Standard Iterators Library

Iterators are used for traversing containers, other sequences (e.g. built-in arrays), streams and strings

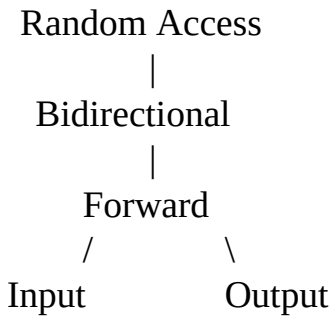
- iterators are generalizations of pointers
  - an iterator can point to a certain element in a range or have a *past-the-end value*
  - allows programs to operate on different data structures in a uniform manner
- many built-in operations for containers have iterators as argument and return value
- algorithms use iterators to operate on containers, other data structures and streams
  - the type of iterators a certain algorithm require states the *minimum* operability required
- all containers, except adaptors, have member functions for providing iterators to their beginning and end
  - was covered by the standard container lecture
- the standard library supports iterating over
  - containers
  - strings
  - streams
  - stream buffers
  - regular expressions
- range access functions
  - uniform iterator interface for containers (class types supplying iterators), arrays, initializer lists, and more
  - was shown as template examples

Regarding exam

- important to have a good overview and understanding of the iterators library
  - good knowledge of different iterator types and their common and specific features
  - sufficient practice in using iterators, in combination with other components
- *cplusplus.com Reference* will be available, using a web browser
- see the course examination page for more information

Iterator categories

- there are five iterator categories:



- *not* a class hierarchy – based on which operations can be applied to an iterator
  - Forward iterators fulfill all requirements on Input and Output iterators
  - Bidirectional iterators fulfill all requirements on Forward iterators
  - Random Access iterators fulfill all requirements on Bidirectional iterators
- algorithms use these as type names to indicate iterator requirements

Iterator operations

Besides copying and assignment the operations below are available for the different iterators. Two distinguishing characteristics:

- how the different iterators can be operated upon for stepping/moving them
- if one can use them to read and/or write the objects they point at

Iterator type:	**Input**	**Output**	**Forward**	**Bidirectional**	**Random Access**
`==  !=`	Yes	Yes	Yes	Yes	Yes
`*`	Read	Write	Read/Write	Read/Write	Read/Write
`->`	Read	Write	Read/Write	Read/Write	Read/Write
`++`	Yes	Yes	Yes	Yes	Yes
`--`	–	–	–	Yes	Yes
`+  +=  -  -=`	–	–	–	–	Yes
`<  <=  >  >=`	–	–	–	–	Yes
`i[n]`	–	–	–	–	Yes
`advance(`*it*`, `*n*`)`	Yes	–	Yes	Yes	Yes
`distance(`*it1*`, `*it2*`)`	Yes	–	Yes	Yes	Yes
Provided by:			`forward_list` unordered associative containers	`list` *(ordered) associative containers*	`vector, deque array`

Note: since only RandomAccess have + and - operators, advance() and distance() are provided for the others.

Range access

Function templates returning range iterators (implementation was shown as template examples).

`auto first = begin(c)`	returns c.begin()	class types
`auto past_end = end(c)`	returns c.end()	
`T* first = begin(a)`	returns a, T is the element type for a	array types
`T* past_end = end(a)`	returns a + N, N is the dimension for a, T is the element type	

- available when including `<iterator>` but also when including

  `<array>` `<deque>` `<forward_list>` `<list>` `<vector>` `<map>` `<set>`

  `<unordered_map>` `<unordered_set>` `<regex>` `<string>` `<initializer_list>`

  `<valarray>`

- applied to a C array

```
int a[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
copy(begin(a), end(a), ostream_iterator<int>{cout, " "});
```

  – `a` could be a container, a string, a regular expression, or any type for which `begin()` and `end()` are defined

Example: Iterator operations on vector (Random Access iterators)

```
vector<int> v{begin(x), end(x)};

vector<int>::iterator past_end{ end(v) };

for (auto it = begin(v); it != past_end; ++it)
{
   *it += 1;
}

cout << *(first + 1) << endl;                    // it + n (iterator)
cout << *(1 + first) << endl;                    // n + it (iterator)

cout << *(past_end - 1) << endl;                 // it - n (iterator)

cout << begin(v) - end(v) << endl;               // it - it (distance)

auto it1 = first + (past_end - first) / 2;

it1 += 2;                                        // it += n (forward two steps)

cout << first[4] << endl                         // indexing, the 5:th element

auto it2 = begin(v);                             // copying

it1 = past_end - 1;                              // assignment

if (it2 == past_end) cout << "it2 == past_end\n"; // comparison
```

The iterator template

Can be used as base class to ease definition of required types for new iterators.

```
template<class Category, class T,
         class Distance = std::ptrdiff_t, class Pointer = T*, class Reference = T&>
struct iterator
{
   using value_type        = T;
   using difference_type   = Distance;
   using pointer           = Pointer;
   using reference         = Reference;
   using iterator_category = Category;
};


template<typename T>
struct my_iterator : public std::iterator<std::forward_iterator_tag, T>
{
   T&          operator*();
   const T&    operator*() const;
   my_iterator& operator++();
   my_iterator  operator++(int);
};
```

Adding iterators to String

```
class String
{
public:
    using value_type = char;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t

    using reference = char&;
    using const_reference = const char&;

    using pointer = char*;
    using const_pointer = const char*;

    using iterator = pointer;
    using const_iterator = const_pointer;

    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
```

- in this case `iterator` and `const_iterator` was simple
  - ordinary character pointer types
  - built-in pointer operators can be used (`++`, `--`, `*`, …)
- when we have `iterator` and `const_iterator`, `reverse_iterator` and `const_reverse_iterator` is simple
  - use the `std::reverse_iterator` template to define them

Iterator functions for String (1)

- the following member functions is the basic set of iterator functions
  - `begin()` and `end()` are overloaded in a non-const and a const version

```
iterator       begin();
const_iterator begin() const;

iterator       end();
const_iterator end() const;
```

  - `cbegin()` and `cend()` are not overloaded, same for both non-const and a const String

```
const_iterator cbegin() const;
const_iterator cend() const;
```

- the following member functions are expected also, when reverse iterators are supplied

```
reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;

reverse_iterator       rend();
const_reverse_iterator rend() const;

const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;
```

Iterator functions for String (2)

- some iterator functions are implemented with `p_` and `size_`, e.g.:

```
String::iterator String::begin() { return iterator(p_); }

String::const_iterator String::begin() const { return const_iterator(p_); }

String::iterator String::end() { return iterator(p_ + size_); }

String::const_iterator String::cbegin() const { return const_iterator(p_); }
```

- others are implemented using `begin()` and `end()`, e.g.:

```
String::reverse_iterator String::rbegin() { return reverse_iterator(end()); }

String::reverse_iterator String::rend() { return reverse_iterator(begin()); }

String::const_reverse_iterator
String::crbegin() const { return const_reverse_iterator(end()); }

String::const_reverse_iterator
String::crend() const { return const_reverse_iterator(begin()); }
```

Stream iterators

```
#include <iterator>
```

- reading one value at a time from a `vector` or from a sequential file does not differ much
  - in both cases operations corresponding to an Input iterator is required
- for streams there are two classes acting as a shell for a stream and provide an iterator interface
  - `istream_iterator`
  - `ostream_iterator`
- `istream_iterator` is instantiated for the data type to be read and bound to an input stream

```
istream_iterator<int>  eos;          // end-of-stream
istream_iterator<int>  is{cin};

vector<int> v{is, eos};
```

Note: `istream_iterator` uses **operator**>> to read values

- `ostream_iterator` is instantiated for the data type to be written and bound to an output stream and a delimiter string

```
ostream_iterator<int> os{cout, " "};

copy(begin(v), end(v), os);
```

or use a temporary as argument

```
copy(begin(v), end(v), ostream_iterator<int>{cout, " "});
```

Note: `ostream_iterator` uses **operator**<< to print values

Stream iterators – example

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <algorithm>
using namespace std;

int main(int argc, char* argv[])
{
   if (argc != 2)
   {
      cout << "usage: " << argv[0] << " file\n";
      return 1;
   }

   ifstream infile{argv[1]};

   if (!infile)
   {
      cout << "could not open " << argv[1] << '\n';
      return 2;
   }

   istream_iterator<int>  eos;
   istream_iterator<int>  input{infile};
   ostream_iterator<int>  output{cout, "\n"};

   copy(input, eos, output);
}
```

Insert iterators

```
#include <iterator>
```

- using algorithm `copy` to copy one vector to another:

```
copy(begin(v1), end(v1), begin(v2));
```

  - this will fail, if the size of the source vector is larger than the capacity of the destination vector
  - `copy` uses assignment, **operator**=

- instead use a `back_insert_iterator` bound to `v2`

```
copy(begin(v1), end(v1), back_inserter(v2));
```

  - `back_insert_iterator` uses `push_back()`
  - utility function `back_inserter` creates a `back_insert_iterator` and binds it to the destination container

- there are three kind of insert iterators
  - `back_insert_iterator` utility function `back_inserter(c)`
  - `front_insert_iterator` utility function `front_inserter(c)`
  - `insert_iterator` utility function `inserter(c, it)`

Implementation of back_insert_iterator, in principle (1)

The problem in algorithm copy is related to **operator**=

```
template<typename InputIterator, typename OutputIterator>
OutputIterator
copy(InputIterator first, InputIterator last, OutputIterator result)
{
   for ( ; first != last; ++result, ++first)
      *result = *first;
   return result;
}
```

- **operator**= does not increase the capacity of the destination container, if needed
  - the container is not at all known
- we would like the assignment statement to be replaced with

```
v2.push_back(*first);
```

- possible, if
  - iterator `result` stores a reference (pointer) to the destination container (`v2`)
  - **operator**`*` applied to `result` just returns `result` – we need `result` on the left hand side of **operator**=
  - **operator**= applied to `result` and `*first` perform `push_back(*first)` on the destination container

Note: **operator**++ need not do anything – the iterator as such does not move in any way

Implementation of back_insert_iterator, in principle (2)

```
template<typename Container>
class back_insert_iterator : public iterator<output_iterator_tag, void, void, void, void>
{
public:
   using container_type = Container;

   explicit back_insert_iterator(Container& x) : container{&x} {}

   back_insert_iterator& operator=(typename Container::const_reference value) & {
      container->push_back(value);
      return *this;
   }

   back_insert_iterator& operator*() { return *this; }
   back_insert_iterator& operator++() { return *this; }
   back_insert_iterator operator++(int) { return *this; }

protected:
   Container* container;
};

template<typename Container>
back_insert_iterator<Container> back_inserter(Container& x)
{
   return back_insert_iterator<Container>{x};
}
```

Insert iterators, stream iterators, and container iterators (range access) – example

Read input from a stream and store in a vector using a back_inserter, sort the vector and output the result.

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
   vector<int> v;

   copy(istream_iterator<int>{cin}, istream_iterator<int>{}, back_inserter(v));

   sort(begin(v), end(v));

   copy(begin(v), end(v), ostream_iterator<int>{cout, " "});
}
```

Note, reading into the vector can in this case be be made easier:

```
   vector<int> v{ istream_iterator<int>{cin}, istream_iterator<int>{} };
```

Move iterators

Class template `move_iterator` is an iterator adaptor.

- same behaviour as the underlying iterator
- its dereference operator converts the value returned by the underlying iterators dereference operator to an *rvalue reference*

```
value_type&& operator*() const { return std::move(*iter); }
```

- some algorithms can be called with move iterators to replace copying with moving

```
list<string> a{ istream_iterator<string>{cin}, istream_iterator<string>{} };
// Copy strings into v1
vector<string> v1{ begin(a), end(a) };
// Move strings into v2
vector<string> v2{ make_move_iterator(begin(a)), make_move_iterator(end(a)) };
// The elements of s are now empty strings
```

Some comments on the Iterators library

- decouples algorithms from data structures and other iterateable structures
  - one algorithm can be used for many different data structures
  - specialized algorithms required only when necessary for a data structure
- iterator categories specifies required functionality
  - specifies minimum requirements
  - actual iterators supplied can be more advanced
- the iterators library provide several base classes and functions to simplify the definition of iterators
- move iterators and range access functions in C++11 adds new interesting possibilities
  - more efficient move semantics can be used instead of copying, when appropriate
  - range access functions makes iterating over simple arrays more “container-like”
  - range access functions appears under the hood of the new *range* **for** *loop*