Generic Algorithms

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 😃)

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

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.
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 problems**

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

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

```cpp
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

```cpp
vector<int>v={1,2,3,4,5};
// [1] through [3], start at 100 \( \rightarrow \) 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

2\textsuperscript{nd} 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

```cpp
accumulate(begin_itr,
    end_itr,
    init, func);
```
These are templated. They require
\begin{verbatim}
#include<functional>
\end{verbatim}
Pre-existing

More on this later, but essentially the question is:
\begin{itemize}
\item \texttt{minus\langle int\rangle()}, why the trailing ()?
\item these are actually objects (in the C++ sense) that respond to the () operator, making it a \textit{function object}
\end{itemize}

\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
Arithmetic & Relational & Logical \\
\hline
\texttt{plus\langle Type\rangle} & \texttt{equal\_to\langle Type\rangle} & \texttt{logical\_and\langle Type\rangle} \\
\texttt{minus\langle Type\rangle} & \texttt{not\_equal\_to\langle Type\rangle} & \texttt{logical\_or\langle Type\rangle} \\
\texttt{multiplies\langle Type\rangle} & \texttt{greater\_Type} & \texttt{logical\_sr\langle Type\rangle} \\
\texttt{divides\langle Type\rangle} & \texttt{greater\_equal\langle Type\rangle} & \texttt{logical\_not\langle Type\rangle} \\
\texttt{modulus\langle Type\rangle} & \texttt{less\langle Type\rangle} & \\
\texttt{negate\langle Type\rangle} & \texttt{less\_equal\langle Type\rangle} & \\
\hline
\end{tabular}
\caption{Library Function Objects}
\end{table}

for the selected function:
\begin{verbatim}
init = init op element;
\end{verbatim}
\begin{verbatim}
where op is predefined or provided.
\end{verbatim}

\begin{verbatim}
returns the result accumulated in init
\end{verbatim}

\begin{verbatim}
\end{verbatim}

\begin{verbatim}
vector\langle int\rangle v={1,2,3,4,5};
// prints 120
cout<<accumulate(v.begin(),v.end(),1
 ,multiplies\langle int\rangle())<<endl;
\end{verbatim}
\begin{verbatim}
// prints -15
cout<<accumulate(v.begin(),v.end(),
 0,minus\langle int\rangle())<<endl;
\end{verbatim}
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>

<table>
<thead>
<tr>
<th>Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumulate()</td>
<td>Combines all element values (processes sum, product, and so forth)</td>
</tr>
<tr>
<td>inner_product()</td>
<td>Combines all elements of two ranges</td>
</tr>
<tr>
<td>adjacent_difference()</td>
<td>Combines each element with its predecessor</td>
</tr>
<tr>
<td>partial_sum()</td>
<td>Combines each element with all its predecessors</td>
</tr>
</tbody>
</table>

Table 11.9. Numeric Algorithms

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.

lambdas
anonymous functions
Ex 12.1
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

```cpp
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;
```

lambda instead of a function
The capture list allows you to use variables defined (but not passed as args) in the outer global scope:

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

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

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

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.
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 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());`
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)!

```cpp
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
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

```cpp
class ostream_iterator {
public:
    template<typename charT, typename Traits, typename Container, typename State = void>
    ostream_iterator(ostream & os, charT delimiter = ' ') : os_(os), delimiter_(delimiter) {
        State = 0;
    }
    ostream_iterator & operator= (charT ch) {
        State = ch - '0';
        return *this;
    }

    void operator++ () {
        if (State == 0)
            os_ << delimiter_;
        State = 0;
    }

    template<typename T>
    ostream_iterator & operator= (const T & value) {
        os_ << value;
        return *this;
    }

private:
    ostream & os_;
    charT delimiter_;
};
```

```cpp
vector<int>v{1, 2, 3, 4, 5};
oststream_iterator out<int>(cout, " ");
for (auto i : v) {
    out << i;
}
```

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

```cpp
char upper(char ch) {
    return toupper(ch);
}
```

```cpp
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
```
sort

sort algorithms and algorithms that depend on sorted containers.

Ex 12.4

```cpp
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.

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

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