Generic Algorithms

generic

One of the biggest advantages of the C++ STL are the *generic algorithms*:

- because every container is ***templated***, each container has potentially many types
- the generic algorithms are designed so that doesn't matter. The algorithms work with any container (mostly ☺)

iterators are the key

Because iterators work with ***any container of any type***, iterators are the key to how the algorithms work.

Each of the algorithms somehow utilizes iterators to perform their task

mostly???

While the algorithms can potentially be used on any container, the type of the container still matters.

Essentially the underlying type, what the iterator points to, dictates operations.

That is the C++ way.

more than 100

There are more than 100 such algorithms, and we can't look at all. However, ***you*** should try to learn them over time. They are very helpful!

I will list ***all*** of them at the end of these slides with a brief overview from the STL bible "The C++ Standard Library, 2nd Edition", Nicolai Josuttis

Helpful tip

Section A.2 (page 870) of the book gives a list of the algorithms and some very helpful, quick summaries of what they do.

Good for later reference!

advantages

- **simple**, reuse of code that does what you want.
- **correct**, proven to work as you expected.
- **efficient**, hard to write loops more efficient than an algorithm
- **clarity**, easier to read and write

Different way to think about prolems

The STL gives you a higher level of abstraction to address your everyday problems. It takes a little getting used to.

For example, you *rarely write loops* in generic algorithms. They loop for you!

Algorithm Categories

- Non-modifying
- Modifying
- Removing (elements)
- Mutating (elements)
  - Sorting (element order changes)
- Operation on sorted collections

accumulate

numeric algorithms
Ex 12.0

accumulate, #include<numeric>

Let's start with the `accumulate` algorithm.

First form:

```
accumulate(begin_itr, end_itr, init)
```

from the value at the beginning iterator up to but not including the value at the end iterator, sum up the values (operator +). The initial value is `init`, and type of `init` sets the type of the return

example

The `accumulate` algorithm "adds", (really applies any binary operator), to the underlying types of the container:

- works for any numeric type and strings
- might not work for others, depends on the type
  - does the underlying type support + as an operation?

examples

```
vector<int>v={1,2,3,4,5};
// prints 15
cout<<accumulate(v.begin(), v.end(), 0);

vector<string>s={"hi","moms"};
// prints "himoms"
cout<<accumulate(s.begin(),
                 s.end(),string(" "));
```


Notes

- no loop needed. Implicitly the algorithm goes through the elements indicated in the half-open range of iterators and performs the operation
- It uses the "+" operator which is overloaded (addition, concatenation)
  - For strings, we need `(" ")` as the initial value. We are working with string objects, not the default C type

change the ranges

```
vector<int>v={1,2,3,4,5};
// [1] through [3], start at 100 → 109
cout<<accumulate(v.begin()+1,
                 v.end()-1,
                 100);
```

Remember, `end()` points to one past the range. `v.end()-1` points to index 4, so iterator goes through 1-3

use a different operation

2nd form allows that you use a different operation than +

- many of the algorithms allow you to enter a function, one predefined or one you make up, to solve some problem

```
accumulate(begin_itr,
           end_itr,
           init, func);
```

Pre-existing

These are templated. They require
`#include<functional>`

These types, listed in Table 14.2, are defined in the `functional` header.

Table 14.2: Library Function Objects

Arithmetic	*Relational*	*Logical*
plus<Type>	equal_to<Type>	logical_and<Type>
minus<Type>	not_equal_to<Type>	logical_or<Type>
multiplies<Type>	greater<Type>	logical_not<Type>
divides<Type>	greater_equal<Type>	
modulus<Type>	less<Type>	
negate<Type>	less_equal<Type>	

Predefined are ***function objects***

More on this later, but essentially the question is:

- `minus<int>()`, why the trailing `()`?
- these are actually objects (in the C++ sense) that respond to the () operator, making it a *function object*

does the following

for the selected function:

```
init = init op element;
```

where op is predefined or provided.

returns the result accumulated in `init`


```
vector<int>v={1,2,3,4,5};
// prints 120
cout<<accumulate(v.begin(),v.end(),1
         ,multiplies<int>())<<endl;

// prints -15
cout<<accumulate(v.begin(),v.end(),
              0,minus<int>())<<endl;
```

Roll your own function

```
template <typename T>
T sum_of_squares(const T &a, const T &b){
  return a + b*b;
}
// prints 55
cout << accumulate(v.begin(), v.end(), 0,
                   sum_of_squares<int>)
```

remember, `init op element` so `init` is first param, `element` is the second.

other in #include<numeric>

Name	Effect
`accumulate()`	Combines all element values (processes sum, product, and so forth)
`inner_product()`	Combines all elements of two ranges
`adjacent_difference()`	Combines each element with its predecessor
`partial_sum()`	Combines each element with all its predecessors

Table 11.9. Numeric Algorithms

lambdas

anonymous functions

Ex 12.1

writing functions is a pain

If you have a simple function you need, say for a generic algorithm, and you aren't going to reuse it, there is a way to do it "simply"

A *lambda* expression is basically an unnamed function that is defined in place.

weird syntax

```
[capture] (params)-> returnType {body};
```

- capture, globals used in function
  - can be empty
- params, parameters of the function
- { body }, the function body
- -> returnType, (optional) if it isn't obvious, what the return type is

the basic lambda

```
auto fn = [] (long l){
                return l*l;
              };
cout << fn(2) << endl;
```

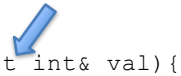
what type is fn? Great question!

only your compiler knows for sure

The type of a lambda is generated by the compiler. auto is kind of essential here.

However, it is a callable object, and where you need a callable object you can use a lambda

```
vector<int> v_i{1,2,3,4,5};
…
cout <<"sum of x+2 is:"
    << accumulate(
          v_i.begin(),
           v_i.end(),
           0,
           [] (const int& tot, const int& val){
                return tot + val + 2;
           }
         )
    << endl;
```



capture list

the capture list allows you to use variables defined (but not passed as args) in the outer global scope

```
long global_l = 23;
auto fun2 = [global_l](long l){
 return global_l + l;
 };
 cout << fun2(23) << endl;
```

copied into the lambda

capture list 2

Or you can use scope by reference. If you don't return, return type is void

```
double global_d = 3.14159;
auto fun3 = [&global_d](double
d) {
  global_d += d;
    };
cout << global_d << endl;
```

by reference global_d changed

why?

So why lambdas? They have use when:

- "close to" their use
- short

If used right, makes it easy to see what is being done, especially in a generic algorithm.

complicated

lambdas are complicated, so we are only covering some basic usage, but even so we will see how convenient they are in generic algorithms.

find, search

non-modifying algorithms
Ex 12.2

find, #include<algorithm>

```
vector<int> v{1,2,3,4,5};
auto mark =find(v.begin(),
                v.end(), 4);
```

Look from beginning to end for target (last arg, here 4). Returns iterator, equal to target or `v.end()` if value not found

the _if names

algorithms whose name ends in `_if` require a condition to be true for their success.

They usually require the user to define a `predicate`, a function that returns a boolean value. It is a measure of some logical condition

find_if

```
bool even (int elem){
   return !(elem % 2); }

vector<int> v{1,2,3,4,5,6};
auto loc=find_if(v.begin(),
                 v.end(),
                 even);
```

Find the first even element.

search

Search looks for an exact subsequence and indicates where the subsequence begins (or end iterator if not found). Search has iterators, so does the target

```
vector<int> v{1,2,3,4,5,6};
vector<int> target{2,3};
auto loc=search(v.begin(), v.end(),
     target.begin(), target.end());
```


Overview

Search for	String Function	STL Algorithm
First occurrence of one element	`find()`	`find()`
Last occurrence of one element	`rfind()`	`find()` with reverse iterators
First occurrence of a subrange	`find()`	`search()`
Last occurrence of a subrange	`rfind()`	`find_end()`
First occurrence of several elements	`find_first_of()`	`find_first_of()`
Last occurrence of several elements	`find_last_of()`	`find_first_of()` with reverse iterators
First occurrence of *n* consecutive elements		`search_n()`

Table 11.2. Comparison of Searching String Operations and Algorithms

other non modify algm

Name	Effect
`for_each()`	Performs an operation for each element
`count()`	Returns the number of elements
`count_if()`	Returns the number of elements that match a criterion
`min_element()`	Returns the element with the smallest value
`max_element()`	Returns the element with the largest value
`minmax_element()`	Returns the elements with the smallest and largest value (since C++11)
`find()`	Searches for the first element with the passed value
`find_if()`	Searches for the first element that matches a criterion
`find_if_not()`	Searches for the first element that matches a criterion not (since C++11)
`search_n()`	Searches for the first *n* consecutive elements with certain properties
`search()`	Searches for the first occurrence of a subrange
`find_end()`	Searches for the last occurrence of a subrange
`find_first_of()`	Searches the first of several possible elements
`adjacent_find()`	Searches for two adjacent elements that are equal (by some criterion)
`equal()`	Returns whether two ranges are equal
`is_permutation()`	Returns whether two unordered ranges contain equal elements (since C++11)

other non modify algm

`mismatch()`	Returns the first elements of two sequences that differ
`lexicographical_compare()`	Returns whether a range is lexicographically less than another range
`is_sorted()`	Returns whether the elements in a range are sorted (since C++11)
`is_sorted_until()`	Returns the first unsorted element in a range (since C++11)
`is_partitioned()`	Returns whether the elements in a range are partitioned in two groups according to a criterion (since C++11)
`partition_point()`	Returns the partitioning element for a range partitioned into elements fulfilling and elements not fulfilling a predicate (since C++11)
`is_heap()`	Returns whether the elements in a range are sorted as a heap (since C++11)
`is_heap_until()`	Returns the first element in a range not sorted as a heap (since C++11)
`all_of()`	Returns whether all elements match a criterion (since C++11)
`any_of()`	Returns whether at least one element matches a criterion (since C++11)
`none_of()`	Returns whether none of the elements matches a criterion (since C++11)

copy transform

modifying algorithms
Ex 12.3

copy #include<algorithm>

Copy is one of the most useful algorithms, but its first form, no so much. Must guarantee there is room in the designation (yuck)!

```
vector<int>v{1,2,3,4,5};
vector<int>t(10,1);
copy(v.begin(), v.end(), t.begin());
```

`t` is size 10, overwrites `t` index 0-4 with contents of `v`.

copy_if

Like other _if algorithms, only copies of predicate is true.

iterator adaptors

using copy with special iterators
Ex 12.3

copy requirements

It is a bit of a problem when copy requires that we have a target big enough to hold what we are copying.

That is the point isn't it? We can to copy regardless of size.

special iterators #include<iterator>

Two special kind of iterators that get around this issue:

- insert iterators
- stream iterators

insert iterators

Each container works best with certain kinds of insert operators

- vectors insert at the back
- deque insert at the back or front
- lists, sets insert at a particular position

#include<iterator> back_inserter

```
vector<string>v_s{"a","b","c"};
vector<string>t; // empty t
copy(v_s.begin(), v_s.end(),
     back_inserter(t));
```

append each element of `v_s` to the end of `t` . `t` started empty, grew to size 3

ostream_iterator

Can connect an iterator to a stream. Most useful is the `ostream_iterator`. Two args, the stream and what separates each element.

- separator is a string, not a char
- requires a ***template*** of the type being output

easy output

```
vector<int>v{1,2,3,4,5};
ostream_iterator out<int>(cout,",");
copy(v.begin(), v.end(), out);
```

Prints the contents of the vector. So easy! Note you can hook it to a `ofstream` or an `ostringstream`.

transform

```
char upper(char ch){
  return toupper(ch);
}
vector<char>c{'a', 'b', 'c'};
vector<string>t;
transform(c.begin(), c.end(),
          back_inserter(t), upper);
```

upper case chars in c, put in t

modify algms

Name	Effect
`for_each()`	Performs an operation for each element
`copy()`	Copies a range starting with the first element
`copy_if()`	Copies elements that match a criterion (since C++11)
`copy_n()`	Copies *n* elements (since C++11)
`copy_backward()`	Copies a range starting with the last element
`move()`	Moves elements of a range starting with the first element (since C++11)
`move_backward()`	Moves elements of a range starting with the last element (since C++11)
`transform()`	Modifies (and copies) elements; combines elements of two ranges
`merge()`	Merges two ranges
`swap_ranges()`	Swaps elements of two ranges
`fill()`	Replaces each element with a given value
`fill_n()`	Replaces *n* elements with a given value
`generate()`	Replaces each element with the result of an operation
`generate_n()`	Replaces *n* elements with the result of an operation
`iota()`	Replaces each element with a sequence of incremented values (since C++11)
`replace()`	Replaces elements that have a special value with another value
`replace_if()`	Replaces elements that match a criterion with another value
`replace_copy()`	Replaces elements that have a special value while copying the whole range
`replace_copy_if()`	Replaces elements that match a criterion while copying the whole range

sort

sort algorithms and algorithms that depend on sorted containers.
Ex 12.4

sort #include<algorithm>

```
vector<string>s{"this","is","a","test"};
sort(s.begin(), s.end());
```

sorts the container (from iterator to iterator) and changes the order of the elements in the container.

- depends on a < (less than) operator for the elements

add your own compare

You can add your own function that returns a boolean and runs as a less-than (<) operator.

Sort will occur on that. If you define a class that has the < operator, it will sort class elements based on that.

sort algorithms

Name	Effect
`sort()`	Sorts all elements
`stable_sort()`	Sorts while preserving order of equal elements
`partial_sort()`	Sorts until the first *n* elements are correct
`partial_sort_copy()`	Copies elements in sorted order
`nth_element()`	Sorts according to the *n*th position
`partition()`	Changes the order of the elements so that elements that match a criterion are at the beginning
`stable_partition()`	Same as `partition()` but preserves the relative order of matching and nonmatching elements
`partition_copy()`	Copies the elements while changing the order so that elements that match a criterion are at the beginning
`make_heap()`	Converts a range into a heap
`push_heap()`	Adds an element to a heap
`pop_heap()`	Removes an element from a heap
`sort_heap()`	Sorts the heap (it is no longer a heap after the call)

make vector, sort with lambda

```
vector<pair<string,int>> v;
copy(dict.begin(), dict.end(), back_inserter(v));
sort(v.begin(), v.end(),
    [ ] (const pair<string,int> &p1, const pair<string,int> &p2)
      {return p1.second > p2.second;});
```

push back each pair onto a vector

sort using a > criteria, and a lambda

depends on sorted container

Name	Effect
binary_search()	Returns whether the range contains an element
includes()	Returns whether each element of a range is also an element of another range
lower_bound()	Finds the first element greater than or equal to a given value
upper_bound()	Finds the first element greater than a given value
equal_range()	Returns the range of elements equal to a given value
merge()	Merges the elements of two ranges
set_union()	Processes the sorted union of two ranges
set_intersection()	Processes the sorted intersection of two ranges
set_difference()	Processes a sorted range that contains all elements of a range that are not part of another range
set_symmetric_difference()	Processes a sorted range that contains all elements that are in exactly one of two ranges
inplace_merge()	Merges two consecutive sorted ranges
partition_point()	Returns the partitioning element for a range partitioned into elements fulfilling and elements not fulfilling a predicate (since C++11)

invalid iterators

If an algorithm (or you) substantially moves stuff around in your container, then an existing iterator may be made invalid!

- if you grow a vector
- if sort a vector