STL: Algorithms And Functors

Miro Jurišić

meeroh@meeroh.org

Introduction

- Algorithms manipulate and inspect STL containers
- Function objects (functors) are objects which behave like functions
- Predicates are stateless functors returning bool
- Algorithms use functors

Overview of standard algorithms

- Non-modifying: `for_each`, `find`, `count`, `equal`, `search`, `binary_search`
- Modifying: `copy`, `swap`, `transform`, `replace`, `fill`, `generate`, `remove`, `unique`, `reverse`, `rotate`, `random_shuffle`
- Sorting: `partition`, `sort`, `merge`, set operations, heap operations, `mix`, `max`
- Other: permutations, `accumulate`, `inner_product`, `partial_sum`, `adjacent_difference`

Algorithm Example

```
vector <int>         v (
    istream_iterator <int>(cin),
    istream_iterator <int>());

cout << "Number of zeros: " << count (v.begin (), v.end (), 0) << endl;
```

Functor Example

```
vector <int>        v1 (
    istream_iterator <int>(cin),
    istream_iterator <int>());

vector <int>        v2;

transform (
    v1.begin (),
    v1.end (),
    back_insert_iterator <vector <int> > (v2.begin ()),
    negate);
```

Algorithms: Non-Modifying

- `for_each` (start, end, function)
- `find` (start, end, value — pred)
- `find_first_of` (start, end, values start, values end, [pred])
- `adjacent_find` (start, end, [pred])
- `count` (start, end, value)
- `count_if` (start, end, pred)
- `mismatch` (start 1, end 1, start 2, [pred])
- `equal` (start 1, end 1, start 2, [pred])
- `search` (start 1, end 1, start 2, end 2, [pred])
- `find_end` (start 1, end 1, start 2, end 2, [pred])
- `search_n` (start, end, count, value, [pred])

Algorithms: Modifying

- `fill` (output start, output end, value)
- `fill_n` (output start, output count, value)
- `generate` (output start, output end, generator)
- `generate_n` (output start, output count, generator)
- `copy` (input start, input en, output start)
- `copy_backward` (input start, input end, output start)
- `swap` (item 1, item 2)
- `iter_swap` (iter 1, iter 2)
- `swap_ranges` (start 1, end 1, start 2)
- `replace` (start, end, old, new)
- `replace_if` (start, end, pred, new)
- `replace_copy` (input start, input end, output start, old, new)
- `replace_copy_if` (input start, input end, output start, pred, new)
- `unique` (start, end)
- `unique_copy` (input start, input end, output start)

- `reverse` (start, end)
- `reverse_copy` (input start, input end, output)
- `rotate` (start, middle, end)
- `rotate_copy` (input start, input middle, input end, output start)
- `random_shuffle` (start, end, [generator])
- `transform` (input start, input end, output start, operator)
- `transform` (input 1 start, input 1 end, input 2 start, output start, operator)

Algorithms: Sorting

- `partition` (start, end, compare)
- `stable_partition` (start, end, compare)
- `sort` (first, last, [compare])
- `stable_sort` (first, last, [compare])
- `partial_sort` (first, middle, last, [compare])
- `partial_sort_copy` (input first, input last, output first, ouput last, [compare])
- nthelement (start, middle, end, [compare])

Algorithms: Binary search

- `lower_bound` (start, end, value, [compare])
- `upper_bound` (start, end, value, [compare])
- `equal_range` (start, end, value, [compare])
- `binary_search` (start, end, value, [compare])

Algorithms: Sorted ranges

- `merge` (input 1 start, input 1 end, input 2 start, input 2 end, output start)
- `inplace_merge` (start, middle, end)
- `includes` (input 1 start, input 1 end, input 2 start, input 2 end, [compare])
- `set_union` (input 1 start, input 1 end, input 2 start, input 2 end, output start, [compare])
- `set_intersection` (input 1 start, input 1e nd, input 2 start, input 2 end, output start, [compare])
- `set_difference` (input 1 start, input 1 end, input 2 start, input 2 end, output start, [compare])
- `set_symmetric_difference` (input 1 start, input 1 end, input 2 start, input 2 end, output start, [compare])

Algorithms: Comparisons

- `min` (item1, item2)
- `max` (item1, item2)
- `min_element` (start, end)
- `max_element` (start, end)
- `lexicographical_compare` (input 1 start, input 1 end, input 2 start, input 2 end, [compare])

Algorithms: Numerics

- `accumulate` (start, end, init, [op])
- `inner_product` (input 1 start, input 1 end, input 2 start, init, [op1, op2])
- `partial_sum` (input start, input end, output start, [op])
- `adjacent_difference` (input start, input end, output start, [op])

Algorithms: Algorithm gotchas

- Don't overrun your output
- Make sure start and end are not reversed

Algorithms: Choosing your algorithm

- Choose the algorithm that does the least work
- Example: `partition` $<$ `stable_partition` $<$ `sort` $<$ `stable_sort`

Functors: basics

- Classes which have an operator () can be used as functions
- More powerful than plain functions, because they can store additional data

```cpp
class EqualToInt:
    public std::unary_function <bool, int> {

    public:
        EqualToInt (int inCompareTo):
            mCompareTo (inCompareTo)
        {
        }

        bool operator () (int inCompare)
        {
            return mCompareTo == inCompare;
        }

    private:
        int mCompareTo;
};
```

Functors: standard

- plus, minus, multiplies, divides, modulus, negate (unary)
- equal_to, not_equal_to, greater, less, greater_equal, less_equal
- logical_and, logical_or, logical_not (unary)

Functors: binders

- Binders convert 2-argument functors to 1-argument functors
- `bind1st` binds to the first argument
- `bind2st` binds to the second argument

```
// Find x, x > 2
find_if (
    v.begin (),
    v.end (),
    bind2nd (greater <int> (), 2));

// Find x, x <= 3
find_if (
    v.begin (),
    v.end (),
    bind2nd (less_equal <int> (), 3));
```

Functors: adaptors

- Adaptors adapt real functions into functors
- `ptr_fun` converts a pointer to a function to a functor
- `mem_fun` converts a pointer to a member function to a functor whose first argument is a pointer
- `mem_fun` converts a pointer to a member function to a functor whose first argument is a reference

```
class Object {
    public:
        int function (int);
};


vector <Object*>        v1;
vector <int>            v2;
vector <int>            v3;

transform (v1.begin (), v1.end (), v2.begin (), v3.begin (),
    mem_fun (&Object::function));
```