NORTHWEST INSTITUTE *for* ADVANCED COMPUTING

AMATH 483/583
High Performance Scientific Computing

Templates and Generic Programming

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

Before

```
class RowMatrix {
public:
  RowMatrix(size_t M, size_t N) : num_rows_(M), num_cols_(N), storage_(num_rows_ * num_cols_) {}

        double& operator()(size_t i, size_t j)       { return storage_[i * num_cols_ + j]; }
  const double& operator()(size_t i, size_t j) const { return storage_[i * num_cols_ + j]; }

  size_t num_rows() const { return num_rows_; }
  size_t num_cols() const { return num_cols_; }

private:
  size_t              num_rows_, num_cols_;
  std::vector<double> storage_;
};
```

After

```
class ColMatrix {
public:
  ColMatrix(size_t M, size_t N) : num_rows_(M), num_cols_(N), storage_(num_rows_ * num_cols_) {}

        double& operator()(size_t i, size_t j)       { return storage_[j * num_rows_ + i]; }
  const double& operator()(size_t i, size_t j) const { return storage_[j * num_rows_ + i]; }

  size_t num_rows() const { return num_rows_; }
  size_t num_cols() const { return num_cols_; }

private:
  size_t              num_rows_, num_cols_;
  std::vector<double> storage_;
};
```

Matrix-Matrix Product

```
Matrix operator*(const Matrix& A, const Matrix& B) {
  Matrix 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;
}
```

```
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-Matrix Product

```
RowMatrix operator*(const RowMatrix& A, const RowMatrix& B) {
  RowMatrix 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;
}
```

Matrix-matrix product is the same for any matrix

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

```
ColMatrix operator*(const ColMatrix& A, const ColMatrix& B) {
  ColMatrix 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;
}
```

Why should we ever write more than one?

What's wrong with writing more than one?

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

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;
}
```

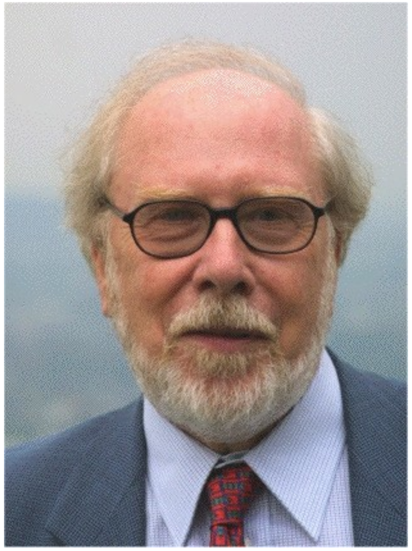
If We Can't Write Binary Search…

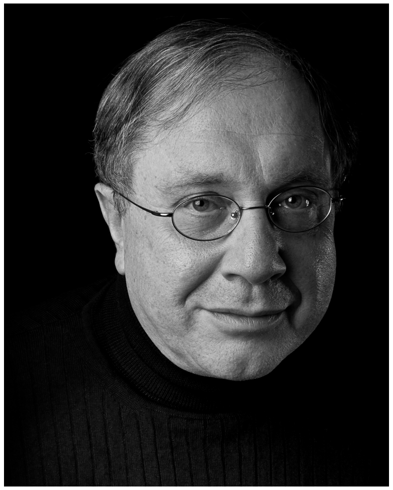
- Jon Bentley's solution is considerably more complicated (and slower)
- Photoshop uses this problem as a take home test for candidates.
  - > 90% of candidates fail.
- Experience teaching algorithms indicate that > 90% of engineers, regardless of experience, cannot write this simple code

- It's about correctness

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.

Lifting

Array of doubles | Beginning | End | Overload (ad-hoc polymorphism)

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

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

Array of floats | Beginning | End | Overload (ad-hoc polymorphism)

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

```
float* fx;
size_t fN;
float  fz = sum(fx, fN);
```

But we need a different implementation for every type

Polymorphism: same function can be called with different types

Lifting

```
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;
}
```

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

float* fx;
size_t fN;
float  fz = sum(fx, fN);
```

Array of Ts

Beginning

End

Parametric polymorphism

Parametric polymorphism

Polymorphism not due to overload, rather parameterized by argument type

Polymorphism: same function can be called with different types

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

After

```
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;
}
```

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;
}
```

The function gets created on instantiation

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

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

This is not a function

It is a function template

A blueprint for a function

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;
}
```

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

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

At this level

At this level

Bind T to be a double

Bind T to be a float

But we know this is a float

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;
}
```

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

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

Bind T to be a double

At this level

Bind T to be a float

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;
}
```

Bind T to be a Vector

```
int main() {

  Vector v(1024);
  double a = sum(v, v.num_rows());

  return 0;
}
```

```
$ c++ vector_sum.cpp
vector_sum.cpp:16:14: error: no matching function for call to 'sum'
  double a = sum(v, v.num_rows());
             ^~~
vector_sum.cpp:4:3: note: candidate template ignored: could not match 'T *' against 'Vector'
T sum (T* x, size_t N) {
  ^
1 error generated.
```

C++ (function) templates

```
int main() {

  Vector v(1024);
  double a = sum(&v[0], v.num_rows());

  return 0;
}
```

Bind T to be a double*

```
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;
}
```

C++ (function) templates

```
struct element {
  double   val;
  element* next;
};

int main() {
  size_t N = 0;
  element* linked_list = null

  // create list
  double a = sum(linked_list,

  return 0;
}
```

Bind T to be an element*

```
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;
}
```

```
$ c++ linked_list_0.cpp
linked_list_0.cpp:22:10: error: no viable conversion from 'element' to 'double'
  double a = sum(linked_list, N);
         ^   ~~~~~~~~~~~~~~~~~~~
linked_list_0.cpp:5:5: error: no viable conversion from 'int' to 'element'
  T s = 0;
    ^   ~
linked_list_0.cpp:22:14: note: in instantiation of function template specialization 'sum<elemen
t>' requested here
  double a = sum(linked_list, N);
             ^
linked_list_0.cpp:12:8: note: candidate constructor (the implicit copy constructor) not viable:
 no known conversion from 'int' to 'const element &' for 1st argument
struct element {
       ^
linked_list_0.cpp:9:10: error: no viable conversion from returned value of type 'int' to functi
on return type 'element'
  return 0;
         ^
linked_list_0.cpp:12:8: note: candidate constructor (the implicit copy constructor) not viable:
 no known conversion from 'int' to 'const element &' for 1st argument
struct element {
       ^
3 errors generated.
```

Requirements for s

We need to be able assign 0 to a T

Pointer to an array of T

Something that can count Ts

But what if I want to sum over a linked list?

```
ate <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;
}
```

We need to index into the array

Perfectly reasonable

R ts for sum so f

We have a thing we want to sum over

Beginning of the thing

End of the thing

```cpp
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

Get value

No less than

Move to next

Check

Check

```
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) : 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;
};
```

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

Move to next

Will s be compatible with “0”

Check

Check

Lifting

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

Rename some things

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;
};
```


WASHINGTON

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;
}
```

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/stream_iterator.h: In member function
   'std::ostream_iterator<_Tp, _CharT, _Traits>& std::ostream_iterator<_Tp,
   _CharT, _Traits>::operator=(const _Tp&) [with _Tp = A, _CharT = char,
   _Traits = std::char_traits<char>]':
/usr/include/c++/3.2/bits/stl_algobase.h:241:
   instantiated from '_OutputIter std::__copy(_RandomAccessIter,
   _RandomAccessIter, _OutputIter, std::random_access_iterator_tag)
   [with _RandomAccessIter = A*, _OutputIter =
   std::ostream_iterator<A, char, std::char_traits<char> >]'
/usr/include/c++/3.2/bits/stl_algobase.h:260: instantiated from
   '_OutputIter std::__copy_aux2(_InputIter, _InputIter, _OutputIter,
   __false_type) [with _InputIter = A*, _OutputIter =
   std::ostream_iterator<A, char, std::char_traits<char> >]'
/usr/include/c++/3.2/bits/stl_algobase.h:303: instantiated from
   '_OutputIter std::__copy_ni2(_InputIter, _InputIter, _OutputIter,
   __false_type) [with _InputIter = A*, _OutputIter =
   std::ostream_iterator<A, char, std::char_traits<char> >]'
/usr/include/c++/3.2/bits/stl_algobase.h:314: instantiated from
   '_OutputIter std::__copy_ni1(_InputIter, _InputIter, _OutputIter,
   __true_type) [with _InputIter = __gnu_cxx::__normal_iterator<A*,
   std::vector<A, std::allocator<A> > >, _OutputIter =
   std::ostream_iterator<A, char, std::char_traits<char> >]'
/usr/include/c++/3.2/bits/stl_algobase.h:349: instantiated from
   '_OutputIter std::copy(_InputIter, _InputIter, _OutputIter) [with
   _InputIter = __gnu_cxx::__normal_iterator<A*, std::vector<A,
   std::allocator<A> > >, _OutputIter = std::ostream_iterator<A, char,
   std::char_traits<char> >]' cpp_templates/cpp_templates.w:1067:
   instantiated from here
/usr/include/c++/3.2/bits/stream_iterator.h:141: no match for '
   std::basic_ostream<char, std::char_traits<char> >& << const A&'
   operator
```

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?

Organizing template files

```
// min.hpp
#ifndef MIN_HPP
#define MIN_HPP
template <class T> T min(const T& a, const T& b);
#endif // MIN_HPP

// min.cpp
#include "min.hpp"
template <class T>
T min(const T& a, const T& b) { return a < b ? a : b; }

// main.cpp
#include "min.hpp"
int main(int, char* []) {
  return min(0, 1);
}
```

Traditional organization

```
$ g++ -c min.cpp
$ g++ -c main.cpp
$ g++ min.o main.o
main.o: In function ‘main’:
main.o(.text+0x2a):
undefined reference to ‘int min<int>(int const&, int const&)’
collect2: ld returned 1 exit status
```

- min<int>has not been instantiated. Why not?

Inclusion model

```
// min.hpp
#ifndef MIN_HPP
#define MIN_HPP
template <class T> T min(const T& a, const T& b) {
  return a < b ? a : b;
}
#endif // MIN_HPP

// main.cpp
#include "min.hpp"
int main(int, char* []) {
  return min(0, 1);
}
```

All source has to be included

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

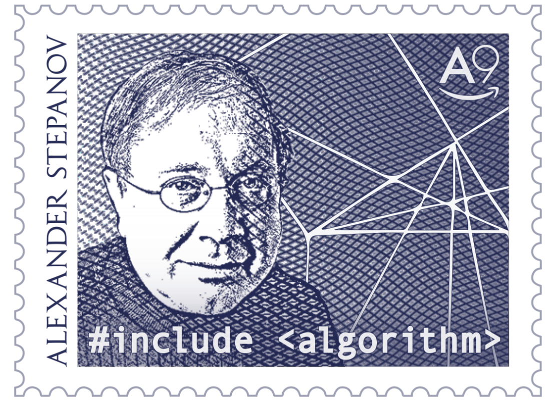
- In our example, the iterator required *, ++, !=
- These are requirements for an InputIterator
- C++ SL has a number of other iterator concepts
- The name of the required concept is usually indicated by the template name

C++ SL Iterator Concepts

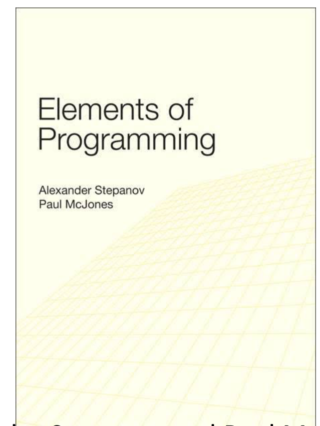
- Trivial Iterator: *
- Input Iterator: *, ++
- Output Iterator: *, ++
- Forward Iterator: *, ++
- Bidirectional Iterator: *, ++, --
- Random Access Iterator: *, ++, --, []

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

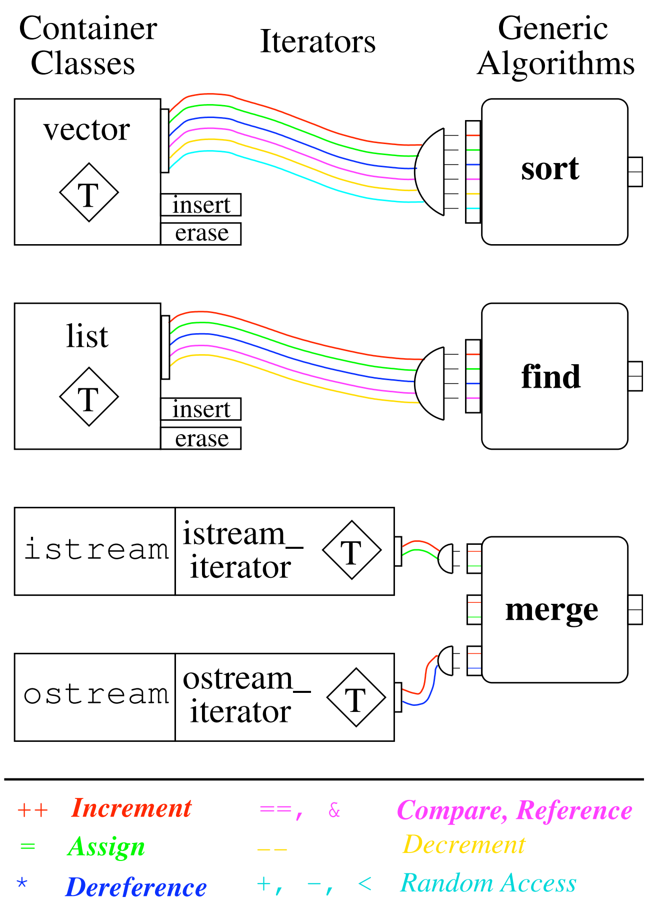
```
vector<double> arrary(N);
 ...
std::accumulate(array.begin(), array.end(), 0.0);
```

iterator

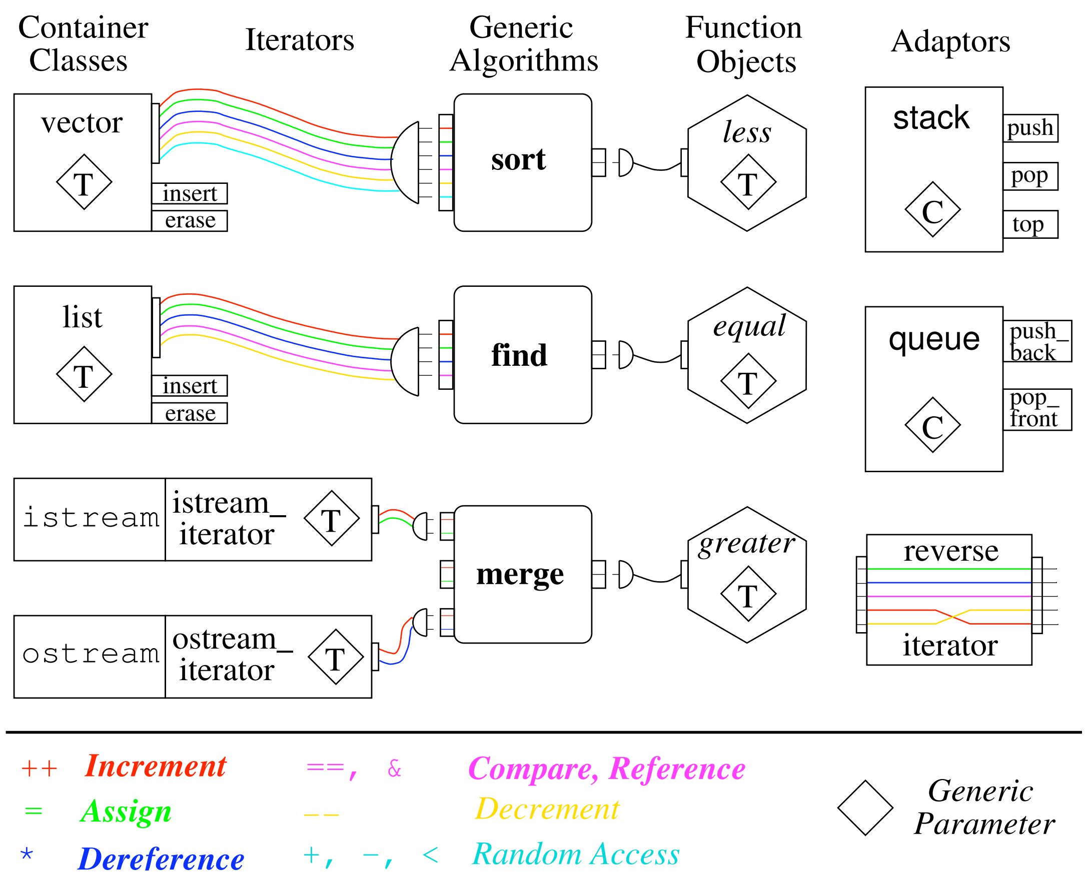
iterator

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<int> array(N);

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

vector of doubles

vector of ints

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

```cpp
#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);
```

Thank you!

Creative Commons BY-NC-SA 4.0 License