Using and expanding STL containers and algorithms

Dimitar Mirchev @DVMirchev

Motivation

C++ Seasoning - Sean Parent, GoingNative 2013, September 4, 2013

- No raw pointers

Motivation

C++ Seasoning - Sean Parent, GoingNative 2013, September 4, 2013

- No raw pointers
- No raw synchronization primitives (mutex, atomic, semaphore, memory fence)

Motivation

C++ Seasoning - Sean Parent, GoingNative 2013, September 4, 2013

- No raw pointers
- No raw synchronization primitives (mutex, atomic, semaphore, memory fence)
- **No raw loops**

Why “no raw loops”?

- Difficult to reason about and difficult to prove postconditions
- Error prone and likely to fail under non-obvious conditions
- Introduce non-obvious performance problems
- Complicates reasoning about the surrounding code

Alternatives

- Use an existing algorithm
  - Prefer standard algorithms if available
- Implement a known algorithm as a general or generic function
  - Contribute/put it in a library
- Invent a new algorithm

Problems

How to use STL algorithms with *custom* containers?

How to use *custom* algorithms with STL containers?

Iterators

Minimal definition:

Iterator: a pointer-like object that can be incremented with ++, dereferenced with *, and compared against another iterator with !=.

<table>
<tr><th colspan="5">Iterator category</th><th>Defined operations</th></tr>
<tr><td rowspan="5">ContiguousIterator</td><td rowspan="4">RandomAccessIterator</td><td rowspan="3">BidirectionalIterator</td><td rowspan="2">ForwardIterator</td><td>InputIterator</td><td>• read
• increment (without multiple passes)</td></tr>
<tr><td></td><td>• increment (with multiple passes)</td></tr>
<tr><td colspan="2"></td><td>• decrement</td></tr>
<tr><td colspan="3"></td><td>• random access</td></tr>
<tr><td colspan="4"></td><td>• contiguous storage</td></tr>
<tr><td colspan="6">Iterators that fall into one of the above categories and also meet the requirements of OutputIterator are called mutable iterators.</td></tr>
<tr><td>OutputIterator</td><td colspan="4"></td><td>• write
• increment (without multiple passes)</td></tr>
</table>

Iterator library - cppreference.com

std::iterator_traits

- **difference_type** - a type that can be used to identify distance between iterators
- **value_type** - the type of the values that can be obtained by dereferencing the iterator.

  This type is void for output iterators.
- **pointer** - defines a pointer to the type iterated over (value_type)
- **reference** - defines a reference to the type iterated over (value_type)
- **iterator_category** - the category of the iterator. Must be one of iterator category tags.

Pointer iterator

```
template<class T, int N = 100>
class ContPIterator
{
      T data[N];

public:
      auto begin() { return &data[0]; }
      auto end() { return &data[N]; } // we never write here

      auto begin() const { return &data[0]; }
      auto end() const { return &data[N]; } // we never write here
};
```

Pointer iterator

```
template<class _Ty>
struct iterator_traits<_Ty *>
{     // get traits from pointer
      typedef random_access_iterator_tag iterator_category;
      typedef _Ty value_type;
      typedef ptrdiff_t difference_type;

      typedef _Ty *pointer;
      typedef _Ty& reference;
};
```

Pointer iterator

```
ContPIterator<int, 10> container;

// fill with numbers from 0 to 9
int n = 0;
std::generate(container.begin(), container.end(), [&n] { return n++; });

// add 5 to each element
std::transform(container.begin(), container.end(), container.begin(),
     [](int& c) { return c + 5; });

// sort, obviously
std::sort(container.begin(), container.end(), std::greater<>());

for (const auto& i : container)
     std::cout << i << ' ';
```

Push_backing

```
template<class T>
struct ContPushBack {
	using value_type = T;

	void push_back(const T& value)
	{
		m_Vect.push_back(value);
	}

	auto begin() { return m_Vect.begin(); }
	auto end() { return m_Vect.end(); }
private:
	std::vector<T> m_Vect;
};
```

Push_backing

```
_Myt& back_insert_iterator::operator=(const _Valty& _Val)
      {     // push value into container
            container->push_back(_Val);
            return (*this);
      }
```

Push_backing

```
ContPushBack<int> container;


// fill with numbers from 0 to 19
std::vector<int> v(20); int n = 0;
std::generate(v.begin(), v.end(), [&n] { return n++; });


// add only the even numbers to our container
std::copy_if(v.begin(), v.end(),
      std::back_inserter<ContPushBack<int>>(container),
      [](int n) { return n % 2 == 0; });


// make each the remainder of  deleting by 3
std::for_each(container.begin(), container.end(),
              [](int& n) { n = n % 3; });
```

Inserting

```
template<class T>
struct ContInsert {
      using iterator = int;
      using value_type = T;

      iterator insert(iterator, const T&  value)
      {
            m_Vect.push_back(value);
            return 0;
      }
      auto begin()  { return m_Vect.begin(); }
      auto end()  { return m_Vect.end(); }

 private:
      std::vector<T> m_Vect;
};
```

Inserting

```
_Myt& insert_iterator::operator=(const _Valty& _Val)
      {     // insert into container and increment stored iterator
            iter = container->insert(iter, _Val);
            ++iter;
            return (*this);
      }
```

Inserting

```
ContInsert<int> container;


// add 5 ones
std::fill_n(std::inserter<ContInsert<int>>(container, 0), 5, 1);


// multiply each by 5 then add 5 or 10 to each element
std::transform(container.begin(), container.end(), container.begin(),
     [](int& n) { return n * 5 + (RandomBool() ? 5 : 10 ); });
```

Custom Iterator and Container

Weird container and iterator

```
template<class T>
class WierdContainer {

      std::unique_ptr<WierdNode<T>> root;

 public:
      using value_type = T;
      using iterator = WierdIterator<WierdContainer<T>>;
      using _Nodeptr = WierdNode<T>*;

      iterator insert(iterator, const T&  value)
      {
            if (root)
                  root->insert(value);
            else
                  root = std::make_unique<WierdNode<T>>(value, nullptr);

            return iterator();
      }

      iterator begin() const  { return iterator(root.get()); }
      iterator end() const  { return iterator(); }
};
```

Weird container and iterator

```
template<class T>
class WierdContainer {

     std::unique_ptr<WierdNode<T>> root;

 public:
     using value_type = T;
     using iterator = WierdIterator<WierdContainer<T>>;
     using _Nodeptr = WierdNode<T>*; // non-owning pointer

     iterator insert(iterator, const T&  value)
     {
          if (root)
               root->insert(value);
          else
               root = std::make_unique<WierdNode<T>>(value, nullptr);

          return iterator();
     }

     iterator begin() const  { return iterator(root.get()); }
     iterator end() const  { return iterator(); }

};
```

Weird container and iterator

```
template<class T>
class WierdContainer {

      std::unique_ptr<WierdNode<T>> root;

 public:
      using value_type = T;
      using iterator = WierdIterator<WierdContainer<T>>;
      using _Nodeptr = WierdNode<T>*;

      iterator insert(iterator, const T&  value)
      {
            if (root)
                  root->insert(value);
            else
                  root = std::make_unique<WierdNode<T>>(value, nullptr);

            return iterator(); // empty iterator - with nullptr
      }

      iterator begin() const  { return iterator(root.get()); }
      iterator end() const  { return iterator(); }

};
```

Weird container and iterator

```
template<class T>
class WierdContainer {

      std::unique_ptr<WierdNode<T>> root;

 public:
      using value_type = T;
      using iterator = WierdIterator<WierdContainer<T>>;
      using _Nodeptr = WierdNode<T>*;

      iterator insert(iterator, const T&  value)
      {
            if (root)
                  root->insert(value);
            else
                  root = std::make_unique<WierdNode<T>>(value, nullptr);

            return iterator(); // empty iterator - with nullptr
      }

      iterator begin() const  { return iterator(root.get()); }
      iterator end() const  { return iterator(); }
};
```

Weird container and iterator

```
template<class Container>
class WierdIterator : public std::iterator<std::forward_iterator_tag, typename Container::value_type>
{
public:
	using _Myiter  = WierdIterator<Container>;
	using _Nodeptr = typename Container::_Nodeptr;

	WierdIterator()
		: _Ptr(nullptr)
	{	// construct with null node pointer
	}

	WierdIterator(_Nodeptr _Pnode)
		: _Ptr(_Pnode)
	{
	}

	value_type& operator*()
	{	// return designated value
		return _Ptr->Value();
	}

	value_type* operator->()
	{	// return pointer to class object
		return &_Ptr->Value();
	}
```

Weird container and iterator

```
template<class Container>
class WierdIterator : public std::iterator<std::forward_iterator_tag, typename Container::value_type>
{
public:
	using value_type = typename Container::value_type;
	using _Myiter  = WierdIterator<Container>;
	using _Nodeptr = typename Container::_Nodeptr;

	WierdIterator()
		: _Ptr(nullptr)
	{	// construct with null node pointer
	}

	WierdIterator(_Nodeptr _Pnode)
		: _Ptr(_Pnode)
	{
	}

	value_type& operator*()
	{	// return designated value
		return _Ptr->Value();
	}

	value_type* operator->()
	{	// return pointer to class object
		return &_Ptr->Value();
	}
```

Weird container and iterator

```
template<class Container>
class WierdIterator : public std::iterator<std::forward_iterator_tag, typename Container::value_type>
{
public:
	using value_type = typename Container::value_type;
	using _Myiter  = WierdIterator<Container>;
	using _Nodeptr = typename Container::_Nodeptr;

	WierdIterator()
		: _Ptr(nullptr)
	{	// construct with null node pointer
	}

	WierdIterator(_Nodeptr _Pnode)
		: _Ptr(_Pnode)
	{
	}

	value_type& operator*()
	{	// return designated value
		return _Ptr->Value();
	}

	value_type* operator->()
	{	// return pointer to class object
		return &_Ptr->Value();
	}
```

Weird container and iterator

```
_Myiter& operator++()
{       // preincrement
        _Ptr = _Ptr != nullptr ? _Ptr->next() : nullptr;
        return (*this);
}
_Myiter operator++(int)
{       // postincrement
        _Myiter _Tmp = *this;
        ++*this;
        return (_Tmp);
}
bool operator==(const _Myiter& _Right) const
{       // test for iterator equality
        return (_Ptr == _Right._Ptr);
}
bool operator!=(const _Myiter& _Right) const
{       // test for iterator inequality
        return (!(*this == _Right));
}
_Nodeptr _Ptr;      // pointer to node
};
```

Weird container and iterator

```
    _Myiter& operator++()
    {       // preincrement
            _Ptr = _Ptr != nullptr ? _Ptr->next() : nullptr;
            return (*this);
    }
    _Myiter operator++(int)
    {       // postincrement
            _Myiter _Tmp = *this;
            ++*this;
            return (_Tmp);
    }
    bool operator==(const _Myiter& _Right) const
    {       // test for iterator equality
            return (_Ptr == _Right._Ptr);
    }
    bool operator!=(const _Myiter& _Right) const
    {       // test for iterator inequality
            return (!(*this == _Right));
    }
    _Nodeptr _Ptr;      // pointer to node
};
```

Weird container and iterator

```
template<class T>
class WierdNode {
 public:
      WierdNode(const T& v, WierdNode<T>* p);
      T& Value();

      void insert(const T& value);
      WierdNode<T>* next();
}
```

Weird container and iterator

```
WierdContainer<int> container;

 // add 5 ones
std::fill_n(std::inserter<WierdContainer<int>>(container, container.begin()), 5, 1);

 // add 0 to 19 in a vector
std::vector<int> v(20);     int n = 0;
std::generate(v.begin(), v.end(), [&n] { return n++; });

 // copy to container only the even ones
std::copy_if(v.begin(), v.end(), std::inserter<WierdContainer<int>>(container, container.begin()), [](int
n) { return n % 2 == 0; });

  // add 5 to each element
std::transform(container.begin(), container.end(), container.begin(),
     [](int& c) { return c + 5; });

// check if all are positive
bool bIsPositive = std::any_of(container.begin(), container.end(), [](auto& n) { return n > 0; })
```

Weird container and iterator

```
WierdContainer<int> container;

 // add 5 ones
std::fill_n(std::inserter<WierdContainer<int>>(container, container.begin()), 5, 1);

 // add 0 to 19 in a vector
std::vector<int> v(20);     int n = 0;
std::generate(v.begin(), v.end(), [&n] { return n++; });

 // copy to container only the even ones
std::copy_if(v.begin(), v.end(), std::inserter<WierdContainer<int>>(container, container.begin()),
            [](int n) { return n % 2 == 0; });

  // add 5 to each element
std::transform(container.begin(), container.end(), container.begin(),
     [](int& c) { return c + 5; });

// check if all are positive
bool bIsPositive = std::any_of(container.begin(), container.end(), [](auto& n) { return n > 0; })
```

Weird container and iterator

```
WierdContainer<int> container;

 // add 5 ones
std::fill_n(std::inserter<WierdContainer<int>>(container, container.begin()), 5, 1);

 // add 0 to 19 in a vector
std::vector<int> v(20);     int n = 0;
std::generate(v.begin(), v.end(), [&n] { return n++; });

 // copy to container only the even ones
std::copy_if(v.begin(), v.end(), std::inserter<WierdContainer<int>>(container, container.begin()),
            [](int n) { return n % 2 == 0; });

  // add 5 to each element
std::transform(container.begin(), container.end(), container.begin(),
             [](int& c) { return c + 5; });

// check if all are positive
bool bIsPositive = std::any_of(container.begin(), container.end(), [](auto& n) { return n > 0; })
```

Weird container and iterator

```
WierdContainer<int> container;

 // add 5 ones
std::fill_n(std::inserter<WierdContainer<int>>(container, container.begin()), 5, 1);

 // add 0 to 19 in a vector
std::vector<int> v(20);     int n = 0;
std::generate(v.begin(), v.end(), [&n] { return n++; });

 // copy to container only the even ones
std::copy_if(v.begin(), v.end(), std::inserter<WierdContainer<int>>(container, container.begin()),
            [](int n) { return n % 2 == 0; });

  // add 5 to each element
std::transform(container.begin(), container.end(), container.begin(),
             [](int& c) { return c + 5; });

// check if all are positive
bool bIsPositive = std::any_of(container.begin(), container.end(), [](auto& n) { return n > 0; })
```

Weird container and iterator

```
// slide them forward by 1 element
std::rotate(container.begin(), ++container.begin(), container.end());
```

Weird container and iterator

```
// slide them forward by 1 element
std::rotate(container.begin(), ++container.begin(), container.end());


 // sum all elements
int sum = std::accumulate(container.begin(), container.end(), 0);
std::cout << sum << '\n';

 // find first adjusted that first is greater than second
auto it = std::adjacent_find(container.begin(), container.end(), std::greater<int>());

 std::cout << *it << '\n';

 WierdContainer<int> container2;
// use boost! algorithm copy until the element that is equal to 17
boost::algorithm::copy_until(container.begin(), container.end(),
                             std::inserter<WierdContainer<int>>(container2, container2.begin()),
                              [](int i) { return i == 17; });
```

Weird container and iterator

```
// slide them forward by 1 element
std::rotate(container.begin(), ++container.begin(), container.end());


 // sum all elements
int sum = std::accumulate(container.begin(), container.end(), 0);
std::cout << sum << '\n';

 // find first adjusted that first is greater than second
auto it = std::adjacent_find(container.begin(), container.end(), std::greater<int>());

 std::cout << *it << '\n';

 WierdContainer<int> container2;
// use boost! algorithm copy until the element that is equal to 17
boost::algorithm::copy_until(container.begin(), container.end(),
                             std::inserter<WierdContainer<int>>(container2, container2.begin()),
                              [](int i) { return i == 17; });
```

Weird container and iterator

```
// slide them forward by 1 element
std::rotate(container.begin(), ++container.begin(), container.end());


 // sum all elements
int sum = std::accumulate(container.begin(), container.end(), 0);
std::cout << sum << '\n';

 // find first adjusted that first is greater than second
auto it = std::adjacent_find(container.begin(), container.end(), std::greater<int>());

 std::cout << *it << '\n';

 WierdContainer<int> container2;
// use boost! algorithm copy until the element that is equal to 17
boost::algorithm::copy_until(container.begin(), container.end(),
                             std::inserter<WierdContainer<int>>(container2, container2.begin()),
                             [](int i) { return i == 17; });
```

Weird container and iterator

```
// slide them forward by 1 element
std::rotate(container.begin(), ++container.begin(), container.end());


 // sum all elements
int sum = std::accumulate(container.begin(), container.end(), 0);
std::cout << sum << '\n';

 // find first adjusted that first is greater than second
auto it = std::adjacent_find(container.begin(), container.end(), std::greater<int>());

 std::cout << *it << '\n';

 WierdContainer<int> container2;
// use boost! algorithm copy until the element that is equal to 17
boost::algorithm::copy_until(container.begin(), container.end(),
                             std::inserter<WierdContainer<int>>(container2, container2.begin()),
                             [](int i) { return i == 17; });
```

Problems

How to use STL algorithms with *custom* containers?

How to use *custom* algorithms with STL containers?

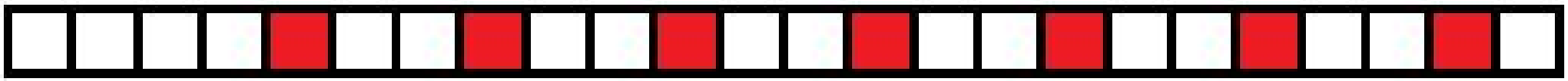
Writing an algorithm

- Use an existing algorithm
  - Prefer standard algorithms if available
- Implement a known algorithm as a general or generic function
  - Contribute/put it in a library
- Invent a new algorithm

Solving a problem

Problem:

Find if a value N occurs every D times after its first occurrence in a sequence:



Solving a problem

```
template <class Iter, class Type>
bool is_every_D_after_first_occurrence(Iter _First, Iter _Last, size_t D, Type N)
{
    auto _Found = std::find(_First, _Last, N);

    if (_First == _Last)
        return false;

    do
    {
        for (auto i = 0; i < D; ++i)
            if (++_Found == _Last)
                return true;

        if (N != *_Found)
            return false;

    } while (true);
}
```

Solving a problem

```
template <class Iter, class Type>
bool is_every_D_after_first_occurrence(Iter _First, Iter _Last, size_t D, Type N)
{
      auto _Found = std::find(_First, _Last, N);

      if (_First == _Last)
            return false;

      do
      {
            for (auto i = 0; i < D; ++i)
                  if (++_Found == _Last)
                        return true;

            if (N != *_Found)
                  return false;

      } while (true);
}
```

Solving a problem

```
template <class Iter, class Type>
bool is_every_D_after_first_occurrence(Iter _First, Iter _Last, size_t D, Type N)
{
      auto _Found = std::find(_First, _Last, N);

      if (_First == _Last)
            return false;

      do
      {
            for (auto i = 0; i < D; ++i)
                  if (++_Found == _Last)
                        return true;

            if (N != *_Found)
                  return false;

      } while (true);
}
```

Solving a problem

```
template <class Iter, class Type>
bool is_every_D_after_first_occurrence(Iter _First, Iter _Last, size_t D, Type N)
{
      auto _Found = std::find(_First, _Last, N);

      if (_First == _Last)
            return false;

      do
      {
            for (auto i = 0; i < D; ++i)
                  if (++_Found == _Last)
                        return true;

            if (N != *_Found)
                  return false;

      } while (true);
}
```

<= raw loop

Solving a problem

```
template <class Iter>
bool is_every_D(Iter _First, Iter _Last, size_t D)
{
	if (_First == _Last)
		return false;

	auto N = *_First;
	do
	{
		for (auto i = 0; i < D; ++i)
			if (++_First == _Last)
				return true;

		if (N != *_First)
			return false;

	} while (true);
}

template <class Iter, class Type>
bool is_every_D_after_first_occurrence(Iter _First, Iter _Last, size_t D, Type N)
{
	auto _Found = std::find(_First, _Last, N);

	return is_every_D(_Found, _Last, D);
}
```

Solving a problem

```
template <class Iter>
bool is_every_D(Iter _First, Iter _Last, size_t D)
{
	if (_First == _Last)
		return false;

	auto N = *_First;
	do
	{
		for (auto i = 0; i < D; ++i)
			if (++_First == _Last)
				return true;

		if (N != *_First)
			return false;

	} while (true);
}

template <class Iter, class Type>
bool is_every_D_after_first_occurrence(Iter _First, Iter _Last, size_t D, Type N)
{
	auto _Found = std::find(_First, _Last, N);

	return is_every_D(_Found, _Last, D);
}
```

Solving a problem

```
template <class Iter>
bool is_every_D(Iter _First, Iter _Last, size_t D)
{
	if (_First == _Last)
		return false;

	auto N = *_First;
	do
	{
		for (auto i = 0; i < D; ++i)
			if (++_First == _Last)
				return true;

		if (N != *_First)
			return false;

	} while (true);
}

template <class Iter, class Type>
bool is_every_D_after_first_occurrence(Iter _First, Iter _Last, size_t D, Type N)
{
	auto _Found = std::find(_First, _Last, N);

	return is_every_D(_Found, _Last, D);
}
```

We are using only forward iterator functionality

Solving a problem

```
template <class Iter>
bool is_every_D_impl(std::forward_iterator_tag, Iter _First, Iter _Last, size_t D)
{
	auto N = *_First;
	do
	{
		if (_First == _Last)
			return false;

		for (auto i = 0; i < D; ++i)
			if (++_First == _Last)
				return true;

		if (N != *_First)
			return false;

	} while (true);
}

template <class Iter>
bool is_every_D(Iter _First, Iter _Last, size_t D)
{
	return is_every_D_impl(std::iterator_traits<Iter>::iterator_category(), _First, _Last, D);
}
```

Solving a problem

```
template <class Iter>
bool is_every_D_impl(std::forward_iterator_tag, Iter _First, Iter _Last, size_t D)
{
	auto N = *_First;
	do
	{
		if (_First == _Last)
			return false;

		for (auto i = 0; i < D; ++i)
			if (++_First == _Last)
				return true;

		if (N != *_First)
			return false;

	} while (true);
}

template <class Iter>
bool is_every_D(Iter _First, Iter _Last, size_t D)
{
	return is_every_D_impl(typename std::iterator_traits<Iter>::iterator_category(), _First, _Last, D);
}
```

Solving a problem

```
template <class Iter>
bool is_every_D_impl(std::forward_iterator_tag, Iter _First, Iter _Last, size_t D)
{
	auto N = *_First;
	do
	{
		if (_First == _Last)
			return false;

		for (auto i = 0; i < D; ++i)
			if (++_First == _Last)
				return true;

		if (N != *_First)
			return false;

	} while (true);
}

template <class Iter>
bool is_every_D(Iter _First, Iter _Last, size_t D)
{
	return is_every_D_impl(typename std::iterator_traits<Iter>::iterator_category(), _First, _Last, D);
}
```

Solving a problem

```
template <class Iter>
bool is_every_D_impl(std::random_access_iterator_tag, Iter _First, Iter _Last, size_t D)
{
    if (_First == _Last)
        return false;

    auto N = *_First;
    do
    {
        if (_Last - _First <= D)
            return true;

        _First += D;

        if (N != *_First)
            return false;

    } while (true);
}
```

Solving a problem

```
template <class Iter>
bool is_every_D_impl(std::random_access_iterator_tag, Iter _First, Iter _Last, size_t D)
{
	if (_First == _Last)
		return false;

	auto N = *_First;
	do
	{
		if (_Last - _First <= D)
			return true;

		_First += D;

		if (N != *_First)
			return false;

	} while (true);
}
```

Solving a problem

```
template <class Iter>
bool is_every_D_impl(std::random_access_iterator_tag, Iter _First, Iter _Last, size_t D);

template <class Iter>
bool is_every_D_impl(std::forward_iterator_tag, Iter _First, Iter _Last, size_t D);

template <class Iter>
bool is_every_D(Iter _First, Iter _Last, size_t D)
{
	return is_every_D_impl(typename std::iterator_traits<Iter>::iterator_category(), _First, _Last, D);
}

template <class Iter, class Type>
bool is_every_D_after_first_occurrence(Iter _First, Iter _Last, size_t D, Type N)
{
	auto _Found = std::find(_First, _Last, N);

	return is_every_D(_Found, _Last, D);
}
```

Solving a problem

```
std::vector<int> v({ 1,1,1,1,1,2,1,1,2,1,1 });

// { 1,1,1,1,1,2,1,1,2,1,1 }
bool bIsVector1 = is_every_D_after_first_occurrence(v.begin(), v.end(), 3, 2);  // true
// { 1,1,1,1,1,2,1,1,2,1,1 }
bool bIsVector2 = is_every_D_after_first_occurrence(v.begin(), v.end(), 3, 1);  // true

std::list<int> l({ 1,1,2,1,1,2,1,1,2,1,1,3 });
// { 1,1,2,1,1,2,1,1,2,1,1,3 }
bool bIsList1 = is_every_D_after_first_occurrence(l.begin(), l.end(), 3, 2);    // false
// { 1,1,2,1,1,2,1,1,2,1,1,3 }
bool bIsList2 = is_every_D_after_first_occurrence(l.begin(), l.end(), 3, 1);    // true
```

What did I skipped?

- const qualifiers
  - The custom iterator should have a const variant (equivalent to iterator_const) with
    - operator*() const
    - operator->() const
    - const comparison, arithmetic operators and operator[]
- Bidirectional iterator
  - Should have operator--()
  - --container.end();
- Random access iterator
  - Should have operator[]()

What did I skipped?

- const qualifiers
  - The custom iterator should have a const variant (equivalent to iterator_const) with
    - operator*() const
    - operator->() const
    - const comparison, arithmetic operators and operator[]
- Bidirectional iterator
  - Should have operator--()
  - --container.end();
- Random access iterator
  - Should have operator[]()

What did I skipped?

- const qualifiers
  - The custom iterator should have a const variant (equivalent to iterator_const) with
    - operator*() const
    - operator->() const
    - const comparison, arithmetic operators and operator[]
- Bidirectional iterator
  - Should have operator--()
  - --container.end();
- Random access iterator
  - Should have operator[]()

What did I skipped?

- const qualifiers
  - The custom iterator should have a const variant (equivalent to iterator_const) with
    - operator*() const
    - operator->() const
    - const comparison, arithmetic operators and operator[]
- Bidirectional iterator
  - Should have operator--()
  - --container.end();
- Random access iterator
  - Should have operator[]()
- Deletion
  - Done by containers
  - erase–remove idiom

Sources

- C++ Seasoning
- 'STL Algorithms – How to Use Them and How to Write Your Own' - Marshall Clow [ACCU 2016]
- Defining C++ Iterators, Chris Riesbeck
-

Thank you

Writing an algorithm

```
void do_something_to_every_is_N_after_X_occurrence_carefully(std::vector<int>& v)
{
   while (std::next_permutation(v.begin(), v.end()));
}
```

Writing an algorithm

```
void do_something_to_every_is_N_after_X_occurrence_carefully(std::vector<int>& v)
{
  while (std::next_permutation(v.begin(), v.end()));
}
```

```
vector<int> v = {3,6,4,2,5,1};

do_something_to_every_is_N_after_X_occurrence_carefully(v);

std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
```