a constraint
void sort(Cont& c);
```

- It means

```
template<typename Cont>
            requires Container<Cont>()
void sort(Cont& c);
```

- Somehow, **Container** must be know to denote a constraint
  - Just use **concept** instead of **constexpr** in its function definition

“Terse Templates”

- Consider

```
void sort(Ran p, Ran q);
```

- Ran is a “random-access iterator” constraint

- How do we know that the two Ran s denote the same type?

```
using Randon_access_iterator{Ran};
```

- So

```
void sort(Ran p, Ran q);
```

- means

```
template<typename Ran>
        requires Random_access_iterator<Ran>()
void sort(Ran p, Ran q);
```

"Terse Templates"

- Consider **std::merge**:

```
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

“Terse Templates”

- Better:

```
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

“Terse Templates”

- Better still:

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

- The traditional notation for function declarations
  - A generalization of the traditional semantics

Parasol
Smarter computing.
Texas A&M University

“Terse Templates”

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

```
template<typename T1,T2,T3>
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

Possible C++14 Features

- suggestions?
  - The C++14 train is just about to leave the platform
  - The C++17 train will soon follow
  - http://isocpp.org/std

Dreams

- Things take time
  - What would like in C++ in 10 years time?
  - In 5 years time if you are really lucky?

- What would help C++ programmers ***a lot***?
  - That we don't already have a dozen people working on
    - Compile-time reflection
    - Task-based concurrency
    - Source code modules
    - Concepts
    - …
- How about "Never write another visitor!"?

Open Multi-methods

- Take a Hierarchy
  - **Class Shape { … };**
- Compute something base on two objects of arbitrary derived classes
  - **bool intersect(virtual Shape*, virtual Shape*);**
  - **bool intersect(Rectangle*, Rectangle*) override;**
  - **bool intersect(Circle*,Rectangle*) override;**
  - **…**
- When called, the correct overrider is picked
  - **Intersect(new Circle{p,100},new Rectangle{p1,p2});**
- Faster than double dispatch
  - P. Pirkelbauer, Y. Solodkyy, B. Stroustrup: *Design and Evaluation of C++ Open Multi-Methods*. In Science of Computer Programming (2009)

FP-style Pattern Matching

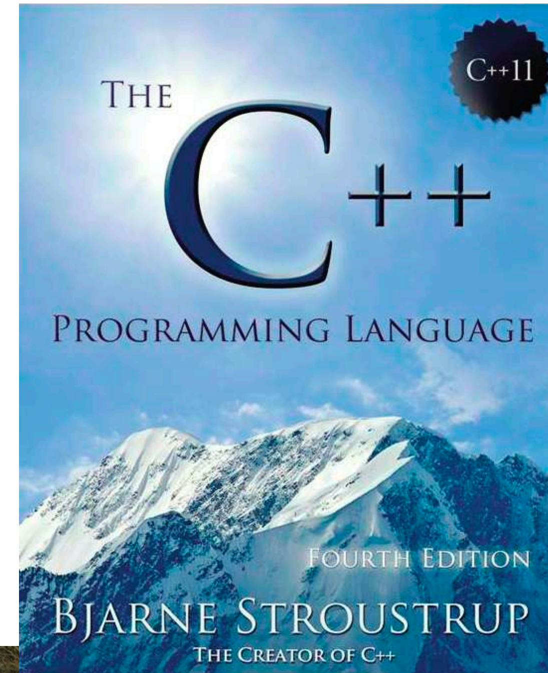
- Dispatch on type of value
  - And bind results to variables

```
int eval(Expr& e)          // expression evaluator
{
    Expr* a,*b;
    int n;
    Match(e)
      Case(C<Value>(n))  return n;
      Case(C<Plus>(a,b)) return eval(*a) + eval(*b);
      Case(C<Minus>(a,b))        return eval(*a) - eval(*b);
      Case(C<Times>(a,b))        return eval(*a) * eval(*b);
      Case(C<Divide>(a,b))       return eval(*a) / eval(*b);
    EndMatch
}
```

- This is running C++11 code (using a small library), and fast
  - Y. Solodkyy, G. Dos Reis, and B. Stroustrup: *Open and Efficient Type Switch for C++*. Proc. OOPSLA'12.

Parasol
Smarter computing.
Texas A&M University

Questions?



static if and/or "Concepts lite"

- Totally unbiased executive summary
  - **static if** is a total abomination
    - Unstructured, can do everything (just like **goto**)
    - Complicates static analysis (AST-based tools get hard to write)
    - Blocks the path for concepts
    - Specifies how things are done (implementation)
    - Is three slightly different "ifs" using a common syntax
    - Redefines the meaning of common notation (such as { … })
    - Proposed by Walter Brown, Herb Sutter, Andrei Alexandrescu
  - Constraints (aka "Concepts lite") is the best thing since sliced bread
    - Simply constrains definitions
    - Can be the first part of a radically simpler "concepts"
    - Specifies what is to be done (intent)
    - Proposed by Andrew Sutton, Bjarne Stroustrup