

STL Algorithms - How to use them; how to write your own

Marshall Clow
Qualcomm, Inc.

CppCon 2016

mclow@qti.qualcomm.com

@mclow

What is an
STL Algorithm?

Algorithms

- A templated function
- It does something useful
 - Usually in a generic way
- Has a useful name

Why use STL Algorithms?

- They're tested and debugged
- Basic Building Blocks
 - Easier to write code using them
 - Easier to debug code using them
 - Easier to review/revise later


```
std::vector<int> v{0,1,3,5,7,9,2,4,6,8};
bool flag = true;
for (int i = 1; (i <= v.size()) && flag; i++) {
    flag = false;
    for (int j = 0; j < (v.size() - 1); j++) {
        if (v[j+1] < v[j]) {
            std::swap(v[j], v[j+1]);
            flag = true;
        }
    }
}
for (int i:v) std::cout << i << " ";
```

```
std::vector<int> v{0,1,3,5,7,9,2,4,6,8};
std::sort(v.begin(), v.end());
for (int i:v) std::cout << i << " ";
```


Examples of Algorithms

std::min

```
template <typename T>  
const T&  
min(const T& a, const T& b)  
{  
    return a < b ? a : b;  
}
```


std::copy

```
template <typename IIter, typename OIter>
OIter copy(IIter first, IIter last, OIter res)
{
    for (; first != last; ++first, ++res)
        *res = *first;
    return res;
}
```


Let's make our own!

adjacent_pair

Similar to `std::for_each`, but calls the functor on each adjacent pair of values in the sequence.

```
template <typename Iter, typename Func>  
void adjacent_pair(Iter first, Iter last, Func f);
```


adjacent_pair

```
template <typename FwIter, typename Func>
void adjacent_pair(FwIter first, FwIter last, Func f)
{
    if (first != last)
    {
        FwIter trailer = first;
        ++first;
        for (; first != last; ++first, ++trailer)
            f(*trailer, *first);
    }
}
```


What if we wanted all the
pairs, not just the
adjacent ones?

for_all_pairs

```
template <typename FwIter, typename Func>
void for_all_pairs(FwIter first, FwIter last, Func f)
{
    if (first != last)
    {
        FwIter trailer = first;
        ++first;
        for (; first != last; ++first, ++trailer)
            for (FwIter it = first; it != last; ++it)
                f(*trailer, *it);
    }
}
```


More Algorithms

copy_while

```
template<typename InIter, typename OutIter, typename Pred>
std::pair<InIter, OutIter>
copy_while(InIter first, InIter last, OutIter result, Pred p)
{
    for (; first != last && p(*first); ++first, ++result)
        *result = *first;
    return std::make_pair(first, result);
}
```


split

```
template <typename InIter, typename T, typename Func>
void split(InIter first, InIter last, const T &t, Func f)
{
    while (true)
    {
        InIter found = std::find(first, last, t);
        f(first, found);
        if (found == last)
            break;
        first = ++found;
    }
}
```


Writing your own

- Write what you need
- Should be general, but not necessarily universal
- Stepwise refinement

Tips

- Handle degenerate cases
- Always be aware of the operations you're using
- Worry about complexity
- Think about iterator categories
 - Sometimes you will want to have completely different implementations depending on the type of iterator you get (`std::find_end`, for example)

adjacent_pair

```
template <typename FwIter, typename Func>
void adjacent_pair(FwIter first, FwIter last, Func f)
{
    if (first != last)
    {
        FwIter trailer = first;
        ++first;
        for (; first != last; ++first, ++trailer)
            f(*trailer, *first);
    }
}
```


adjacent_pair (revised)

```
template <typename InIter, typename Func>
void adjacent_pair(InIter first, InIter last, Func f)
{
    if (first != last)
    {
        ??? trailer = *first; // What type is this?
        ++first;
        for (; first != last; ++first)
        {
            f(trailer, *first);
            trailer = *first;
        }
    }
}
```


adjacent_pair (revised)

```
template <typename InIter, typename Func>
void adjacent_pair(InIter first, InIter last, Func f)
{
    if (first != last)
    {
        typename std::iterator_traits<InIter>::value_type
trailer = *first;
        ++first;
        for (; first != last; ++first)
        {
            f(trailer, *first);
            trailer = *first;
        }
    }
}
```


adjacent_pair (revised)

```
template <typename InIter, typename Func>
void adjacent_pair(InIter first, InIter last, Func f)
{
    if (first != last)
    {
        auto trailer = *first;
        ++first;
        for (; first != last; ++first)
        {
            f(trailer, *first);
            trailer = *first;
        }
    }
}
```


How to choose an implementation?

- `iterator_traits<Iter>::iterator_category` tells you what kind of iterator you have.
- There is an inheritance relationship between the categories
 - You can add template specializations based on that
 - You can add a parameter of a particular type and let the compiler call the right version.

Dispatch on iterator category

```
template <typename FWIter, typename Func>
void adjacent_pair_impl(FWIter first, FWIter last, Func f,
std::forward_iterator_tag); // Forward iterators
```

```
template <typename InIter, typename Func>
void adjacent_pair_impl(InIter first, InIter last, Func f,
std::input_iterator_tag); // Input iterators
```

```
template <typename Iterator, typename Func>
void adjacent_pair(Iterator first, Iterator last, Func f)
{
    return adjacent_pair_impl(first, last, f,
        typename
std::iterator_traits<Iterator>::iterator_category());
}
```


Questions?

Thank you