Modern C++ for High Performance Computing

Andrew Lumsdaine
Northwest Institute for Advanced Computing
Pacific Northwest National Laboratory
University of Washington
Seattle, WA
### Backstory

- Group at Indiana University (Jaakko Jarvi, Jeremy Siek, Jeremiah Willock, Doug Gregor, et al)
- Contributed numerous features to C++11 (variadic templates, lambda, decltype, enable_if, &c.)
- (But not “concepts”)
- New institution, new course, new approach
- C++11 is a new language
- Clang is awesome
- Visual studio code is awesome
What this webinar is not about: Language Features

C++11 has many new features compared to C++03

The features per se aren’t the source of power

A language is not just its features

C++11 is not just C++03 with more features

C++11 has many features, period
Pedagogy recapitulates ontogeny considered harmful

Maturation of languages over time (ontogeny)

Features added, etc

Unfortunately, C++11 is taught as C plus 45 years of new features
Pedagogy recapitulates ontogeny

<table>
<thead>
<tr>
<th>Table of Contents</th>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: A Tutorial Introduction.</td>
<td>1: Introduction to Computers and C++</td>
</tr>
<tr>
<td>2: Types, Operators, and Expressions.</td>
<td>2: Introduction to C++ Programming, Input/Output and Operators</td>
</tr>
<tr>
<td>3: Control Flow.</td>
<td>3: Introduction to Classes, Objects, Member Functions and Strings</td>
</tr>
<tr>
<td>4: Functions and Program Structure.</td>
<td>4: Algorithm Development and Control Statements: Part 1</td>
</tr>
<tr>
<td>5: Pointers and Arrays.</td>
<td>5: Control Statements: Part 2; Logical Operators</td>
</tr>
<tr>
<td>6: Structures.</td>
<td>6: Functions and an Introduction to Recursion</td>
</tr>
<tr>
<td>7: Input and Output.</td>
<td>7: Class Templates array and vector; Catching Exceptions</td>
</tr>
<tr>
<td>8: The UNIX System Interface.</td>
<td>8: Pointers</td>
</tr>
<tr>
<td>Appendix A.</td>
<td>9: Classes: A Deeper Look</td>
</tr>
<tr>
<td>Appendix B.</td>
<td>10: Operator Overloading; Class string</td>
</tr>
</tbody>
</table>

Anonymous modern C++ book

NORTHWEST INSTITUTE for ADVANCED COMPUTING

Better Scientific Software: Modern C++ for HPC
Andrew Lumsdaine June 12, 2019
What this webinar is about: Tasteful programming

• How to write your programs in C++11
• Not how to write C++11 in your programs
• Code is a medium for communication (a language)
• Any language has syntax and vocabulary – and style

A language that doesn't affect the way you think about programming, is not worth knowing
What is a programming language for?

• Managing (not causing) complexity

• What is the most powerful mental tool for managing complexity?
Two simple rules for writing (tasteful) software

THE FIRST RULE OF WRITING SOFTWARE
DON'T BE CLEVER

THE SECOND RULE OF WRITING SOFTWARE
DON'T BE CLEVER
C++ core guidelines development philosophy

P.1: Express ideas directly in code
P.2: Write in ISO Standard C++
P.3: Express intent
P.4: Ideally, a program should be statically type safe
P.5: Prefer compile-time checking to run-time checking
P.6: What cannot be checked at compile time should be checkable at run time
P.7: Catch run-time errors early
P.8: Don't leak any resources
P.9: Don't waste time or space
P.10: Prefer immutable data to mutable data
P.11: Encapsulate messy constructs, rather than spreading through the code
P.12: Use supporting tools as appropriate
P.13: Use support libraries as appropriate
C++ Core Guidelines

In: Introduction
P: Philosophy
I: Interfaces
F: Functions
C: Classes and class hierarchies
Enum: Enumerations
R: Resource management
ES: Expressions and statements
Per: Performance
CP: Concurrency
E: Error handling
Con: Constants and immutability
T: Templates and generic programming
CPL: C-style programming
SF: Source files
SL: The Standard library

A: Architectural Ideas
N: Non-Rules and myths
RF: References
Pro: Profiles
GSL: Guideline support library
NL: Naming and layout
FAQ: Frequently asked questions
Appendix A: Libraries
Appendix B: Modernizing code
Appendix C: Discussion
Appendix D: Tools support
Glossary
To-do: Unclassified proto-rules

GSL: Guidelines support library.
C++ Performance Core Guidelines (selected)

Per.1: Don’t optimize without reason
Per.2: Don’t optimize prematurely
Per.3: Don’t optimize something that’s not performance critical
Per.4: Don’t assume that complicated code is necessarily faster than simple code
Per.5: Don’t assume that low-level code is necessarily faster than high-level code
Per.6: Don’t make claims about performance without measurements
Per.14: Minimize the number of allocations and deallocations
Per.19: Access memory predictably
**Case Study: Numerical Linear Algebra**

- Building abstractions with data: A Vector class
- Building abstractions with procedures: A Matrix class

P.1: Express ideas directly in code
P.2: Write in ISO Standard C++
P.3: Express intent
P.8: Don't leak any resources
P.9: Don't waste time or space
P.11: Encapsulate messy constructs, rather than spreading through the code
P.12: Use supporting tools as appropriate
P.13: Use support libraries as appropriate
Vector Desiderata

- **Math**
- \( x \in \mathbb{R}^N \)
- **Access with subscript** \( x_i \)
- \( \alpha(x + y) = \alpha x + \alpha y \)
- **Code**
- **Vector** \( x(N) \)
- **Access elements with “subscript”** \( x(i) \)
- \( a \ast x(i) \)

**Definition.** (Halmos) A vector space is a set \( V \) of elements called vectors satisfying the following axioms:

1. To every pair \( x \) and \( y \) of vectors in \( V \) there corresponds a vector \( x + y \) called the sum of \( x \) and \( y \) such a way that
   - (a) addition is commutative, \( x + y = y + x \)
   - (b) addition is associative, \( x + (y + z) = (x + y) + z \)
   - (c) there exists in \( V \) a unique vector \( 0 \) (called the origin) such that \( x + 0 = x \) for every vector \( x \), and
   - (d) to every vector \( x \) in \( V \) there corresponds a unique vector \( -x \) such that \( x + (-x) = 0 \)

2. To every pair \( a \) and \( x \) where \( a \) is a scalar and \( x \) is a vector in \( V \), there corresponds a vector \( ax \) in \( V \) called the product of \( a \) and \( x \) in such a way that
   - (a) multiplication by scalars is associative \( a(bx) = (ab)x \), and
   - (b) \( 1x = x \) for every vector \( x \).

3. (a) Multiplications by scalar is distributive with respect to vector addition. \( a(x + y) = ax + ay \)
   - (b) multiplication by vectors is distributive with respect to scalar addition \( (a + b)x = ax + by \)
Class Vector

class Vector {
public:
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

    double& operator()(size_t i) { return storage_[i]; } const double& operator()(size_t i) const { return storage_[i]; }

    size_t num_rows() const { return num_rows_; }

private:
    size_t num_rows_; 
    std::vector<double> storage_; 
};
Class Vector

```cpp
class Vector {
public:
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

    double& operator()(size_t i) { return storage_[i]; }
    const double& operator()(size_t i) const { return storage_[i]; }

    size_t num_rows() const { return num_rows_; }

private:
    size_t num_rows_;  // C.9: Minimize exposure of members
    std::vector<double> storage_; // C.4: Make a function a member only if it needs direct access to the representation of the class
};
```

C.41: A constructor should create a fully initialized object

C.49: Prefer initialization to assignment in constructors

C.4: Make a function a member only if it needs direct access to the representation of the class

F.5: If a function is small and time-critical, declare it inline
class Vector {
public:
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

    double& operator()(size_t i) { return storage_[i]; }
    const double& operator()(size_t i) const { return storage_[i]; }

    size_t num_rows() const { return num_rows_; }

private:
    size_t num_rows_;    // C.31: All resources acquired by a class must be released by the class's destructor
    std::vector<double> storage_;  // C.31: All resources acquired by a class must be released by the class's destructor
};

C.20: If you can avoid defining any default operations, do

C.31: All resources acquired by a class must be released by the class's destructor
class Vector {
public:
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

    double& operator()(size_t i) { return storage_[i]; }
    double& operator()(size_t i) const { return storage_[i]; }
    size_t num_rows() const { return num_rows_; }

private:
    size_t num_rows_;  
    std::vector<double> storage_; 
};
Using Vector class

```cpp
int main() {
    const size_t size = 1024;

    Vector x(size), y(size);
    for (size_t i = 0; i < x.size(); ++i) {
        x(i) = i;
    }

    Vector z = add(x, y);

    Vector w(size);
    add(x, y, w);

    Vector u = x + y;
}
```

- ES.20: Always initialize an object
- ES.20: Always initialize an object
- ES.21: Don’t introduce a variable before you need it
- ES.22: Don’t declare a variable until you have a value to initialize it with
Copy constructors (before)

```cpp
Vector operator+(Vector& x, Vector& y) {
    Vector z(x.num_rows());
    // ...
    return z;
}

int main() {
    Vector u(1024), v(1024), w(1024);
    u = v;
    u = v + w;
    Vector x = u + v;
}
```

1. **Constructor**
2. **Return by value**
3. **Copy assignment**
4. **Copy assignment**
5. **Copy constructor**
6. **Call operator+**
   - Calls constructor (for temporary)
   - Return by value: copy constructor
     - (compiler tricks: NVRO)
Move: Overload on value class

```cpp
class Vector {
public:
  Vector(size_t);
  Vector();
  Vector(const Vector&);
  Vector(Vector&&);
  Vector& operator=(const Vector&);
  Vector& operator=(Vector&&);
  ~Vector();
  // ...
};
```

- Copy constructor
- Move constructor
- Copy Assignment
- Move assignment
- Takes const reference
- Takes rvalue reference
- Takes rvalue reference
- Takes const reference
- Overload on whether parameter is temporary or not
Move and rvalue references

```cpp
Vector operator+(Vector& x, Vector& y) {
    Vector z(x.num_rows());
    // ...
    return z;
}

int main() {
    Vector u(1024), v(1024), w(1024);
    Vector x = u + v;
    u = v;
    u = v + w;
}
```

- **Constructor**
- **Move (or elided move)**
- **Return by value**
- **Copy constructor not called**
- **Returned by value from function**
- **Copy assignment**
- **Not copy assignment**
- **Move from temporary**

1. [https://godbolt.org/z/e7t60N](https://godbolt.org/z/e7t60N)
2. Better Scientific Software: Modern C++ for HPC
   Andrew Lumsdaine   June 12, 2019
Swap

Old generic swap template: 3 deep copies

New generic swap template: 3 moves (shallow copies)

Specialized (class specific) overload for efficient swap

Ready for (move) assignment

1 template<typename T>
2 void old_swap(T& a, T& b) {
3     T tmp = a;
4     a = b;
5     b = tmp;
6 }

1 template<typename T>
2 void new_swap(T& a, T& b) {
3     T tmp = std::move(a);
4     a = std::move(b);
5     b = std::move(tmp);
6 }

1 int main() {
2     Vector u(1024), v(1024);
3     old_swap(u, w);
4     new_swap(u, w);
5 }
### Keep It Simple: Argument Passing

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>Cheap or impossible to copy (e.g., int, unique_ptr)</td>
<td>X f()</td>
</tr>
<tr>
<td>In/Out</td>
<td>Cheap to move (e.g., vector&lt;T&gt;, string)</td>
<td>f(X&amp;)</td>
</tr>
<tr>
<td>In</td>
<td>Moderate cost to move (e.g., array&lt;vector&gt;, BigPOD)</td>
<td>f(const X&amp;)</td>
</tr>
<tr>
<td>In &amp; retain “copy”</td>
<td>Don’t know (e.g., unfamiliar type, template)</td>
<td></td>
</tr>
</tbody>
</table>

“Cheap” $\approx$ a handful of hot int copies

“Moderate cost” $\approx$ memcpy hot/contiguous $\sim$1KB and no allocation

* or return `unique_ptr<X>/make_shared_<X>` at the cost of a dynamic allocation
## Keep It Simple: Argument Passing

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Special Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out</td>
<td>Cheap or impossible to copy (e.g., int, unique_ptr)</td>
<td></td>
</tr>
<tr>
<td>In/Out</td>
<td>Cheap to move (e.g., vector&lt;T&gt;, string) or Moderate cost to move (e.g., array&lt;vector&gt;, BigPOD) or Don’t know (e.g., unfamiliar type, template)</td>
<td></td>
</tr>
<tr>
<td>In</td>
<td>Expensive to move (e.g., BigPOD[], array&lt;BigPOD&gt;)</td>
<td></td>
</tr>
</tbody>
</table>

- **Out**
  - `f()`

- **In/Out**
  - `f(X&)`

- **In**
  - `f(X)`
    - `f(const X&)`

- **In & retain copy**
  - `f(const X&) + f(X&&) & move`

- **In & move from**
  - `f(X&&)`

* or return `unique_ptr<X>/make_shared_<X>` at the cost of a dynamic allocation

** special cases can also use perfect forwarding (e.g., multiple in+copy params, conversions)
We are using Vectors just like built-in types

And Vectors are non-trivial compound types

Resource Acquisition is Initialization (RAII)

```c
int main() {
    Vector x(1024), y(1024);
    Vector u = foo(x);
    Vector v = bar(x);
    v = foo(x);
    u = bar(x);
    return 0;
}
```

```c
int main() {
    int x = 1024, y = 1024;
    int u = foo(x);
    int v = bar(x);
    v = foo(x);
    u = bar(x);
    return 0;
}
```
Resource Acquisition is Initialization (RAII)

- The “big six” (plus ordinary constructor) give us complete control over the lifetime of the resources contained in the object
- Resource Acquisition Is Initialization (RAII)
- Let each resource have an owner in some scope and by default be released at the end of its owners scope
- Note *we never use “new” or “delete”* (or, worse, malloc() and free()) – and never should use them
- Vector is a “resource handle”
- No abstraction penalty (no funny business with copies)

R.10: Avoid malloc() and free()

R.11: Avoid calling new and delete explicitly
Arithmetic operators

```cpp
Vector operator+(const Vector& x, const Vector& y) {
    Vector z(x.num_rows());
    for (size_t i = 0; i < x.num_rows(); ++i) {
        z(i) = x(i) + y(i);
    }
    return z;
}
```
Arithmetic operators

Vector operator\*(const double& a, const Vector& x) {

    Vectory(x.num_rows());
    for (size_t i = 0; i < x.num_rows(); ++i) {
        y(i) = a * x(i);
    }

    return y;
}
### Arithmetic operators

```cpp
Vector& operator+=(Vector& y, const Vector& x) {
    for (size_t i = 0; i < x.num_rows(); ++i) {
        y(i) += x(i);
    }
    return y;
}
```

- **Return by reference**
- **Note no construction**
Timing comparisons

1 Vector x(size), y(size);
2 double a = 3.14159;
3 for (size_t j = 0; j < iter; ++j) {
4     y += a*x;
5 }

Operator notation

Raw loop
Per.6: Don’t make claims about performance without measurements

Per.14: Minimize the number of allocations and deallocations.

Due to Temporary
Vector axpy with no temporary

```cpp
Vector x(size), y(size);
scalar a = 3.14159;

for (size_t j = 0; j < iter; ++j) {
    y += a * x;
}
```

No temporary

As fast as hand-written

As fast as daxpy()
Lazy evaluation

```cpp
struct scaledVector {
    public:
        scaledVector(const double& a, const Vector& v) : scalar_(a), vector_(v) {}
        const double& scalar_;  
        const Vector& vector_;  
    }

    scaledVector operator*(const double& a, const Vector& x) {
        return scaledVector(a, x);
    }

    Vector& operator+=(Vector& y, const scaledVector& x) {
        for (size_t i = 0; i < y.num_rows(); ++i) {
            y(i) += x.scalar_ * x.vector_(i);
        }
        return y;
    }
}
```

- Resist urge to be clever
- Wrap up vector and scalar for later
- Scalar times vector doesn’t do anything but wrap up scalar and vector
- When we use the scaled vector, we do the scaling
- Cf: MTL
You should be able to explain the design decisions of this class.

Note 1D storage
void lu(Matrix& A) {
    size_t m = A.numRows(), n = A.numCols();
    for (size_t k = 0; k < m - 1; ++k) {
        for (size_t i = k + 1; i < m; ++i) {
            double z = A(i, k) / A(k, k);
            A(i, k) = z;
            for (size_t j = k + 1; j < n; ++j)
                A(i, j) -= z * A(k, j);
        }
    }
}

LU implemented only with Matrix accessors

Partial pivoting takes two more lines of code

void lu(Matrix& A, std::vector<size_t>& perm) {
    size_t m = A.numRows(), n = A.numCols();
    std::iota(perm.begin(), perm.end(), 0);
    for (size_t k = 0; k < m - 1; ++k) {
        pivot(A, k, perm);
        for (size_t i = k + 1; i < m; ++i) {
            double z = A(i, k) / A(k, k);
            A(i, k) = z;
            for (size_t j = k + 1; j < n; ++j)
                A(i, j) -= z * A(k, j);
        }
    }
}

Again, only need specified interface (we will generalize this)
void multiply(const Matrix& A, const Matrix&B, Matrix&C) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j) {
      for (size_t k = 0; k < A.num_cols(); ++k) {
        C(i,j) += A(i,k) * B(k,j);
      }
    }
  }
}
Hoisting and tiling

```cpp
void hoistedTiledMultiply2x2(const Matrix& A, const Matrix&B, Matrix&C) {
    for (size_t i = 0; i < A.num_rows(); i += 2) {
        for (size_t j = 0; j < B.num_cols(); j += 2) {
            double t00 = C(i, j);  // Well known optimization
            double t01 = C(i, j + 1);
            double t10 = C(i + 1, j);
            double t11 = C(i + 1, j + 1);
            for (size_t k = 0; k < A.num_cols(); ++k) {
                t00 += A(i, k) * B(k, j);
                t01 += A(i, k) * B(k, j + 1);
                t10 += A(i + 1, k) * B(k, j);
                t11 += A(i + 1, k) * B(k, j + 1);
            }
            C(i, j) = t00;  // Written with external
            C(i, j + 1) = t01;
            C(i + 1, j) = t10;
            C(i + 1, j + 1) = t11;
        }
    }
}
```
void blockedTiledMultiply2x2(const Matrix& A, const Matrix&B, Matrix&C) {
    const int blocksize = std::min(A.num_rows(), 32);
    for (size_t ii = 0; ii < A.num_rows(); ii += blocksize) {
        for (size_t jj = 0; jj < B.num_cols(); jj += blocksize) {
            for (size_t kk = 0; kk < A.num_cols(); kk += blocksize) {
                for (size_t i = ii; i < ii+blocksize; i += 2) {
                    for (size_t j = jj; j < jj+blocksize; j += 2) {
                        for (size_t k = kk; k < kk+blocksize; ++k) {
                            C(i , j ) += A(i , k) * B(k, j );
                            C(i , j+1) += A(i , k) * B(k, j+1);
                            C(i+1, j ) += A(i+1, k) * B(k, j );
                            C(i+1, j+1) += A(i+1, k) * B(k, j+1);
                        }
                    }
                }
            }
        }
    }
}

Well known optimization

Written with external interface of Matrix class
Hierarchical memory optimizations

Matrix Multiply Abstraction Penalty

- matrix_0
- matrix_1
- matrix_2
- matrix_3
- matrix_4

Gflop/s vs. Matrix Size
Vectorization with intrinsics

```c
for (int i = ii; i < ii+blocksize; i += 4) {
    for (int j = jj, jb = 0; j < jj+blocksize; j += 4, jb += 4) {
        __m256d t0x = _mm256_load_pd(&C(i, j));
        __m256d t1x = _mm256_load_pd(&C(i+1,j));
        __m256d t2x = _mm256_load_pd(&C(i+2,j));
        __m256d t3x = _mm256_load_pd(&C(i+3,j));
        for (int k = kk, kb = 0; k < kk+blocksize; k += kb) {
            __m256d bx = _mm256_setr_pd(BB(jb,kb), BB(jb+1,kb), BB(jb+2,kb), BB(jb+3,kb));
            __m256d a0 = _mm256_broadcast_sd(&A(i ,k));
            t0x = _mm256_mul_pd(bx, a0);
            __m256d a1 = _mm256_broadcast_sd(&A(i+1,k));
            t1x = _mm256_mul_pd(bx, a1);
            __m256d a2 = _mm256_broadcast_sd(&A(i+2,k));
            t2x = _mm256_mul_pd(bx, a2);
            __m256d a3 = _mm256_broadcast_sd(&A(i+3,k));
            t3x = _mm256_mul_pd(bx, a3);
        }        _mm256_store_pd(&C(i, j), t0x);
        _mm256_store_pd(&C(i+1,j), t1x);
        _mm256_store_pd(&C(i+2,j), t2x);
        _mm256_store_pd(&C(i+3,j), t3x);
    }
}
```
Clang can do it better

for (int i = ii; i < ii+blocksize; i += 2) {
    for (int j = jj, jb = 0; j < jj+blocksize; j += 2, jb += 2) {
        double t00 = C(i,j); double t01 = C(i,j+1);
        double t10 = C(i+1,j); double t11 = C(i+1,j+1);
        for (int k = kk, kb = 0; k < kk+blocksize; ++k, ++kb) {
            t00 += A(i , k) * BB(jb , kb);
            t01 += A(i , k) * BB(jb+1, kb);
            t10 += A(i+1, k) * BB(jb , kb);
            t11 += A(i+1, k) * BB(jb+1, kb);
        }
        C(i, j) = t00; C(i, j+1) = t01;
        C(i+1,j) = t10; C(i+1,j+1) = t11;
    }
}

Fused Multiply-Add

256 bit registers

vmovupd (%r8,%r13,8), %ymm4
vmovupd (%r11,%r13,8), %ymm5
vfmaddd231pd %ymm4, %ymm5, %ymm3
vmovupd -32 (%r9,%r13,8), %ymm6
vfmaddd231pd %ymm4, %ymm5, %ymm2
vmovupd (%rdx,%r13,8), %ymm4
vfmaddd231pd %ymm5, %ymm4, %ymm1
vfmaddd231pd %ymm6, %ymm4, %ymm0
vmovupd (%rcx,%r13,8), %ymm4
vfmaddd231pd %r11,%r13,8), %ymm5
vfmaddd231pd %ymm4, %ymm5, %ymm3
vfmaddd231pd (%r9,%r13,8), %ymm6
vfmaddd231pd %ymm4, %ymm6, %ymm2
vfmaddd231pd (%rbx,%r13,8), %ymm4
vfmaddd231pd %ymm5, %ymm4, %ymm1
vfmaddd231pd %ymm6, %ymm4, %ymm0
With clang optimizations

Matrix Multiply Abstraction Penalty

- matrix_0
- matrix_1
- matrix_2
- matrix_3
- matrix_4
Parallelization with tasks

Matrix Matrix Product Performance

GFlops

Matrix Dimension

128 256 512 1024 2048 4096

1 2 4 7 8 9
The Standard Template Library

- In early-mid 90s Stepanov, Musser, Lee applied principles of *generic programming* to C++
- Leveraged templates / parametric polymorphism

std::set  std::list  ForwardIterator
std::map   ReverseIterator
std::vector RandomAccessIterator
...

std::for_each  std::sort  std::accumulate  std::min_element
...

Containers  Iterators  Algorithms
template <typename Matrix>
void lu(Matrix& A) {
    size_t m = A.numRows(), n = A.numCols();
    for (size_t k = 0; k < m - 1; ++k) {
        for (size_t i = k + 1; i < m; ++i) {
            double z = A(i, k) / A(k, k);
            A(i, k) = z;
            for (size_t j = k + 1; j < n; ++j)
                A(i, j) -= z * A(k, j);
        }
    }
}

template <typename Matrix>
void lu(Matrix& A, std::vector<size_t>& perm) {
    size_t m = A.numRows(), n = A.numCols();
    std::iota(perm.begin(), perm.end(), 0);
    for (size_t k = 0; k < m - 1; ++k) {
        pivot(A, k, perm);
        for (size_t i = k + 1; i < m; ++i) {
            double z = A(i, k) / A(k, k);
            A(i, k) = z;
            for (size_t j = k + 1; j < n; ++j)
                A(i, j) -= z * A(k, j);
        }
    }
}
A cautionary tale

“I’ve assigned this problem in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the description into a programming language of their choice; a high-level pseudo code was fine… Ninety percent of the programmers found bugs in their programs (and I wasn’t always convinced of the correctness of the code in which no bugs were found).”

- Jon Bentley, Programming Pearls, 1986

This must be a complicated algorithm!
Binary search solution

```cpp
int* lower_bound(int* first, int* last, int x) {
    while (first != last) {
        int* middle = first + (last - first) / 2;
        if (*middle < x) first = middle + 1;
        else last = middle;
    }
    return first;
}
```

```cpp
template<typename ForwardIterator, typename T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const T& x) {
    while (first != last) {
        auto middle = first + (last - first) / 2;
        if (*middle < x) first = middle + 1;
        else last = middle;
    }
    return first;
}
```

Not an indictment of “kids these days”

Programming is really hard

When you get it right, make it generic
We use name “InputIt” to hint to programmer that this should be an InputIterator

Concepts (finally) with C++20

Generic accumulate

std::accumulate

Defined in header <numeric>

\[
\text{template \< \text{class InputIt, class T > \>}
\]

\[
\text{T accumulate( InputIt first, InputIt last, T init );}
\]

\[
\text{template \< \text{class InputIt, class T, class BinaryOperation > \>}
\]

\[
\text{T accumulate( InputIt first, InputIt last, T init, BinaryOperation op );}
\]

Computes the sum of the given value init and the elements in the range \([\text{first, last}]\). The first version uses \text{operator+} to sum up the elements, the second version uses the given binary function \text{op}, both applying \text{std::move} to their operands on the left hand side (since C++20).

- \text{op} must not have side effects. (until C++11)
- \text{op} must not invalidate any iterators, including the end iterators, or modify any elements of the range involved. (since C++11)

**Type requirements**

- \text{InputIt} must meet the requirements of InputIterator.
- \text{T} must meet the requirements of CopyAssignable and CopyConstructible.
Sort

std::sort

Defined in header `<algorithm>`

```cpp
template< class RandomIt >
void sort( RandomIt first, RandomIt last );

template< class ExecutionPolicy, class RandomIt >
void sort( ExecutionPolicy&& policy, RandomIt first, RandomIt last );

template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );

template< class ExecutionPolicy, class RandomIt, class Compare >
void sort( ExecutionPolicy&& policy, RandomIt first, RandomIt last, Compare comp );
```

Type requirements

- RandomIt must meet the requirements of ValueSwappable and RandomAccessIterator.
- The type of dereferenced RandomIt must meet the requirements of MoveAssignable and MoveConstructible.
- Compare must meet the requirements of Compare.
Execution Policies

```cpp
std::execution::seq, std::execution::par, std::execution::par_unseq

Defined in header `<execution>

inline constexpr std::execution::sequenced_policy seq { /* unspecified */ };

inline constexpr std::execution::parallel_policy par { /* unspecified */ };

inline constexpr std::execution::parallel_unsequenced_policy par_unseq { /* unspecified */ };
```
Parallel standard library algorithms

- std::adjacent_difference
- std::adjacent_find
- std::all_of
- std::any_of
- std::copy
- std::copy_if
- std::copy_n
- std::count
- std::count_if
- std::equal
- std::fill
- std::fill_n
- std::find
- std::find_end
- std::find_first_of
- std::find_if
- std::find_if_not
- std::generate
- std::generate_n
- std::includes
- std::inner_product
- std::inplace_merge
- std::is_heap
- std::is_heap_until
- std::is_partitioned
- std::is_sorted
- std::is_sorted_until
- std::lexicographical_compare
- std::max_element
- std::merge
- std::min_element
- std::minmax_element
- std::mismatch
- std::move
- std::none_of
- std::nth_element
- std::partial_sort
- std::partial_sort_copy
- std::partition
- std::partition_copy
- std::remove
- std::remove_copy
- std::remove_copy_if
- std::remove_if
- std::replace
- std::replace_copy
- std::replace_copy_if
- std::replace_if
- std::reverse
- std::reverse_copy
- std::rotate
- std::rotate_copy
- std::search
- std::search_n
- std::set_difference
- std::set_intersection
- std::set_symmetric_difference
- std::set_union
- std::sort
- std::stable_partition
- std::stable_sort
- std::swap_ranges
- std::transform
- std::uninitialized_copy
- std::uninitialized_copy_n
- std::uninitialized_fill
- std::uninitialized_fill_n
- std::unique
- std::unique_copy

Where is accumulate?

There is no parallel accumulate

Why not?
### New parallel algorithms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for_each</code></td>
<td>similar to <code>std::for_each</code> except returns void</td>
</tr>
<tr>
<td><code>for_each_n</code></td>
<td>applies a function object to the first n elements of a sequence</td>
</tr>
<tr>
<td><code>reduce</code></td>
<td>similar to <code>std::accumulate</code>, except out of order</td>
</tr>
<tr>
<td><code>exclusive_scan</code></td>
<td>similar to <code>std::partial_sum</code>, excludes the ith input element from the ith sum</td>
</tr>
<tr>
<td><code>inclusive_scan</code></td>
<td>similar to <code>std::partial_sum</code>, includes the ith input element in the ith sum</td>
</tr>
<tr>
<td><code>transform_reduce</code></td>
<td>applies a functor, then reduces out of order</td>
</tr>
<tr>
<td><code>transform_exclusive_scan</code></td>
<td>applies a functor, then calculates exclusive scan</td>
</tr>
<tr>
<td><code>transform_inclusive_scan</code></td>
<td>applies a functor, then calculates inclusive scan</td>
</tr>
</tbody>
</table>

Instead of accumulate
Reduce

`std::experimental::parallel::reduce`

Defined in header `<experimental/numeric>`

```cpp
template<class InputIt>
typename std::iterator_traits<InputIt>::value_type reduce(
    InputIt first, InputIt last);
```

```cpp
template<class ExecutionPolicy, class InputIterator>
typename std::iterator_traits<InputIt>::value_type reduce(
    ExecutionPolicy&& policy, InputIt first, InputIt last);
```

```cpp
template<class InputIt, class T>
T reduce(InputIt first, InputIt last, T init);
```

```cpp
template<class ExecutionPolicy, class InputIt, class T>
T reduce(ExecutionPolicy&& policy, InputIt first, InputIt last, T init);
```

```cpp
template<class InputIt, class T, class BinaryOp>
T reduce(InputIt first, InputIt last, T init, BinaryOp binary_op);
```

```cpp
template<class ExecutionPolicy, class InputIt, class T, class BinaryOp>
T reduce(ExecutionPolicy&& policy,
    InputIt first, InputIt last, T init, BinaryOp binary_op);
```
Example

```cpp
{ Timer t; t.start();
  for (size_t k = 0; k < loops; ++k)
    result = std::accumulate(&v(0), &v(v.num_rows()), 0.0);
  t.stop();
  std::cout << "std::accumulate result " << result << " took " << t.elapsed() << " ms\n"; }

{ Timer t; t.start();
  for (size_t k = 0; k < loops; ++k)
    result = std::reduce(pstl::execution::seq, &v(0), &v(v.num_rows()), 0.0);
  t.stop();
  std::cout << "std::reduce result " << result << " took " << t.elapsed() << " ms\n"; }
```

Regular accumulate

Sequential execution
Example

```cpp
{ Timer t; t.start();
    for (size_t k = 0; k < loops; ++k)
        result = std::reduce(pstl::execution::par, &v(0), &v(v.num_rows()), 0.0);
    t.stop();
    std::cout << "std::reduce result " << result << " took " << t.elapsed() << " ms\n"; }

{ Timer t; t.start();
    for (size_t k = 0; k < loops; ++k)
        result = std::reduce(pstl::execution::par_unseq, &v(0), &v(v.num_rows()), 0.0);
    t.stop();
    std::cout << "std::reduce result " << result << " took " << t.elapsed() << " ms\n"; }
```
Results

std::accumulate result -2310.8 took 1155 ms
std::reduce result -2310.8 took 1167 ms
std::reduce result -2310.8 took 329 ms
std::reduce result -2310.8 took 337 ms
Developing your code

- That includes (especially) mental labor
- Use productivity tools
- **VS code**
- Intellisense
What about …?

- Muscle memory for typing is not the same as productivity (know the difference)
  - Stretch yourself
For More Information

- AMATH 483/583 web site (shameless plug)
  - https://lums658.github.io/amath583s19/
- C++ Core Guidelines
- Tour of C++
- C++ Programming Language (4th edition)
- http://cppreference.com
- andrew.lumsdaine@pnnl.gov

Insomnia? There is a podcast
Sequels (again, about writing tastefully)

- C++ threads, tasks, futures
- C++ lambda
- constexpr
- ranges / range-based for
- Generic programming, templates, concepts
- decltype
- Tuples, array, variadic templates
- shared_ptr<T>, et al

- C++ and OpenMP
- C++ and MPI
- Thrust
Thanks and Acknowledgments

- BSSW Program
- Mike Heroux, Lois Curfman-McInnes, David Bernholdt, Hai Ah Nam, Osni Marques
- Marcin Zalewski
- Bjarne Stroustrup, Alex Stepanov, Sean Parent
- Students of AMATH 483/583 (and CSEP 524)
Creative Commons BY-NC-SA 4.0 License

© Andrew Lumsdaine, 2017-2019

Except where otherwise noted, this work is licensed under

https://creativecommons.org/licenses/by-nc-sa/4.0/