USING THE STANDARD TEMPLATE LIBRARY (STL)

Lars Ole Schwen

MEVIS Developer Seminar, 2015-09-02

```
#include<algorithm>
#include<vector>

int main ( const int, const char** const ) {
  const unsigned short myArray[5] = { 4, 7, 1, 1, 42 };
  std::vector<unsigned short> myVec ( myArray, myArray+5 );
  std::shuffle ( myVec.begin(), myVec.end(), std::mt19937 ( 23 ) );
  std::vector<unsigned short>::const_iterator
    it = std::find ( myVec.begin(), myVec.end(), 42 );
  return ( EXIT_SUCCESS );
}
```

USING THE STANDARD TEMPLATE LIBRARY (STL)

1. Introduction
2. Concepts
3. Containers
4. Iterators
5. Algorithms and Utilities

1. Introduction

Contents

1. Introduction

Disclaimer

This talk is about …

- concepts of STL
- examples of STL usage
- examples should work with C++-11 (compiled on gcc-4.6.3, clang 3.6, and Visual Studio 2013)
- code examples available (incomplete excerpts in slides)
- Thanks to Hans M. for feedback!

1. Introduction

Disclaimer

This talk is about …

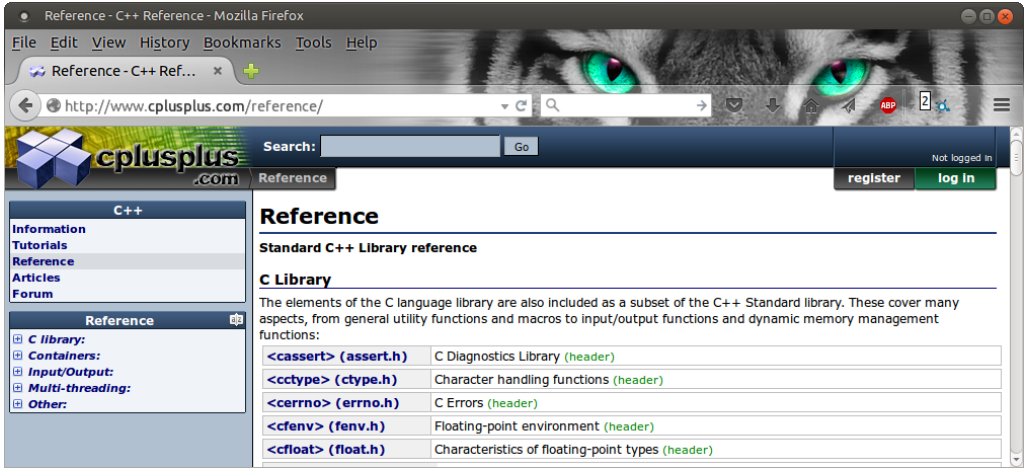
- concepts of STL
- examples of STL usage
- examples should work with C++-11 (compiled on gcc-4.6.3, clang 3.6, and Visual Studio 2013)
- code examples available (incomplete excerpts in slides)
- Thanks to Hans M. for feedback!

… but

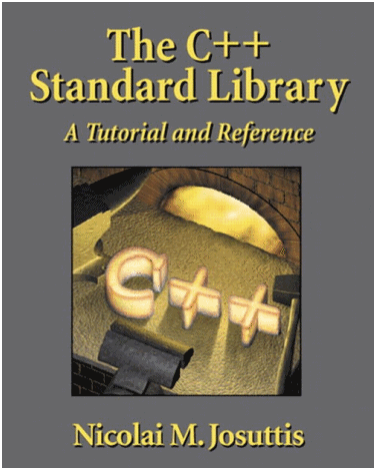
- I'm not a native STL speaker
- not exhaustive about STL
- no reference, see `http://www.cplusplus.com/reference` instead
- some observations may not be guaranteed
- boost or other libraries may offer better solutions for your needs
- example code quality not optimal
- not covered:
  - STL smart pointers
  - multithreading
  - exception handling
  - strings, stringstreams

1. Introduction

Further Reading



Further Reading



2. Concepts

Contents

2. Concepts

STL

- templates extensively used (with all pros and cons)

2. Concepts

STL

- templates extensively used (with all pros and cons)
- `lowercase_names_with_underscores`

2. Concepts

STL

- templates extensively used (with all pros and cons)
- `lowercase_names_with_underscores`
- data structures (containers) for specific use cases

2. Concepts

STL

- templates extensively used (with all pros and cons)
- `lowercase_names_with_underscores`
- data structures (containers) for specific use cases
- generic access by iterators

2. Concepts

STL

- templates extensively used (with all pros and cons)
- `lowercase_names_with_underscores`
- data structures (containers) for specific use cases
- generic access by iterators
- algorithms and utilities **independent of containers**

2. Concepts

STL

- templates extensively used (with all pros and cons)
- `lowercase_names_with_underscores`
- data structures (containers) for specific use cases
- generic access by iterators
- algorithms and utilities **independent of containers**
- efficient (memory, time) if used correctly

2. Concepts

STL

- templates extensively used (with all pros and cons)
- `lowercase_names_with_underscores`
- data structures (containers) for specific use cases
- generic access by iterators
- algorithms and utilities **independent of containers**
- efficient (memory, time) if used correctly
- complexity guarantees

2. Concepts

First Example

```
#include<cstdlib>
#include<vector>

int main ( const int, const char** const ) {
  const unsigned int size = 10;
  std::vector<unsigned int> myVec(size);
  for ( unsigned int i = 0; i < size; ++i ) {
    myVec[i] = 42*i+23;
  }
  return ( EXIT_SUCCESS );
}
```

⇝ STLSimple.cpp

✘ not very STLish

3. Containers

Contents

3. Containers

Selected Containers

- `std::vector`
  - one-dimensional array
  - random access via index

3. Containers

Selected Containers

- `std::vector`
  - one-dimensional array
  - random access via index
- `std::list`
  - bidirectionally linked list
  - cheap insertion
  - extension: `std::forward_list`

3. Containers

Selected Containers

- `std::vector`
  - one-dimensional array
  - random access via index
- `std::list`
  - bidirectionally linked list
  - cheap insertion
  - extension: `std::forward_list`
- `std::set`
  - mathematical set: objects contained once
  - internally sorted
  - extensions: `std::multiset`, `std::unordered_set`

3. Containers

Selected Containers

- `std::vector`
  - one-dimensional array
  - random access via index
- `std::list`
  - bidirectionally linked list
  - cheap insertion
  - extension: `std::forward_list`
- `std::set`
  - mathematical set: objects contained once
  - internally sorted
  - extensions: `std::multiset`, `std::unordered_set`
- `std::map`
  - “dictionary”, values for unique keys
  - internally sorted
  - extensions: `std::multimap`, `std::unordered_map`

3. Containers

Vectors

`std::vector<DataType>`

- one-dimensional array with dynamic size
- random access via index
- special `std::vector<bool>`

3. Containers

Vector 1

```
#include<vector>
// ...
std::vector<unsigned int> myVec(size); // create, 0-initialized

myVec[5] = 42; // random access
myVec.at(3) = 23; // random access with bounds checking

myVec.resize(12); // change size

myVec.push_back(23); // insert at (change size)
myVec.pop_back();    // remove last element

myVec.reserve(120); // allocation is logarithmic, but still ...

// iterate, note the ”auto” keyword (C++ 11)
for ( auto it = myVec.begin(); it != myVec.end(); ++it ) {
  std::cout << *it << std::endl;
}

myVec.clear(); // resize to 0
```

⇝ STLVector_Basic.cpp

3. Containers

Vector 2

```
#include<vector>
#include<algorithm>
// ...
std::vector<signed int> myVecA, myVecB;
// ... // fill vectors

// insert via iterator
std::vector<signed int>::iterator destPos = myVecA.begin();
++destPos; ++destPos;
myVecA.insert ( destPos, myVecB.rbegin(), myVecB.rend() );

// delete values (not indices) 60 (included) … 275 (excluded!)
signed int
  firstEntryToDelete = 60,
  firstEntryAfterDeletion = 275;
std::vector<signed int>::iterator
  rangeBegin = std::find ( myVecA.begin(), myVecA.end(), firstEntryToDelete ),
  rangeEnd = std::find ( myVecA.begin(), myVecA.end(), firstEntryAfterDeletion );
myVecA.erase ( rangeBegin, rangeEnd );
```

⇝ STLVector_Advanced.cpp

3. Containers

Lists

`std::list<DataType>`

- bidirectionally linked list
- no random access
- cheap insertion
- extension: `std::forward_list`

3. Containers

List 1

```
#include<list>

std::list<unsigned int> myList; // no size

for ( int i = 0; i < 12; ++i ) {
  myList.push_back ( 4 * i + 3 );
}

std::cout << ”myList␣is␣of␣size␣” << myList.size() << std::endl;
// iterate over list
for ( auto it = myList.begin(); it != myList.end(); ++it ) {
  std::cout << *it << std::endl;
}
```

3. Containers

List 2

```
std::list<unsigned int>::iterator it = myList.begin();
++it; ++it;
myList.insert ( it, 21859275 ); // insert new entry before it
                                // this is cheaper than for vectors
--it;
myList.erase ( it );

myList.push_front ( 4711 );
myList.push_back ( 28359 );

myList.pop_back();
myList.pop_front();

myList.clear(); // resize to 0
```

⇝ STLList_Basic.cpp

3. Containers

List 3

```
#include<list>

struct isOdd {
  template<typename IntegerType>
  bool operator() ( const IntegerType &Value ) {
    return ( ( Value % 2 ) == 1 );
  }
};

// in main program:
std::list<signed long int> myListA, myListB;

std::list<signed long int>::iterator myIt = myListA.begin();
std::advance ( myIt, 3 );
myListA.splice ( myIt, myListB );

myListA.sort();         // list-specific variants of generic
myListA.remove ( 65 );  // sort, remove, unique, remove_if
myListA.unique();
myListA.remove_if ( isOdd() );
```

⇝ STLList_Advanced.cpp

3. Containers

Vector vs. List 1

When using objects that are expensive to handle …

```
struct ExpensiveToCreate {
  ExpensiveToCreate ( ) : _val ( 0 ) {
    std::cout << ”␣␣EXPENSIVE␣create” << std::endl;
  }
  ExpensiveToCreate ( const size_t Val ) : _val ( Val ) {
    std::cout << ”␣␣EXPENSIVE␣create” << std::endl;
  }
  // ...
  // similar for destructor, copy constructor, operator=

  bool operator< ( const ExpensiveToCreate& Other ) const {
    return ( ( this->_val < Other._val ) );
  }

private:
  size_t _val;
};
```

3. Containers

Vector vs. List 2

… choose an efficient data structure!

```
const size_t mySize = 4;

std::vector<ExpensiveToCreate> myVector ( mySize );
for ( size_t i = 0; i < mySize; ++i ) {
  myVector[i] = ExpensiveToCreate ( mySize - i );
}
std::sort ( myVector.begin(), myVector.end() );

std::list<ExpensiveToCreate> myList;
for ( size_t i = 0; i < mySize; ++i ) {
  myList.emplace_back ( i );
}
myList.sort();
```

⇝ STLVectorVsList.cpp

3. Containers

Benchmark: List vs. Forward List

```
std::list<signed long int> myListC;
fillWithSomeData ( myListC );
std::cout << ”Modifying␣list␣…␣” << std::flush;
std::chrono::high_resolution_clock::time_point
  t1 = std::chrono::high_resolution_clock::now();

for ( size_t i = 0; i < ListSize; ++i ) {
  typename std::list<signed long int>::iterator it = myListC.begin();
  ++it;
  myListC.insert ( it, 4711 );
}

std::chrono::high_resolution_clock::time_point
  t2 = std::chrono::high_resolution_clock::now();
const std::chrono::duration<double>
  timeSpan = std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);
std::cout << ”done,␣took␣” << timeSpan.count() << std::endl;
```

Benchmark: List vs. Forward List

```
std::forward_list<signed long int> myListD;
fillWithSomeData ( myListD );
std::cout << ”Modifying␣forward␣list␣…␣” << std::flush;
std::chrono::high_resolution_clock::time_point
  t1 = std::chrono::high_resolution_clock::now();

for ( size_t i = 0; i < ListSize; ++i ) {
  typename std::forward_list<signed long int>::iterator it = myListD.begin();
  ++it;
  myListD.insert_after ( it, 4711 );
}

std::chrono::high_resolution_clock::time_point
  t2 = std::chrono::high_resolution_clock::now();
const std::chrono::duration<double>
  timeSpan = std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);
std::cout << ”done,␣took␣” << timeSpan.count() << std::endl;
```

⇝ STLForwardList.cpp

3. Containers

A Note on Code Optimization

Avoid premature optimization (runtime, memory).

3. Containers

A Note on Code Optimization

Avoid premature optimization (runtime, memory). Instead,

1. make it **work**
2. make it **reproducible**
3. **quantify performance** by profiling
4. make it **fast**

3. Containers

A Note on Code Optimization

Avoid premature optimization (runtime, memory). Instead,

1. make it **work**
   - write a simple implementation of what you want
   - don't try to be clever (but don't be stupid, either)
2. make it **reproducible**
3. **quantify performance** by profiling
4. make it **fast**

3. Containers

A Note on Code Optimization

Avoid premature optimization (runtime, memory). Instead,

1. make it **work**
   - write a simple implementation of what you want
   - don't try to be clever (but don't be stupid, either)
2. make it **reproducible**
   - make sure you have appropriate regression tests
   - this will simplify later optimization
3. **quantify performance** by profiling

4. make it **fast**

3. Containers

A Note on Code Optimization

Avoid premature optimization (runtime, memory). Instead,

1. make it **work**
   - write a simple implementation of what you want
   - don't try to be clever (but don't be stupid, either)
2. make it **reproducible**
   - make sure you have appropriate regression tests
   - this will simplify later optimization
3. **quantify performance** by profiling
   - consider typical use cases (and document them)
   - be surprised where time is spent (and where compilers have already done a good job)
4. make it **fast**

3. Containers

A Note on Code Optimization

Avoid premature optimization (runtime, memory). Instead,

1. make it **work**
   - write a simple implementation of what you want
   - don't try to be clever (but don't be stupid, either)
2. make it **reproducible**
   - make sure you have appropriate regression tests
   - this will simplify later optimization
3. **quantify performance** by profiling
   - consider typical use cases (and document them)
   - be surprised where time is spent (and where compilers have already done a good job)
4. make it **fast**
   - optimize implementation where needed
   - use regression tests to avoid breaking it

3. Containers

A Note on Code Optimization

Avoid premature optimization (runtime, memory). Instead,

1. make it **work**
   - write a simple implementation of what you want
   - don't try to be clever (but don't be stupid, either)
2. make it **reproducible**
   - make sure you have appropriate regression tests
   - this will simplify later optimization
3. **quantify performance** by profiling
   - consider typical use cases (and document them)
   - be surprised where time is spent (and where compilers have already done a good job)
4. make it **fast**
   - optimize implementation where needed
   - use regression tests to avoid breaking it

"More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity." (William Wulf)

3. Containers

Sets

`std::set<DataType>`

- mathematical set: objects contained once
- internally sorted according to comparison
- no direct (write) access
- fast find
- extensions
  - `std::multiset<DataType>` can contain multiple instances
  - `std::unordered_set<DataType>` hashed, not sorted

3. Containers

Set 1

```
#include<set>
// ...
std::set<short int> mySet;
for ( int i = 0; i < 25; ++i ) {
  const short int val = ( 12 * i + 2 ) % 26;
  mySet.insert ( val );
}
std::cout << ”mySet␣is␣of␣size␣” << mySet.size() << std::endl;
// note: no random access

std::cout << ”␣Removed␣” << mySet.erase ( 12 ) << ”␣times␣entry␣12.” << std::endl
          << ”␣Removed␣” << mySet.erase ( 13 ) << ”␣times␣entry␣13.” << std::endl;
```

3. Containers

Set 2

```
// iterate over set, as before
for ( auto it = mySet.begin(); it != mySet.end(); ++it ) {
  std::cout << *it << std::endl;
}

std::set<short int>::iterator pos = mySet.find ( 16 );
std::cout << "16 found: " << *pos << std::endl;

std::set<short int>::iterator otherPos = mySet.find ( 23 );
if ( otherPos == mySet.end() ) {
  std::cout << "the following results in undefined behavior" << std::endl;
  std::cout << "23 not found: " << *otherPos << std::endl; // dereferencing undefined!
}
```

⇝ **STLSet_Basic.cpp**

3. Containers

Unordered Set and Multiset 1

```
template<typename SetType>
void fillAndPrint ( ) {
  SetType mySet;

  for ( int i = 0; i < 25; ++i ) {
    const short int val = ( 12 * i + 2 ) % 26;
    std::cout << ”inserting␣” << val << std::endl;
    mySet.insert ( val );
  }

  std::cout << ”mySet␣is␣of␣size␣” << mySet.size() << std::endl;

  for ( auto it = mySet.begin(); it != mySet.end(); ++it ) {
    std::cout << *it << std::endl;
  }
  std::cout << std::endl;
}
```

3. Containers

Unordered Set and Multiset 2

```
#include<set>
#include<unordered_set>
// ...
std::cout << ”Multiset:␣” << std::endl;
fillAndPrint < std::multiset<short int> > ();

std::cout << ”Unordered␣Set:␣” << std::endl;
fillAndPrint < std::unordered_set<short int> > ();

std::cout << ”Unordered␣Multiset:␣” << std::endl;
fillAndPrint < std::unordered_multiset<short int> > ();
```

Unordered Set: More about Hashing

```
#include<set>
#include<unordered_set>
// ...

std::unordered_set<std::string> myUnorderedSet;

for ( int i = 0; i < 25; ++i ) {
  std::string number = ”Number␣” + std::to_string ( i );
  std::cout << ”inserting␣” << number << ”␣with␣hash␣”
            << myUnorderedSet.hash_function() ( number ) << std::endl;
  myUnorderedSet.insert ( number );
}

std::cout << ”Load␣factor␣is␣” << myUnorderedSet.load_factor()
          << ”,␣there␣are␣” << myUnorderedSet.bucket_count() << ”␣buckets.” << std::endl;

std::string num23 ( ”Number␣23” );
std::cout << num23 << ”␣is␣in␣bucket␣” << myUnorderedSet.bucket ( num23 ) << std::endl;
```

⇝ STLSet_Extensions.cpp

3. Containers

Maps

`std::map<KeyType, ValueType>`

- “dictionary”, values for unique keys
- internally sorted according to comparison
- fast lookup
- access via `operator[]` somewhat tricky
- extensions
  - `std::multimap<KeyType, ValueType>` can contain multiple value per key
  - `std::unordered_map<KeyType, ValueType>` hashed, not sorted

3. Containers

Map 1

```
#include<map>
#include<string>
// ...
std::map<short int, std::string> myMap;
myMap[1] = ”One”; // inserting elements works via operator[]
myMap.insert ( std::pair<short int, std::string> ( 2, ”Two” ) );
myMap.insert ( std::make_pair ( 3, ”Three” ) );

std::cout << ”1:␣” << myMap[1] << std::endl; // access also works via operator[]

std::cout << ”4:␣” << myMap[4] << std::endl;
                  // there is no const access, default object created instead!

auto it = myMap.find ( 1 );
std::cout << it->first << ”␣” << it->second << std::endl; // iterator points to pair

it = myMap.find ( 5 );
std::cout << ( it == myMap.end() ) << std::endl;
```

3. Containers

Map 2

```
#include<map>
#include<string>
// ...
std::map< std::string, short int>
  myOtherMap = { { ”Zero␣”, 0 },
                 { ”One␣␣”, 1 },
// ...
                 { ”Eight”, 8 },
                 { ”Nine␣”, 9 } };
// alternative form of initialization

// this is how one can iterate over a map
for ( auto it = myOtherMap.begin(); it != myOtherMap.end(); ++it ) {
  std::cout << it->first << ”␣” << it->second << std::endl;
} // note: the map is sorted
```

⇝ STLMap_Basic.cpp

3. Containers

Map 3

```
#include<map>
#include<string>
// ...
std::map<short int, std::string> myMap;
myMap[1] = ”One”; // inserting elements works via operator[]
myMap[2] = ”Two”;
std::string dwarf = myMap[3];

// access including existence check and const access via at()
for ( short int i = 0; i < 5; ++i ) {
  std::string myString;
  try {
    myString = myMap.at ( i );
  } catch ( std::exception &ex ) {
    std::cout << ”exception␣caught:␣” << ex.what() << std::endl;
    myString = ”not␣found”;
  }
  std::cout << i << ”:␣” << myString << std::endl;
}
```

Unordered Map

```
#include<unordered_map>
#include<string>
// ...
std::unordered_map< std::string, short int>
  myOtherMap = { { ”Zero␣”, 0 },
                 { ”One␣␣”, 1 },
                 { ”Two␣␣”, 2 },
                 { ”Three”, 3 },
                 { ”Four␣”, 4 },
                 { ”Five␣”, 5 },
                 { ”Six␣␣”, 6 },
                 { ”Seven”, 7 },
                 { ”Eight”, 8 },
                 { ”Nine␣”, 9 } };
// alternative form of initialization

// iteration as usual (for maps)
for ( auto it = myOtherMap.begin(); it != myOtherMap.end(); ++it ) {
  std::cout << it->first << ”␣” << it->second << std::endl;
}
```

3. Containers

Multimap 1

```
#include<map>
#include<string>
// ...
std::multimap<unsigned int, std::string>
  myMap = { { 1, ”one” },
            { 1, ”eins” },
// ...
            { 3, ”three” },
            { 3, ”drei” } };
// no access via operator[]
for ( auto it = myMap.begin(); it != myMap.end(); ++it ) {
  std::cout << it->first << ”␣” << it->second << std::endl;
}
```

Multimap 2

```
std::cout << ”number␣of␣occurences␣of␣1␣and␣4:␣”
          << myMap.count ( 1 ) << ”␣” << myMap.count ( 4 ) << std::endl;

// here’s how to iterate over values with the same key
typedef std::multimap<unsigned int, std::string>::const_iterator citType;
std::pair< citType, citType >
  range = myMap.equal_range ( 2 );
for ( citType it = range.first; it != range.second; ++it ) {
  std::cout << it->first << ”␣” << it->second << std::endl;
}
```

⇝ STLMap_Advanced.cpp

Caution: Set and Map with Floating Point Numbers

```
const double
  oneThirdA = 1.0/3.0f,
  oneThirdB = A::getOneOverThirty() / 2 + 9.5f * A::getOneOverThirty(),
  oneThirdC = 10.0f * A::getOneOverThirty();

std::map<double, std::string> myMap;
myMap [ oneThirdA ] = ”one␣third␣A”;
myMap [ oneThirdB ] = ”one␣third␣B”;

std::cout << ”should␣be␣oneThird␣A␣or␣B:␣”
          << myMap [ 10.0f * A::getOneOverThirty() ] << std::endl;

myMap [ oneThirdC ] = ”one␣third␣C”;

for ( auto it = myMap.begin(); it != myMap.end(); ++it ) {
  std::cout << it->first << ”␣” << it->second << ”␣”;
  printAsHex ( it->first );
  std::cout << std::endl;
}
```

⇝ STLMap_HowNotTo.cpp

4. Iterators

Contents

Iterators

- iterators can be used similarly for different containers

Iterators

- iterators can be used similarly for different containers
- replacable by pointers (see sorting example later) or anything that behaves like an iterator

Iterators

- iterators can be used similarly for different containers
- replacable by pointers (see sorting example later) or anything that behaves like an iterator
- unidirectional vs. bidirectional vs. random access iterators

Iterators

- iterators can be used similarly for different containers
- replacable by pointers (see sorting example later) or anything that behaves like an iterator
- unidirectional vs. bidirectional vs. random access iterators
  - `++it`, `--it`, `itA < itB`,

Iterators

- iterators can be used similarly for different containers
- replacable by pointers (see sorting example later) or anything that behaves like an iterator
- unidirectional vs. bidirectional vs. random access iterators
  - `++it`, `--it`, `itA < itB`,
- `reverse_iterator` vs. `iterator`

4. Iterators

Iterators

- iterators can be used similarly for different containers
- replacable by pointers (see sorting example later) or anything that behaves like an iterator
- unidirectional vs. bidirectional vs. random access iterators
  - `++it`, `--it`, `itA < itB`,
- `reverse_iterator` vs. `iterator`
  - `for ( std::vector<...>::reverse_iterator it = vec.rbegin(); it != myVec.rend(); ++it )`

Iterators

- iterators can be used similarly for different containers
- replacable by pointers (see sorting example later) or anything that behaves like an iterator
- unidirectional vs. bidirectional vs. random access iterators
  - `++it`, `--it`, `itA < itB`,
- `reverse_iterator` vs. `iterator`
  - `for ( std::vector<...>::reverse_iterator it = vec.rbegin(); it != myVec.rend(); ++it )`
- `const_iterator` vs. `iterator`

Iterators

- iterators can be used similarly for different containers
- replacable by pointers (see sorting example later) or anything that behaves like an iterator
- unidirectional vs. bidirectional vs. random access iterators
  - `++it`, `--it`, `itA < itB`,
- `reverse_iterator` vs. `iterator`
  - `for ( std::vector<...>::reverse_iterator it = vec.rbegin(); it != myVec.rend(); ++it )`
- `const_iterator` vs. `iterator`
  - prevent modifying container contents

Iterators

- iterators can be used similarly for different containers
- replacable by pointers (see sorting example later) or anything that behaves like an iterator
- unidirectional vs. bidirectional vs. random access iterators
  - `++it`, `--it`, `itA < itB`,
- `reverse_iterator` vs. `iterator`
  - `for ( std::vector<...>::reverse_iterator it = vec.rbegin(); it != myVec.rend(); ++it )`
- `const_iterator` vs. `iterator`
  - prevent modifying container contents
  - `const_iterator`s can still be incremented

Iterators

- iterators can be used similarly for different containers
- replacable by pointers (see sorting example later) or anything that behaves like an iterator
- unidirectional vs. bidirectional vs. random access iterators
  - `++it`, `--it`, `itA < itB`,
- `reverse_iterator` vs. `iterator`
  - `for ( std::vector<...>::reverse_iterator it = vec.rbegin(); it != myVec.rend(); ++it )`
- `const_iterator` vs. `iterator`
  - prevent modifying container contents
  - `const_iterator`s can still be incremented
- iterators (pointers, references) may survive container modification or not

5. Algorithms and Utilities

Contents

5. Algorithms and Utilities

Debugging Support

compiler define `_GLIBCXX_DEBUG`:

```
g++ -std=c++0x -g -D_GLIBCXX_DEBUG
  -lm -o STLDebugging.bin_d STLDebugging.cpp
```

```
#include<cstdlib>
#include<iostream>
#include<vector>

int main ( const int, const char** const ) {
  const unsigned int size = 2;
  std::vector<unsigned int> myVec(size);
  myVec.clear();
  myVec[1] = 42;
  // myVec.at(1) = 42;
  return ( EXIT_SUCCESS );
}
```

⇝ **STLDebugging.cpp**

5. Algorithms and Utilities

Predicates

- functions returning Boolean value
- always return same value (no internal state)
- e.g., comparison for sorting

```
bool isDivisibleByFour ( const int &Arg ) {
  return ( ( ( Arg % 4 ) == 0 ) );
}
```

5. Algorithms and Utilities

Functors (Function Objects)

- object with `operator()`
- behave like a function, but more general

```
class UniqueNumberGenerator {
public:
  UniqueNumberGenerator() : _num ( 0 ) {
  }

  int operator()() {
    _num += 3;
    return ( _num );
  }

private:
  int _num;
};
```

5. Algorithms and Utilities

Algorithms

- some examples of container methods seen above
- STL offers **generic** algorithms independent of STL data structures
- access via iterators or iterator-likes

5. Algorithms and Utilities

Pairs

```
#include<list>
#include<limits>

template<typename T>
std::pair<T, unsigned int> findMaximumAndItsPosition ( std::list<T> &Arg ) {
  std::pair<T, unsigned int> res ( - std::numeric_limits<T>::infinity(),
                                   std::numeric_limits<unsigned int>::max() );
  unsigned int i = 0;
  for ( typename std::list<T>::const_iterator it = Arg.begin();
        it != Arg.end(); ++it, ++i ) {
    if ( (*it) > (res.first) ) {
      res.first  = (*it);
      res.second = i;
    }
  }
  return ( res );
}
// ...
std::pair<double, unsigned int> maxAndPos = findMaximumAndItsPosition ( myList );
```

⇝ **STLUtilities_Pair.cpp**

5. Algorithms and Utilities

Min, Max, Swap

```
#include<cstdlib>
#include<iostream>

int main ( const int, const char** const ) {
  int a = 23, b = 42;
  std::cout << ”Maximum␣is␣” << std::max ( a, b) << std::endl;
  std::cout << ”Maximum␣is␣” << std::min ( a, b) << std::endl;

  std::swap ( a, b );
  std::cout << ”Now␣a␣=␣” << a << ”,␣b␣=␣” << b << std::endl;

  return ( EXIT_SUCCESS );
}
```

⇝ **STLUtilities_MinMaxSwap.cpp**

5. Algorithms and Utilities

Maximum with Comparator

```
int weightedTimeSinceMeal ( const int Hour ) {
  static const int hPD = 24; // hours per day

  const int hour = Hour % hPD;
       if ( hour <  8 ) { return ( ( ( hPD + hour - 20 ) % hPD ) / 2 ); }
  else if ( hour < 12 ) { return ( 2 * ( hour - 8 ) );                  }
  else if ( hour < 20 ) { return ( ( hour - 12 ) );                     }
  else {                  return ( ( ( hour - 20 ) % hPD ) / 2 );       }

}

bool hungerCompare ( const int HourA, const int HourB ) {
  return ( weightedTimeSinceMeal ( HourA ) < weightedTimeSinceMeal ( HourB ) );
}
// ...

const int timeA = 11, timeB = 14;

std::cout << "Comparing " << timeA << " and " << timeB
          << ", more hungry at " << std::max ( 11, 14, hungerCompare ) << std::endl;
```

⇝ STLUtilities_Compare.cpp

5. Algorithms and Utilities

Sorting 1

```
#include<vector>
#include<set>
#include<algorithm>

const size_t mySize = 8;
const float myArray[mySize] = { 0.2, 0.8, 1.9, 1.7, 0.3, 1.1, 1.5, 0.4 };

std::vector<float> myVector ( myArray, myArray+mySize );
std::sort ( myVector.begin(), myVector.end() );

std::sort ( myVector.begin(), myVector.end(), std::greater<float>() );

std::set<float, std::greater<float> > mySet ( myArray, myArray+mySize );
```

5. Algorithms and Utilities

Sorting 2

```
#include<vector>
#include<algorithm>

template<typename FloatDataType>
struct lessAsInt {
  bool operator() ( const FloatDataType A, const FloatDataType B ) const {
    return ( ( static_cast<int> ( A ) < static_cast<int> ( B ) ) );
  }
};

std::vector<float> myVectorB ( myArray, myArray+mySize );
std::sort ( myVectorB.begin(), myVectorB.end(), lessAsInt<float>() );

std::sort ( myArray, myArray+mySize ); // sort C array
```

⇝ STLAlgorithms_Sorting.cpp

5. Algorithms and Utilities

Generators 1

```
int consecutiveNumber() {
  static int counter = 0;
  return ( counter++ );
}

class UniqueNumberGenerator {
public:
  UniqueNumberGenerator() : _num ( 0 ) {
  }

  int operator()() {
    _num += 3;
    return ( _num );
  }

private:
  int _num;
};
```

5. Algorithms and Utilities

Generators 2, Transform, Accumulate

```
#include<vector>
#include<algorithm>
#include<functional>
// ...
const size_t mySize = 10;
std::vector<int> myVectorA ( mySize );
std::generate ( myVectorA.begin(), myVectorA.end(), consecutiveNumber );

std::vector<int> myVectorB ( mySize );
std::generate ( myVectorB.begin(), myVectorB.end(), UniqueNumberGenerator() );

std::vector<int> myVectorC ( mySize );
std::transform ( myVectorA.begin(), myVectorA.end(),
                 myVectorB.begin(), myVectorC.rbegin(), std::plus<int>() );
const int
  init = 50,
  result = std::accumulate ( myVectorA.begin(), myVectorA.end(),
                             init, std::minus<int>() );
std::cout << "result␣=␣" << result << std::endl;
```

⇝ **STLAlgorithms_Generate.cpp**

Find and Iterator Arithmetics

```
#include<vector>
#include<algorithm>

typedef unsigned short ushort;
const size_t mySize = 10;

// std::find works with C arrays as well as STL data structures

const ushort myArray[mySize] = { 0, 2, 4, 6, 8, 10, 12, 14, 12, 12 };
const ushort* p = std::find ( myArray, myArray+mySize, 12 );
if ( ( p - myArray ) < mySize ) {
  std::cout << "Found " << *p << " at position " << p - myArray << std::endl;
} // else, p == myArray+mySize indicating not found

std::vector<ushort> myVector ( myArray, myArray+mySize );
std::vector<ushort>::iterator
  pos = std::find ( myVector.begin(), myVector.end(), 14 );
if ( pos != myVector.end() ) {
  std::cout << "Found " << *pos << " at position "
            << pos - myVector.begin() << std::endl; // iterator difference
} // else not found
```

5. Algorithms and Utilities

Count, Shuffle and Minmax_Element

```
#include<ctime>
#include<random>
// ...

const int numCount = std::count ( myArray, myArray+mySize, 12 );
std::cout << ”12␣is␣contained␣” << numCount << ”␣times.” << std::endl;

unsigned int mySeed = static_cast<int> ( std::time(0) );
std::shuffle ( myVector.begin(), myVector.end(), std::mt19937 ( mySeed ) );

std::pair< std::vector<ushort>::iterator,
           std::vector<ushort>::iterator >
  minMax = std::minmax_element ( myVector.begin(), myVector.end() );

std::cout << ”Minimum␣of␣” << * ( minMax.first  )
          <<  ”␣at␣” << minMax.first  - myVector.begin() << ”,␣”
          << ”maximum␣of␣” << * ( minMax.second )
          <<  ”␣at␣” << minMax.second - myVector.begin() << std::endl << std::endl;
```

similarly: **std::count_if**, **std::min_element**, **std::max_element**

5. Algorithms and Utilities

Partitioning and Binary Search

```
bool isDivisibleByFour ( const ushort &Arg ) {
  return ( ( ( Arg % 4 ) == 0 ) );
}

bool veryExpensiveComparison ( const ushort &A, const ushort &B ) {
  std::cout << "VERY EXPENSIVE COMPARISON" << std::endl;
  return ( ( A < B ) );
}

// ...

std::vector<ushort> myVector2 ( myVector );

std::partition ( myVector2.begin(), myVector2.end(), isDivisibleByFour );

std::stable_partition ( myVector.begin(), myVector.end(), isDivisibleByFour );

bool found = std::binary_search ( myVector.begin(), myVector.end(),
                                  14, veryExpensiveComparison );
```

⇝ STLAlgorithms_Misc.cpp

6. Summary

1. Introduction
2. Concepts
3. Containers
4. Iterators
5. Algorithms and Utilities

Contact: Ole Schwen ‹ole.schwen@mevis.fraunhofer.de›

6. Summary

»Pinguine gibt es diesmal keine. Versprochen!«

Last time the organizers promised this, it didn't work …

6. Summary

»Pinguine gibt es diesmal keine. Versprochen!«

Last time the organizers promised this, it didn't work …