AMATH 483/583
High Performance Scientific Computing

STL, Parallel STL

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

A programming problem

- "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

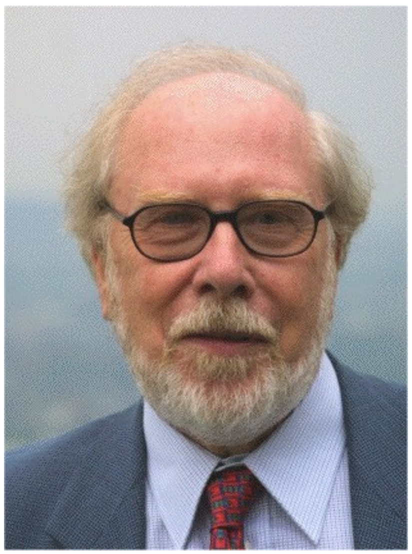
This must be a complicated algorithm!

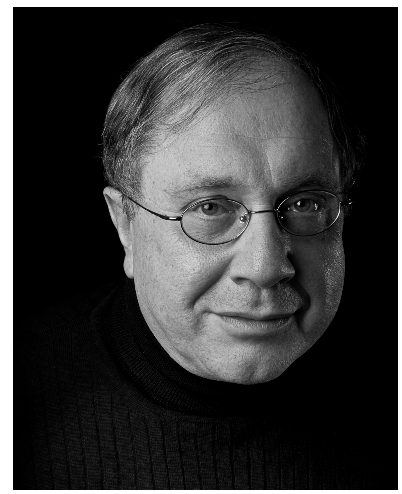
Binary search solution

```
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;
}
```

Let an expert write binary search

- Once and for all





Matrix-Matrix Product

```
template <typename MatrixType>
MatrixType operator*(const MatrixType& A, const MatrixType& B) {
  MatrixType C(A.num_rows(), B.num_cols());
  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);
      }
    }
  }
  return C;
}
```

This will work for any type that meets requirements for MatrixType

Constructor

Accessor

```
NewMatrix A(32, 32);
NewMatrix B(32, 32);
NewMatrix C = A * B;
```

```
RowMatrix A(32, 32);
RowMatrix B(32, 32);
RowMatrix C = A * B;
```

```
ColMatrix A(32, 32);
ColMatrix B(32, 32);
ColMatrix C = A * B;
```

```
Matrix A(32, 32);
Matrix B(32, 32);
Matrix C = A * B;
```

Generic Programming Methodology

1. Study the concrete implementations of an algorithm
2. **Lift** away unnecessary requirements to produce a more abstract algorithm
   a) Catalog these requirements.
   b) Bundle requirements into **concepts**.
3. Repeat the lifting process until we have obtained a generic algorithm that:
   a) Instantiates to efficient concrete implementations.
   b) Captures the essence of the "higher truth" of that algorithm.

Before

```
double sum(double* x, size_t N) {
  double s = 0;
  for (size_t i = 0; i < N; ++i)
    s += x[i];
  return s;
}
```

Before

```
float sum(float* x, size_t N) {
  float s = 0;
  for (size_t i = 0; i < N; ++i)
    s += x[i];
  return s;
}
```

Exactly the same loop

C++ (function) templates

```
template <typename T>
T sum(T* x, size_t N) {
  T s = 0;
  for (size_t i = 0; i < N; ++i)
    s += x[i];
  return s;
```

Introduce template

Template parameter

Begin

Return a T

End

Array of T

Declare and initialize a T

Function body

C++ (function) templates

```
template <typename T>
T sum(T* x, size_t N)
  T s = 0;
  for (size_t i = 0; i < N; ++i)
    s += x[i];
  return s;
}
```

At this level

```
double* dx;
size_t  dN;
double  dz = sum(dx, dN);
```

Bind T to be a double

```
float*  dx;
size_t  dN;
float   dz = sum(dx, dN);
```

Bind T to be a float

R… ts for sum so f…

We have a thing we want to sum over

Beginning of the thing

End of the thing

```
template <typename T>
T sum(T* x, size_t N) {
  T s = 0;
  for (size_t i = 0; i < N; ++i)
    s += x[i];
  return s;
}
```

Go through the thing

The type of s can be set to zero

Get a value out of the thing

The value is not the same as the thing

The value of the thing can be added to s

The value is not the same as the thing

Lifting

```
template <typename T>
T sum (T* begin, T* end) {
  T s = 0;
  for (T* p = begin; p < end; ++p) {
    s += *p;
  }
  return s;
}
```

We don't know what kind of thing, so parameterize

From begin to end

Use pointer

Dereference pointer

Get a value

OK for linked list?

Almost, actually

Lifting

```
template <typename T>
T sum (T* begin, T* end) {
  T s = 0;
  for (T* p = begin; p < end; ++p) {
    s += *p;
  }
  return s;
}
```

Go to next element

Get value

Linked list can do this

Lifting

```
template <typename T>
T sum (T* begin, T* end) {
  T s = 0;
  for (T* p = begin; p < end; ++p) {
    s += *p;
  }
  return s;
}
```

```
double sum(element* x) {
  double s = 0;
  while (x != nullptr) {
    s += x->val;
    x = x->next;
  }
  return s;
}
```

begin

Get value

Go to next element

end

We can do all the things needed for sum

But we can't use it with our sum

Because of syntax

For your consideration (son)

Element "thing"

```
struct element_ptr {
  element_ptr(element* x) : x(x) {}
  element_ptr operator++() { x = x->next; return x; }
  element_ptr operator++(int) { element* y = x; x = x->next; return y; }
  double operator*() { return x->val; }
  bool operator==(element_ptr y) { return x == y.x; }
  bool operator!=(element_ptr y) { return x != y.x; }

  element* x;
};
```

We also need to compare

Get value

Go to next

Lifting

```
template <typename T>
T sum (T begin, T end) {
  T s = 0;
  for (T p = begin; p != end; ++p) {
    s += *p;
  }
  return s;
}
```

```
struct element_ptr {
  element_ptr(element* x) :
  element_ptr operator++() {                eturn x; }
  element_ptr operator++(int) { element* y = x; x = x->next; return y; }
  double operator*() { return x->val; }
  bool operator==(element_ptr y) { return x == y.x; }
  bool operator!=(element_ptr y) { return x != y.x; }

  element* x;
};
```

Get value

Compare for equality

Check

Wrong type for s

Will s be compatible with “0”

Move to next

Check

Check

Lifting

Rename some things

```
template <typename Iter, typename T>
T sum (Iter begin, Iter end, T init) {
  for (T p = begin; p != end; ++p) {
    init += *p;
  }
  return init;
}
```

And pass in initial value of s

Parameterize the type of s

```
struct element_ptr {
  element_ptr(element* x) : x(x) {}
  element_ptr operator++() { x = x->next; return x; }
  element_ptr operator++(int) { element* y = x; x = x->next; return y; }
  double operator*() { return x->val; }
  bool operator==(element_ptr y) { return x == y.x; }
  bool operator!=(element_ptr y) { return x != y.x; }

  element* x;
};
```

Finall

Lets us iterate through our thing

The thing is holding values of type T

```
template <typename ForwardIterator, typename T>
T sum(ForwardIterator begin, ForwardIterator end, T init) {
  while (begin != end)
    init += *begin++;
  return init;
}
```

Compare

Use iterators to mark begin and end of what we want to sum

Get value

Move to next

Add to init

Requirements

```
template <typename ForwardIterator, typename T>
T sum(ForwardIterator begin, ForwardIterator end, T init) {
  while (begin != end)
    init += *begin++;
  return init;
}
```

If the type we bind to ForwardIterator has these expressions, we can use sum

- Need dereference – *i
- Need increment – i++
- Need equality comparison – i == j (equiv i != j)

Lifting

```
template <typename ForwardIterator, typename T>
T sum(ForwardIterator begin, ForwardIterator end, T init) {
  while (begin != end)
    init += *begin++;
  return init;
}
```

```
double dd = sum(dx, dx + dN, 0.0);
double ff = sum(fx, fx + fN, 0.0);
double ll = sum(lx, nullptr, 0.0);
```

Specialization

```
double dd = sum(dx, dx + dN, 0.0);
double ff = sum(fx, fx + fN, 0.0);
double ll = sum(lx, nullptr, 0.0);
```

```
double sum(double* x, size_t N) {
  double s = 0;
  for (size_t i = 0; i < N; ++i)
    s += x[i];
  return s;
}
```

```
float sum(float* x, size_t N) {
  float s = 0;
  for (size_t i = 0; i < N; ++i)
    s += x[i];
  return s;
}
```

```
double sum(element* x) {
  double s = 0;
  while (x != nullptr) {
    s += x->val;
    x = x->next;
  }
  return s;
}
```

Interface Specification

- Lets formalize what we just did
- What does the interface really consist of?
- Operations supported by the parameterized type
- Other types associated with the parameterized type
- Semantics, complexity guarantees
- This set of requirements is called a *concept*
- A type meeting the requirements of a concept is said to *model* the concept

Concepts in Generic Programming

- Generic programming is sometimes called "programming with concepts"
- Syntax
  - Valid expressions
  - Associated types
- Semantics
- Complexity

Iterator Concepts

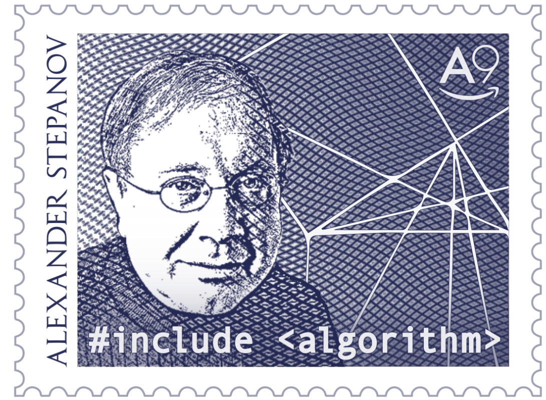
- In our example, the iterator required *, ++, !=
- These are requirements for an InputIterator
- C++ SL has a number of other iterator concepts
- The name of the required concept is usually indicated by the template name

C++ SL Iterator Concepts

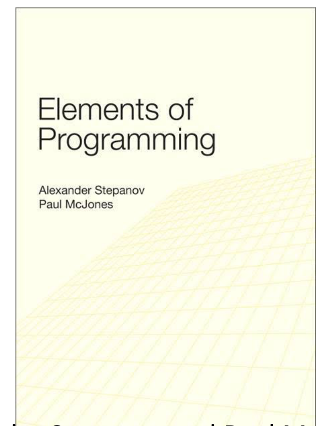
- Trivial Iterator: *
- Input Iterator: *, ++
- Output Iterator: *, ++
- Forward Iterator: *, ++
- Bidirectional Iterator: *, ++, --
- Random Access Iterator: *, ++, --, []

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::for_each`
`std::list`	ForwardIterator	`std::sort`
`std::map`	ReverseIterator	`std::accumulate`
`std::vector`	RandomAccessIterator	`std::min_element`
...		...



Alexander Stepanov and Paul McJones. 2009. *Elements of Programming* (1st ed.). Addison-Wesley Professional.

Generic Programming

- Algorithms are ***generic*** (parametrically polymorphic)
- Algorithms can be used on ***any*** type that meets algorithmic reqts
  - Valid expressions, associated types
  - Not just std. ::types

Standard Library container

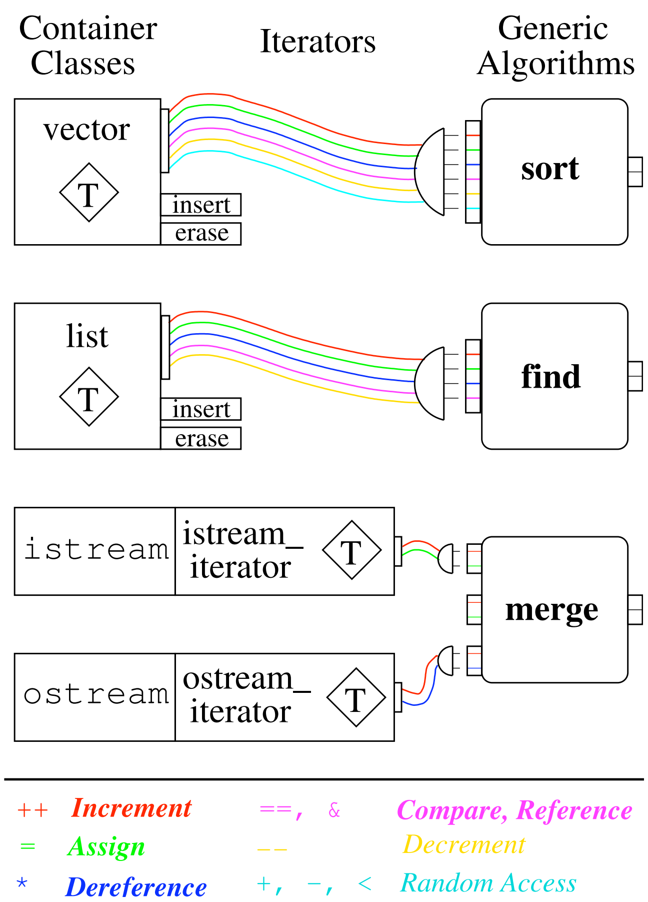
```
vector<double> arrary(N);
 ...
std::accumulate(array.begin(), array.end(), 0.0);
```

iterator

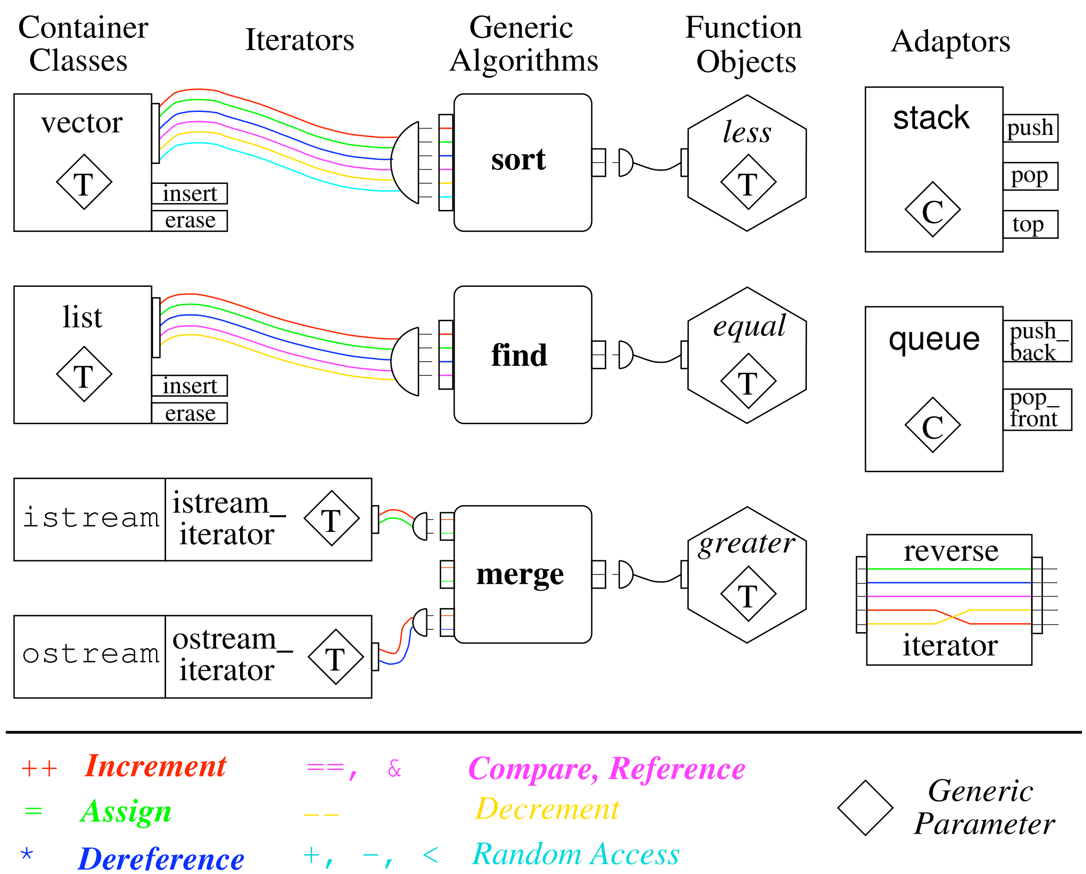
iterator

Initial value

Algorithms and data structures connected by iterators



Five of the six major STL components



std Containers

- Note that all containers have ***same*** interface
- (Actually a hierarchy, we'll come back to this)
- We will primarily be focusing on vector

Headers		<vector>	<deque>	<list>
Members		vector	deque	list
	constructor	vector	deque	list
	operator=	operator=	operator=	operator=
iterators	begin	begin	begin	begin
	end	end	end	end
capacity	size	size	size	size
	max_size	max_size	max_size	max_size
	empty	empty	empty	empty
	resize	resize	resize	resize
element access	front	front	front	front
	back	back	back	back
	operator[]	operator[]	operator[]	
modifiers	insert	insert	insert	insert
	erase	erase	erase	erase
	push_back	push_back	push_back	push_back
	pop_back	pop_back	pop_back	pop_back
	swap	swap	swap	swap

std Containers

- std containers "contain" elements

```
vector<double> array(N);
```

vector of doubles

```
vector<int> array(N);
```

vector of ints

```
list<vector<complex<double> > > thing;
```

list of vectors of complex doubles

- Implementation of list, vector, complex is the same regardless of what is being contained

Generic Programming

- Algorithms are ***generic*** (parametrically polymorphic)
- Algorithms can be used on ***any*** type that meets algorithmic reqts
  - Valid expressions, associated types
  - Not just std. ::types

Standard Library container

```
list<vector<complex<double> > > thing(N);
 ...
std::accumulate(thing.begin(), thing.end(), 0.0);
```

iterator

iterator

Initial value

std Containers

- The std containers are ***class templates*** (not “template classes”)

```
template <typename T> class vector;
template <typename T> class deque;
template <typename T> class list;
```

What follows is a template

The template parameter is a type placeholder

A class template

- Don’t need details for now

```
vector<double>
```

Example

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <numeric>

int main() {

  std::vector<int> x(10);
  std::iota(x.begin(), x.end(), 0);
  std::copy(x.begin(), x.end(),
            std::ostream_iterator<int>(std::cout,"\n"));

  return 0;
}
```

```
$ c++ copy_print_vector.cpp
$ ./a.out
0
1
2
3
4
5
6
7
8
9
```

Class templates

```
template <typename T>
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<T> storage_;
};
```

What is the first rule?

```
Vector<double> av(10);
```

Instantiation model

- Instantiation model— For each template instance, a separate piece of code is generated, and compiled.
- Not directly required by the standard, but all language rules assume it. Used by every C++compiler (except for Lunar).
- Different from Generic Java, Eiffel, ... which compile generic definitions to a single skeleton code.
- Speed/space trade-off: instantiation model often faster, but prone to code bloat.
- Some evidence suggest otherwise (Mark Jones, Haskell dictionary passing).

Unconstrained genericity

```
template <typename ForwardIterator, type
T sum(ForwardIterator begin, ForwardIterator end, T init) {
  while (begin != end)
    init += *begin++;
  return init;
}
```

Even though we have a spec for this

How much of this can we type check (and when)?

```
double dd = sum(dx, dx + dN, 0.0);
double ff = sum(fx, fx + fN, 0.0);
double ll = sum(lx, nullptr, 0.0);
```

For each template instance, a separate piece of code is generated, and compiled

Unconstrained genericity

- Maximal reusability (structural conformance).Concise: no need to express constraints.
- No separate type checking, error diagnostics must be delayed until instantiation.
- Errors may occur deep inside a generic library. Errors difficult to interpret, difficult to assign blame.

Error messages

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <vector>
class A {};

int main() {
  std::vector<A> a;
  // ...
  std::copy(a.begin(), a.end(),
  ↪   std::ostream_iterator<A>(std::cout,
  ↪   "\n"));
}
```

Error messages

```
/usr/include/c++/3.2/bits/ostream.tcc:55: candidates are:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(std::basic_ostream<_CharT,
_Traits>&(*)(std::basic_ostream<_CharT, _Traits>&)) [with _CharT =
char, _Traits = std::char_traits<char>]
/usr/include/c++/3.2/bits/ostream.tcc:77:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(std::basic_ios<_CharT,
_Traits>&(*)(std::basic_ios<_CharT, _Traits>&)) [with _CharT =
char, _Traits = std::char_traits<char>]
/usr/include/c++/3.2/bits/ostream.tcc:99:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(std::ios_base&(*)(std::ios_base&)) [with
_CharT = char, _Traits = std::char_traits<char>]
/usr/include/c++/3.2/bits/ostream.tcc:171:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(long int) [with _CharT = char, _Traits =
std::char_traits<char>] /usr/include/c++/3.2/bits/ostream.tcc:208:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(long unsigned int) [with _CharT = char,
_Traits = std::char_traits<char>]
/usr/include/c++/3.2/bits/ostream.tcc:146:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(bool) [with _CharT = char, _Traits =
std::char_traits<char>] /usr/include/c++/3.2/ostream:104:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(short int) [with _CharT = char, _Traits =
std::char_traits<char>] /usr/include/c++/3.2/ostream:115:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(short unsigned int) [with _CharT = char,
_Traits = std::char_traits<char>] /usr/include/c++/3.2/ostream:119:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(int) [with _CharT = char, _Traits =
std::char_traits<char>] /usr/include/c++/3.2/ostream:130:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(unsigned int) [with _CharT = char, _Traits =
std::char_traits<char>] /usr/include/c++/3.2/bits/ostream.tcc:234:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(long long int) [with _CharT = char, _Traits =
std::char_traits<char>] /usr/include/c++/3.2/bits/ostream.tcc:272:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(long long unsigned int) [with _CharT = char,
_Traits = std::char_traits<char>]
/usr/include/c++/3.2/bits/ostream.tcc:298:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(double) [with _CharT = char, _Traits =
std::char_traits<char>] /usr/include/c++/3.2/ostream:145:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(float) [with _CharT = char, _Traits =
std::char_traits<char>] /usr/include/c++/3.2/bits/ostream.tcc:323:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(long double) [with _CharT = char, _Traits =
std::char_traits<char>] /usr/include/c++/3.2/bits/ostream.tcc:348:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(const void*) [with _CharT = char, _Traits =
std::char_traits<char>] /usr/include/c++/3.2/bits/ostream.tcc:120:
std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
_Traits>::operator<<(std::basic_streambuf<_CharT, _Traits>*) [with
_CharT = char, _Traits = std::char_traits<char>]
/usr/include/c++/3.2/ostream:211: std::basic_ostream<_CharT,
_Traits>& std::operator<<(std::basic_ostream<_CharT, _Traits>&,
char) [with _CharT = char, _Traits = std::char_traits<char>]
/usr/include/c++/3.2/bits/ostream.tcc:500: std::basic_ostream<char,
_Traits>& std::operator<<(std::basic_ostream<char, _Traits>&, char)
[with _Traits = std::char_traits<char>]
/usr/include/c++/3.2/ostream:222: std::basic_ostream<char,
_Traits>& std::operator<<(std::basic_ostream<char, _Traits>&,
signed char) [with _Traits = std::char_traits<char>]
/usr/include/c++/3.2/ostream:227: std::basic_ostream<char,
_Traits>& std::operator<<(std::basic_ostream<char, _Traits>&,
unsigned char) [with _Traits = std::char_traits<char>]
/usr/include/c++/3.2/bits/ostream.tcc:572:
std::basic_ostream<_CharT, _Traits>&
std::operator<<(std::basic_ostream<_CharT, _Traits>&, const char*)
[with _CharT = char, _Traits = std::char_traits<char>]
/usr/include/c++/3.2/bits/ostream.tcc:622: std::basic_ostream<char,
_Traits>& std::operator<<(std::basic_ostream<char, _Traits>&, const
char*) [with _Traits = std::char_traits<char>]
/usr/include/c++/3.2/ostream:246: std::basic_ostream<char,
_Traits>& std::operator<<(std::basic_ostream<char, _Traits>&, const
signed char*) [with _Traits = std::char_traits<char>]
/usr/include/c++/3.2/ostream:251: std::basic_ostream<char,
_Traits>& std::operator<<(std::basic_ostream<char, _Traits>&, const
unsigned char*) [with _Traits = std::char_traits<char>]
```

```cpp
#include <iostream>
#include <algorithm>
#include <iterator>
#include <vector>
class A {};

int main() {
  std::vector<A> a;
  // ...
  std::copy(a.begin(), a.end(),
   ↪ std::ostream_iterator<A>(std::cout,
   ↪ "\n"));
}
```

What is wrong here?

Terminology

- A ***parametrically polymorphic*** function can accept ***any*** types to be bound to its parameters
  - Including ones that don't work
- A function that is ***bounded polymorphic*** only accepts a specific set of types to be bound to its parameters
- Two approaches for conformance
  - Structural conformance: a type satisfies every requirement listed in the concept/base class/type class/signature/…
  - Nominal conformance: in addition to structural conformance, an explicit declaration is required to establish conformance.

C++ Concepts

- Why change templates?
  - Templates enable generic programming in C++
  - Overloading permits natural abstractions
  - Istantiation eliminates cost of abstractions
  - Many successful, generic libraries in C++
- Big problems remain
  - Expression of ideas of generic programming not direct
  - C++ generic libraries can be very brittle
- Goal for concepts:
  - Improve support for generic programming in C++, retaining performance and flexibility of templates

Unconstrained

We use name "InputIt" to hint to programmer that this should be an InputIterator

std::accumulate

Defined in header <numeric>

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );                              (1)

template< class InputIt, class T, class BinaryOperation >
T accumulate( InputIt first, InputIt last, T init,                                 (2)
              BinaryOperation op );
```

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

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

What is this?

Type requirements

- InputIt must meet the requirements of InputIterator.
- T must meet the requirements of CopyAssignable and CopyConstructible.

InputIterator

C++ concepts: InputIterator

An `InputIterator` is an `Iterator` that can read from the pointed-to element. `InputIterators` only guarantee validity for single pass algorithms: once an `InputIterator` `i` has been incremented, all copies of its previous value may be invalidated.

Requirements

The type `It` satisfies `InputIterator` if

- The type `It` satisfies `Iterator`
- The type `It` satisfies `EqualityComparable`

And, given

- `i` and `j`, values of type `It` or `const It`
- `reference`, the type denoted by `std::iterator_traits<It>::reference`
- `value_type`, the type denoted by `std::iterator_traits<It>::value_type`

InputIterator

Expression	Return	Equivalent expression	Notes
`i != j`	contextually convertible to `bool`	`!(i == j)`	**Precondition**: `(i, j)` is in the domain of `==`.
`*i`	`reference`, convertible to `value_type`	If `i == j` and `(i, j)` is in the domain of `==` then this is equivalent to `*j`.	**Precondition**: `i` is dereferenceable. The expression `(void)*i, *i` is equivalent to `*i`.
`i->m`		`(*i).m`	**Precondition**: `i` is dereferenceable.
`++i`	`It&`		**Precondition**: `i` is dereferenceable. **Postcondition**: `i` is dereferenceable or `i` is past-the-end. **Postcondition**: Any copies of the previous value of `i` are no longer required to be either dereferenceable or to be in the domain of `==`.
`(void)i++`		`(void)++i`	
`*i++`	convertible to `value_type`	`value_type x = *i;` `++i;` `return x;`	

Unconstrained

We use name "InputIt" to hint to programmer that this should be an InputIterator

std::accumulate

Defined in header <numeric>

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );                    (1)

template< class InputIt, class T, class BinaryOperation >
T accumulate( InputIt first, InputIt last, T init,                      (2)
              BinaryOperation op );
```

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

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

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.
- `T` must meet the requirements of `CopyAssignable` and `CopyConstructible`.

Iterator Concepts

`Iterator`	general concept to access data within some data structure (concept)
`InputIterator`	iterator that can be used to read data (concept)
`OutputIterator`	iterator that can be used to write data (concept)
`ForwardIterator`	iterator that can be used to read data multiple times (concept)
`BidirectionalIterator`	iterator that can be both incremented and decremented (concept)
`RandomAccessIterator`	iterator that can be advanced in constant time (concept)

Sort

std::sort

Defined in header <algorithm>

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

template< class ExecutionPolicy, class RandomIt >
void sort( ExecutionPolicy&& policy, RandomIt first, RandomIt las

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 );
```

Wait, what's this?

Defaultx

Customizable

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

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

Instead of accumulate

for_each	similar to std::for_each except returns void (function template)
for_each_n	applies a function object to the first n elements of a sequence (function template)
Defined in header <experimental/numeric>	
reduce (parallelism TS)	similar to std::accumulate, except out of order (function template)
exclusive_scan	similar to std::partial_sum, excludes the ith input element from the ith sum (function template)
inclusive_scan	similar to std::partial_sum, includes the ith input element in the ith sum (function template)
transform_reduce (parallelism TS)	applies a functor, then reduces out of order (function template)
transform_exclusive_scan	applies a functor, then calculates exclusive scan (function template)
transform_inclusive_scan	applies a functor, then calculates inclusive scan (function template)

Reduce

std::experimental::parallel::reduce

Defined in header <experimental/numeric>

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

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

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

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

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

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

Notes

- If `policy` is an instance of `sequential_execution_policy`, all operations are performed in the calling thread.
- If `policy` is an instance of `parallel_execution_policy`, operations may be performed in unspecified number of threads, indeterminately sequenced with each other
- If `policy` is an instance of `parallel_vector_execution_policy`, execution may be both parallelized and vectorized: function body boundaries are not respected and user code may be overlapped and combined in arbitrary manner (in particular, this implies that a user-provided Callable must not acquire a mutex to access a shared resource)

Example

```
  { 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

```
{ 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"; }
```

Parallel execution

Parallel execution

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
```

Accumulate

Sequential reduce

Parallel execution

Parallel execution

Example II

```
std::list<double> v(&x(0), &x(x.num_rows()));

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

Example II

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

Results II

```
std::accumulate result 694.824 took 1107 ms
std::reduce result 694.824 took 1997 ms
std::reduce result 694.824 took 2042 ms
std::reduce result 694.824 took 1980 ms
```

Accumulate

Sequential reduce

Parallel execution

Parallel execution

Why no speedup?

Recall

```
for (size_t k = 0; k < parts; ++k) {
  futs.push_back(std::async(std::launch::async, [&, k]()->double {
      double sum = 0.0;
          for (size_t i = k*blocksize; i < (k+1)*blocksize; ++i)
        sum += x(i) * x(i);
      return sum;
  }));
}
```

What is the first access in each thread?

How long should it take to get there?

std::list

Member types

Member type	Definition
value_type	T
allocator_type	Allocator
size_type	Unsigned integer type (usually std::size_t)
difference_type	Signed integer type (usually std::ptrdiff_t)
reference	Allocator::reference (until C++11) value_type& (since C++11)
const_reference	Allocator::const_reference (until C++11) const value_type& (since C++11)
pointer	Allocator::pointer (until C++11) std::allocator_traits<Allocator>::pointer (since C++11)
const_pointer	Allocator::const_pointer (until C++11) std::allocator_traits<Allocator>::const_pointer (since C++11)
iterator	BidirectionalIterator
const_iterator	Constant BidirectionalIterator
reverse_iterator	std::reverse_iterator<iterator>
const_reverse_iterator	std::reverse_iterator<const_iterator>

C++ concepts: **BidirectionalIterator**

A BidirectionalIterator is a ForwardIterator that can be moved in both directions (i.e. incremented and decremented).

std::vector

Member types

Member type	Definition
value_type	T
allocator_type	Allocator
size_type	Unsigned integer type (usually std::size_t)
difference_type	Signed integer type (usually std::ptrdiff_t)
reference	Allocator::reference (until C++11) value_type& (since C++11)
const_reference	Allocator::const_reference (until C++11) const value_type& (since C++11)
pointer	Allocator::pointer (until C+ std::allocator_traits<Allocator>::pointer (since C+
const_pointer	Allocator::const_pointer std::allocator_traits<Allocator>::const_pointer
iterator	RandomAccessIterator
const_iterator	Constant RandomAccessIterator
reverse_iterator	std::reverse_iterator<iterator>
const_reverse_iterator	std::reverse_iterator<const_iterator>

C++ concepts: RandomAccessIterator

A RandomAccessIterator is a BidirectionalIterator that can be moved to point to any element in constant time.

A pointer to an element of an array satisfies all requirements of RandomAccessIterator

Example II

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

Random Access!

Thank you!

Creative Commons BY-NC-SA 4.0 License