# 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 it 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, why the iterator points to, dictates operations
- This is the C++ way

## More than 100

- There are more than 100 such algorithms, and we can’t look at them all.
- **You** should try to learn them over time.
- They are very helpful

# Helpful Tip

- Section A.2 (page 870) of the book give 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**: proved to work as you expect
- **efficient**: hard to write loops more efficient than an algorithm
- **clarity**: easier to read and write

## Different way to think about problems

- The STL give 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
- Example 13.1

## 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 the 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
  - Work 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>`
- See Table 14.2

| 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`

# Examples

```
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 the first param, `element` is the second

## Others in #include<numeric>

- `accumulate()` : Combines all element values
- `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

# Lambdas

## Lambdas

- Anonymous functions
- Example 13.1 (end)

## 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

## Lambda 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
- It is a callable object and where you need a callable object you can use a lambda

## Example

```
vector<int> v {1, 2, 3, 4, 5};
cout << “sum of x+2 is:”
    << accumulate(v.begin(), v.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 fn2 = [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;
};
fun3(1.0);
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

# More Generic Algorithms

# find, search

- non-modifying algorithms
- Ex 13.2

## 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 (here 4).
  - If found, return the iterator pointing to target
  - If not found, returns `v.end()`

## 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);
```

- Finds 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 for 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());
```

## Some other non-modifying algorithms

- `for_each()` : Perform an operation for each element
- `count()` / `count_if()` : Returns the number of elements
- `min_element()` : Returns the smallest element
- `max_element()` : Returns the biggest element
- `equal()` : Returns whether two ranges are equal
- `all_of()` : Returns whether all elements match a criterion
- `any_of()` : Returns whether at least one element matches a criterion
- `none_of()` : Returns whether no elements match a criterion

# Copy transform

- Modifying algorithms
- Ex 13.3

# 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 destination ☹️

```
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 if predicate is true

## iterator adaptors

- using copy with special iterators
- Example 13.3

## 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 copy regardless of size.

## Special iterators #include<iterator>

- Two special kinds of iterators that get around this issue
  - insert iterators
  - stream iterators

## insert iterators

- Each container works best with certain kinds of insert operators
  - `vector` : 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;
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 `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<int> out(cout, “,”);
copy(v.begin(), v.end(), out);
```

- Prints the contents of a 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);
```

- Uppercase chars in c, put in t

## More modifying algorithms

- `copy()` : Copies a range starting with the first element
- `move()` : Moves elements of a range starting with the first element
- `transform()` : Modifies (and copies) elements
- `merge()` : Merges two ranges
- `fill()` : Replaces each element with a given value
- `generate()` : Replaces each element with the result of an operation
- `replace()` : Replaces elements that have a special value with another value

## sort

- sort algorithms and algorithms that depend on sorted containers
- Example 13.4

# sort #include <algorithm>

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

- Sort 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

## Sorting with a lambda

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

- Push back each pair onto a vector
- Sort in reverse order of the second item in each pair

## Sort algorithms

- `sort()` : Sorts all elements
- `stable_sort()` : Sorts while preserving order of equal elements
- `partial_sort()` : Sorts until the first n elements are correct
- `nth_element()` : Sorts according to the nth position
- `partition()` : Changes the order so that elements that match a criterion are at the beginning

## Algorithms that use a sorted container

- `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 the given value
- `merge()` : Merges the elements of two ranges

## Warning about Invalid Iterators

- If an algorithm (or you) substantially moves stuff around in your container then any existing iterators may be made invalid
  - If you grow a vector
  - If you sort a vector