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)
- IMHO: sometimes prefer a ranged-for loop for readability

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

- Simply removed the for loop
- 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: ranged for

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

- 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: conclusions

- std::sort is more expressive
- std::sort is implemented smarter
- No need to write a sort function, can do it locally

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)
  );
}
```

Example 2: conclusions

- std::copy is more expressive
- (std::copy is implemented faster)
- No need to write a copy function, can do it locally

Example 3: raw for

```
template <class C>
void f(const C& v)
{
   const int sz{static_cast<int>(v.size())};
   for (int i=0; i!=sz; ++i) {
     std::cout << w[i] << '\n';
   }
}
```

Example 3: algorithm

```
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 3: ranged for

```
template <class C>
void f(const C& v)
{
  for (const auto& i:v) {
    std::cout << i << '\n';
  }
}
```

Example 3: conclusions

- May be a case pro ranged for?
- Use of std::ostream_iterator is a bit cumbersome (have not checked if C++14 has fixed this)
- Algorithm call can be writtem locally

Example 4: raw for loop

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

Example 4: raw for loop

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

- A predicated copy

Example 4: ranged for-loop

```
template <class C, class D>
void f(C& v, const D& w)
{
  for (const auto& i: w) {
    if (i > 0) {
      v.push_back(i);
    }
  )
}
```

Example 4: algorithm

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

Example 4: conclusion

- `std::back_inserter` can do `push_back`
- Lambda expression is short (note: from C++14)
- Can do algorithm locally and tweak it there

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)
);
```

Reversed ranges

```
std::sort( //C++98
  v.rbegin(),
  v.rend()
);

std::sort( //C++11
  std::rbegin(v),
  std::rend(std::sort)
);
```

Overview of algorithms

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

```
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

```
const int sz{static_cast<int>(v.size()};
for (int i=0; i!=sz; ++i) { v[i] = i; }
```

```
std::iota(std::begin(v),std::end(v),0);
```

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));
}
```

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

And . . .

- There are many more algorithms, here a list of my favorites I left out:
- accumulate: sum up a range of elements
- min_element: returns the smallest element in a range
- max_element: returns the largest element in a range
- remove: remove elements equal to certain value
- remove_if: remove all elements for which a predicate is true
- replace: replace every occurrence of some value in a range with another value
- replace_if: change the values of elements for which a predicate is true
- shuffle: shuffles the elements randomly
- transform: applies a function to a range of elements
- unique: remove consecutive duplicate elements in a range

Legal stuff

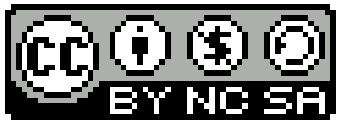

Figure 1:CC-BY-NC-SA

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.