C++ algorithms 1

Richel Bilderbeek

Algorithms

- Looped operations that have a name (approximately 80)
- Work on containers
- Work with lambda expressions

Why algorithms?

- More expressive
- Less error prone
- Can call these locally (although syntax is sometimes cumbersome)
- (run-time speed)
- Prefer algorithms over raw for-loops [Bjarne Stroustrup. The C++ Programming Language (3rd edition). Chapter 18.12.1][Scott Meyers. Effective STL. Item 43]

Example 0: for-loops

```
template <class T>
void f(std::vector<T>& v)
{
   const int size{static_cast<int>(v.size())};
   for(int i=0; i!=size; ++i) {
     ++v[i];
   }
}
```

Example 0: algorithm

```
template <class C>
void f(C& v)
{
  std::for_each(
    std::begin(v),
    std::end(v),
    [](auto& i) { ++i; }
  );
}
```

- Delegated the for loop to `for_each`
- Can be written and tweaked locally
- More extensible: can also use `f` for other containers
- Probably faster (confirmed by my benchmark)

Example 0: ranged for

```
template <class C>
void f(C& v)
{
   for (auto& i: v) ++i;
}
```

- As local and tweakable
- May be a case pro a ranged for?
- No clear coding standards on this

Example 1: for-loops

```
template <class T>
void f(std::vector<T>& v)
{
  const int size{static_cast<int>(v.size())};
  for(int i=0; i!=size-1; ++i) {
    for(int j=0; j!=size-i-1; ++j) {
      if(v[j] > v[j+1]) {
        std::swap(v[j],v[j+1]);
      }
    }
  }
}
```

Example 1: for-loops

```
template <class T>
void f(std::vector<T>& v)
{
  const int size{static_cast<int>(v.size())};
  for(int i=0; i!=size-1; ++i) {
    for(int j=0; j!=size-i-1; ++j) {
      if(v[j] > v[j+1]) {
        std::swap(v[j],v[j+1]);
      }
    }
  }
}
```

- Bubble-sort, average complexity O(n^2)

Example 1: algorithm

```
template <class T>
void f(std::vector<T>& v)
{
   std::sort(std::begin(v),std::end(v));
}
```

- Quicksort, average complexity O(n*log(n))

Example 1: algorithm

- Can sort with a custom operator< supplied as a lambda expression

```
void f(std::vector<std::string>& v)
{
  std::sort(
    std::begin(v),
    std::end(v),
    [](const auto& lhs, const auto& rhs) {
      return lhs.size() < rhs.size()
    }
  );
}
```

Example 1: conclusions

- std::sort is more expressive
- std::sort is implemented smarter
- No need to write a sort function, can do it locally
- Can sort in any way
- There are multiple sort algorithms in the STL, e.g. `std::stable_sort`, `std::partial_sort`, `std::nth_element`

Example 2: raw for

```
template <class C, class D>
void f(C& v, const D& w)
{
  assert(v.size() >= w.size());
  const int sz{static_cast<int>(w.size())};
  for (int i=0; i!=sz; ++i) {
    v[i] = w[i];
  }
}
```

Example 2: raw for

```
template <class C, class D>
void f(C& v, const D& w)
{
  assert(v.size() >= w.size());
  const int sz{static_cast<int>(w.size())};
  for (int i=0; i!=sz; ++i) {
    v[i] = w[i];
  }
}
```

- A copy operation

Example 2: algorithm

```
template <class C, class D>
void f(C& v, const D& w)
{
  assert(v.size() >= w.size());
  std::copy(
    std::begin(w),
    std::end(w),
    std::begin(v)
  );
}
```

- Overwrites v

Example 2: algorithm that appends

```
template <class T, class D>
void f(std::vector<T>& v, const D& w)
{
  std::copy(
    std::begin(w),
    std::end(w),
    std::back_inserter(v)
  );
}
```

- Appends to `v` using `push_back`
- Use `inserter` to call `insert`
- To transform the values, use `std::transform`

Example 2: algorithm that copies to stream

```
template <class C>
void f(const C& v)
{
  std::copy(
    std::begin(v),
    std::end(v),
    std::ostream_iterator<
      typename C::value_type
    >(std::cout,"\n")
  );
}
```

Example 2: predicate

```
template <class T, class D>
void f(std::vector<T>& v, const D& w)
{
  std::copy_if(
    std::begin(w),
    std::end(w),
    std::back_inserter(v),
    [](const auto& i) { return i > 0; }
  );
}
```

Example 2: conclusion

- More expressive
- Can be written and tweaked locally
- Can use different inserters
- Can use different predicates

How algorithms work

- Algorithms work on ranges, i.e. from a begin to (not including) the end
- The begin and end are indicated by an iterator
- 'An algorithm operates on its data through iterators and knows nothing about the container in which the elements are stored' [Stroustrup]

Ranges

```
std::sort( //C++98
  v.begin(),
  v.end()
);

std::sort( //C++11
  std::begin(v),
  std::end(v)
);

std::sort( //C++11
  std::begin(v),
  std::begin(v) + (v.size() / 2)
);
```

- Prefer using `std::begin(v)` over `v.begin()`

Which algorithms?

- 'An algorithm operates on its data through iterators and knows nothing about the container in which the elements are stored' [Stroustrup]

```
std::set<int> s;
assert(s.count(42)==0);

std::set<int> s;
assert(std::count(
    std::begin(s),std::end(s),42
  ) == 0
);
```

- Prefer using the member function over the algorithm with the same name
- Unsure about current best practice (`std::count` can call `std::set<T>::count`)

Iterators cannot modify containers

- 'An algorithm operates on its data through iterators and knows nothing about the container in which the elements are stored' [Stroustrup]
- This code will not change the size of the container:

```
std::remove(
  std::begin(v),
  std::end(v),
  7
);
```

Iterators cannot modify containers

- This code will rearrange the container its contents
- std::remove will return an iterator to the new end

```
const auto new_end = std::remove(
  std::begin(v),
  std::end(v),
  7
);

v.erase(new_end,std::end(v));
```

- This is called the erase-remove idiom

Some more examples

std::all_of, std::any_of, std::none_of

- Are all,any or none of these true?

```
const bool all_positive{
  std::all_of(
    std::begin(v),
    std::end(v),
    [](const auto& i) { return i > 0; }
  )
};
```

std::count, std::count_if

- Count all sevens

```
std::count(std::begin(v),std::end(v),7);

std::count_if(
  std::begin(v),
  std::end(v),
  [](const auto& i) { return i == 7; }
);
```

std::find

- A big family: `std::find`, `std::find_if`, `std::find_if_not`, `std::find_first_of`, `std::adjacent_find`, `std::find_end`
- All return an iterator to either within the container (where the element is found) or beyond the container (did not find it)

```
const auto iter = std::find_if(
  std::begin(v),
  std::end(v),
  [](const auto& i) { return i == 7; }
);

assert(iter == std::end(v)
  || *iter == 7
);
```

std::adjacent_find

```
std::vector<int> v = {0,1,2,3,2,5,6,7};

const auto iter =
  std::adjacent_find(
    std::begin(v),
    std::end(v),
    [](const auto& lhs, const auto& rhs) {
      return lhs > rhs;
    }
  )
;
assert(*iter == 3);
```

std::iota

- Sets increasing values, starting at a certain value

```
std::vector<int> v(3);
std::iota(std::begin(v),std::end(v),42);
// v will be {42,43,44}
```

std::min_element

- Returns an iterator to the lowest element

```
std::vector<int> v;

const int lowest
  = *std::min_element(
    std::begin(v),
    std::end(v)
  );
```

- Can have custom operator<
- There is also std::max_element

std::accumulate

- To calculate a sum

```
const int sum
  = std::accumulate(
    std::begin(v),
    std::end(v),
    0, // '0' is the initial value
    [](const int sum, const MyClass& my_class) {
      return sum + my_class.get();
    }
  );
```

std::transform

- To transform one range (in)to another
- Very flexible

```
std::set<int> v;
std::vector<double> w;

std::transform(
  std::begin(v),std::end(v),
  std::back_inserter(w),
  [](const int i) { return 1.0 / static_cast<double>(i); }
);
```

Extensions

- If range is from begin to end, it is easy to extend these:

```
template <class C>
void sort(C& v)
{
  std::sort(std::begin(v),std::end(v));
}
```

- Not known if/when these very common extensions will enter the STL

Conclusion

- Algorithms allow a higher expressiveness of code (due to removal of for loops)
- Some algorithms have an `_if` or predicated version
- Lambda function allow for in-place functions
- (Code will probably be faster)
- There are many more algorithms, especially find, sort and set algorithms
- You can assume others know the STL algorithms

Legal stuff

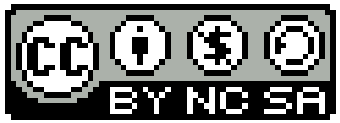

Figure 1:CC-BY-NC-SA

Download at:

`www.github.com/richelbilderbeek/CppPresentations/class_design1.pdf`

GitHub

Figure 2:GitHub

Send feedback by adding an issue or doing a pull request.