The STL design: an overview

HCM City C++ users meetup

Germán Diago Gómez

May 22nd, 2016

Goals of this talk

- introduce STL fundamental ideas

Goals of this talk

- introduce STL fundamental ideas
- understand the design choices that drove its current design

Goals of this talk

- introduce STL fundamental ideas
- understand the design choices that drove its current design
- light introduction to each of its fundamental parts

Goals of this talk

- introduce STL fundamental ideas
- understand the design choices that drove its current design
- light introduction to each of its fundamental parts

Goals of this talk

- introduce STL fundamental ideas
- understand the design choices that drove its current design
- light introduction to each of its fundamental parts

Non-goals

- not a detailed, fine-grained overview

Non-goals

- not a detailed, fine-grained overview
- not lots of examples

Non-goals

- not a detailed, fine-grained overview
- not lots of examples
  - due to time constraints

STL

- STL stands for Standard Template Library

STL

- STL stands for Standard Template Library
- it is a subset of the C++ Standard Library

STL

- STL stands for Standard Template Library
- it is a subset of the C++ Standard Library
- its main author is Alexander Stepanov

STL

- STL stands for Standard Template Library
- it is a subset of the C++ Standard Library
- its main author is Alexander Stepanov
- got accepted into ISO C++98 standard library

Design

- general

Design

- general
- allows a high degree of customization through composition

Design

- general
- allows a high degree of customization through composition
- extensible: you can add new components to the framework

Design

- general
- allows a high degree of customization through composition
- extensible: you can add new components to the framework
- reusable without loss of performace compared to hand-written code

Design

- general
- allows a high degree of customization through composition
- extensible: you can add new components to the framework
- reusable without loss of performace compared to hand-written code
- intensive use of generic programming

STL main components

- containers

STL main components

- containers
- iterators

STL main components

- containers
- iterators
- algorithms

STL main components

- containers
- iterators
- algorithms
- utilities (not the focus of this talk)

Why generic programming

- allows compile-time polymorphism combined with overloading

Why generic programming

- allows compile-time polymorphism combined with overloading
- allows customized code generation at compile-time

Why generic programming

- allows compile-time polymorphism combined with overloading
- allows customized code generation at compile-time
- allows choose best optimization based on overloading, which is a compile-time mechanism

Why generic programming

- allows compile-time polymorphism combined with overloading
- allows customized code generation at compile-time
- allows choose best optimization based on overloading, which is a compile-time mechanism
- allows high degree of customization, for both algorithms and data structures without loss of performance

Why generic programming (2)

- types can retroactively model a concept

Why generic programming (2)

- types can retroactively model a concept
  - example: array pointers are iterators

Why generic programming (2)

- types can retroactively model a concept
  - example: array pointers are iterators
- OO alternative: uses run-time polymorphism and dispatch

Why generic programming (2)

- types can retroactively model a concept
  - example: array pointers are iterators
- OO alternative: uses run-time polymorphism and dispatch
  - performance penalty

Concepts briefly explained

- Concepts are the foundation of generic programming

Concepts briefly explained

- Concepts are the foundation of generic programming
- a Concept is a set of requirements that a type must meet

Concepts briefly explained

- Concepts are the foundation of generic programming
- a Concept is a set of requirements that a type must meet
  - the syntactic requirements that a type must meet

Concepts briefly explained

- Concepts are the foundation of generic programming
- a Concept is a set of requirements that a type must meet
  - the syntactic requirements that a type must meet
  - the semantic requirements that these syntactic requirements must meet

Concepts briefly explained

- Concepts are the foundation of generic programming
- a Concept is a set of requirements that a type must meet
  - the syntactic requirements that a type must meet
  - the semantic requirements that these syntactic requirements must meet
  - optionally, associated types for that concept

Concepts briefly explained

- Concepts are the foundation of generic programming
- a Concept is a set of requirements that a type must meet
  - the syntactic requirements that a type must meet
  - the semantic requirements that these syntactic requirements must meet
  - optionally, associated types for that concept
  - algorithms: include the big O cost associated to them

Concepts briefly explained

- Concepts are the foundation of generic programming
- a Concept is a set of requirements that a type must meet
  - the syntactic requirements that a type must meet
  - the semantic requirements that these syntactic requirements must meet
  - optionally, associated types for that concept
  - algorithms: include the big O cost associated to them
- In C++ these are the requirements that your template parameters need to fullfill. They are stated in the documentation, since there is no language support (yet) for Concepts

Concepts briefly explained (2)

- similar to the way a virtual function sets requirements on derived types

Concepts briefly explained (2)

- similar to the way a virtual function sets requirements on derived types
  - but at compile-time

Concepts briefly explained (2)

- similar to the way a virtual function sets requirements on derived types
  - but at compile-time
- when a type fullfills the requirements of a concept we say it models that concept

Concepts briefly explained (2)

- similar to the way a virtual function sets requirements on derived types
  - but at compile-time
- when a type fullfills the requirements of a concept we say it models that concept
  - similar to when we say a derived class implements requirements from a base class

Concepts briefly explained (2)

- similar to the way a virtual function sets requirements on derived types
  - but at compile-time
- when a type fullfills the requirements of a concept we say it models that concept
  - similar to when we say a derived class implements requirements from a base class
- types do not need to inherit from any class to model a Concept $\rightarrow$ more loosely coupled

Concepts example: LessThanComparable

- syntactic requirements

Concepts example: LessThanComparable

- syntactic requirements
  - your type must work with `operator<` and return a type convertible to bool

Concepts example: LessThanComparable

- syntactic requirements
  - your type must work with `operator<` and return a type convertible to bool
- semantic requirements for `LessThanComparable operator<`

Concepts example: LessThanComparable

- syntactic requirements
  - your type must work with `operator<` and return a type convertible to bool
- semantic requirements for `LessThanComparable operator<`
  - $\forall a, !(a < a)$

Concepts example: LessThanComparable

- syntactic requirements
  - your type must work with `operator<` and return a type convertible to bool
- semantic requirements for `LessThanComparable operator<`
  - $\forall a, !(a < a)$
  - $a < b \iff !(b < a)$

Concepts example: LessThanComparable

- syntactic requirements
  - your type must work with `operator<` and return a type convertible to bool
- semantic requirements for `LessThanComparable operator<`
  - $\forall a, !(a < a)$
  - $a < b \iff !(b < a)$
  - $a < b \land b < c \implies a < c$

Concepts example: LessThanComparable

- syntactic requirements
  - your type must work with `operator<` and return a type convertible to bool
- semantic requirements for `LessThanComparable operator<`
  - $\forall a, !(a < a)$
  - $a < b \iff !(b < a)$
  - $a < b \land b < c \implies a < c$
- associated types (optionally)

More Concepts examples

- BinaryPredicate
- DefaultConstructible
- Many others

Introduction

Containers allow you to store collections of data in different data structures that best suite your needs.

STL containers classification

- sequence containers

STL containers classification

- sequence containers
  - `std::array<T>`, `std::vector<T>`, `std::list<T>`...

STL containers classification

- sequence containers
  - `std::array<T>`, `std::vector<T>`, `std::list<T>`...
- associative containers

STL containers classification

- sequence containers
  - `std::array<T>`, `std::vector<T>`, `std::list<T>`...
- associative containers
  - ordered → `std::set`...

STL containers classification

- sequence containers
  - `std::array<T>`, `std::vector<T>`, `std::list<T>`...
- associative containers
  - ordered → `std::set`...
  - unordered → `std::unordered_map`...

STL containers classification

- sequence containers
  - `std::array<T>`, `std::vector<T>`, `std::list<T>`...
- associative containers
  - ordered → `std::set`...
  - unordered → `std::unordered_map`...
- container adaptors

STL containers classification

- sequence containers
  - `std::array<T>`, `std::vector<T>`, `std::list<T>`...
- associative containers
  - ordered → `std::set`...
  - unordered → `std::unordered_map`...
- container adaptors
  - `std::priority_queue`, `std::stack`...

STL containers traversal

- depends on the kind iterators provided (more later)

STL containers traversal

- depends on the kind iterators provided (more later)
- random access, bidirectional, forward containers, etc.

Introduction

- Iterators are used to traverse ranges

Introduction

- Iterators are used to traverse ranges
- Iterators are an abstraction for positions

Introduction

- Iterators are used to traverse ranges
- Iterators are an abstraction for positions
  - they are modeled after C pointers in C++'s concrete case

Introduction

- Iterators are used to traverse ranges
- Iterators are an abstraction for positions
  - they are modeled after C pointers in C++'s concrete case
- you can do a traversal with two iterators

Introduction

- Iterators are used to traverse ranges
- Iterators are an abstraction for positions
  - they are modeled after C pointers in C++'s concrete case
- you can do a traversal with two iterators
- they are not the same as an iterator in other languages

Introduction

- Iterators are used to traverse ranges
- Iterators are an abstraction for positions
  - they are modeled after C pointers in C++'s concrete case
- you can do a traversal with two iterators
- they are not the same as an iterator in other languages
  - Java/C# iterator similar to a pair of iterators in C++

Introduction

- Iterators are used to traverse ranges
- Iterators are an abstraction for positions
  - they are modeled after C pointers in C++'s concrete case
- you can do a traversal with two iterators
- they are not the same as an iterator in other languages
  - Java/C# iterator similar to a pair of iterators in C++
- a pair of iterators is usually called a range in C++

Iterator categories

- several kinds of iterators

Iterator categories

- several kinds of iterators
- iterators form a hierarchy of Concepts

Iterator categories

- several kinds of iterators
- iterators form a hierarchy of Concepts
  - not implemented as inheritance in the library

Iterator categories

- several kinds of iterators
- iterators form a hierarchy of Concepts
  - not implemented as inheritance in the library
  - they are Concepts

Iterator categories

- several kinds of iterators
- iterators form a hierarchy of Concepts
  - not implemented as inheritance in the library
  - they are Concepts
- `RandomAccessIterator` → `BidirectionalIterator` → `ForwardIterator` → `InputIterator`

Iterator categories

- several kinds of iterators
- iterators form a hierarchy of Concepts
  - not implemented as inheritance in the library
  - they are Concepts
- `RandomAccessIterator` → `BidirectionalIterator` → `ForwardIterator` → `InputIterator`
- `OutputIterator` is any iterator that can write to the element it points to

Iterator categories (2)

- every iterator category supports a set of operations

Iterator categories (2)

- every iterator category supports a set of operations
- `InputIterator`

Iterator categories (2)

- every iterator category supports a set of operations
- `InputIterator`
  - element access

Iterator categories (2)

- every iterator category supports a set of operations
- `InputIterator`
  - element access
  - increment iterator position by one

Iterator categories (2)

- every iterator category supports a set of operations
- `InputIterator`
  - element access
  - increment iterator position by one
  - single traversal

Iterator categories (2)

- every iterator category supports a set of operations
- `InputIterator`
  - element access
  - increment iterator position by one
  - single traversal
- `ForwardIterator`

Iterator categories (2)

- every iterator category supports a set of operations
- `InputIterator`
  - element access
  - increment iterator position by one
  - single traversal
- `ForwardIterator`
  - `InputIterator` + support for multiple traversals

Iterator categories (2)

- every iterator category supports a set of operations
- `InputIterator`
  - element access
  - increment iterator position by one
  - single traversal
- `ForwardIterator`
  - `InputIterator` + support for multiple traversals
- `BidirectionalIterator`

Iterator categories (2)

- every iterator category supports a set of operations
- `InputIterator`
  - element access
  - increment iterator position by one
  - single traversal
- `ForwardIterator`
  - `InputIterator` + support for multiple traversals
- `BidirectionalIterator`
  - `ForwardIterator` + supports decrementing position by one

Iterator categories (2)

- every iterator category supports a set of operations
- `InputIterator`
  - element access
  - increment iterator position by one
  - single traversal
- `ForwardIterator`
  - `InputIterator` + support for multiple traversals
- `BidirectionalIterator`
  - `ForwardIterator` + supports decrementing position by one
- `RandomAccessIterator`

Iterator categories (2)

- every iterator category supports a set of operations
- `InputIterator`
  - element access
  - increment iterator position by one
  - single traversal
- `ForwardIterator`
  - `InputIterator` + support for multiple traversals
- `BidirectionalIterator`
  - `ForwardIterator` + supports decrementing position by one
- `RandomAccessIterator`
  - `BidirectionalIterator` + supports jumping to arbitrary position in $O(1)$

Iterator categories (2)

- every iterator category supports a set of operations
- `InputIterator`
  - element access
  - increment iterator position by one
  - single traversal
- `ForwardIterator`
  - `InputIterator` + support for multiple traversals
- `BidirectionalIterator`
  - `ForwardIterator` + supports decrementing position by one
- `RandomAccessIterator`
  - `BidirectionalIterator` + supports jumping to arbitrary position in $O(1)$
  - think of a pointer to C array position

Why are iterators important

- abstract traversal from the underlying data structure

Why are iterators important

- abstract traversal from the underlying data structure
- decouple algorithms implementation from data structure (later more)

Why are iterators important

- abstract traversal from the underlying data structure
- decouple algorithms implementation from data structure (later more)
- can implement algorithms based on its iterator category (later more)

Why are iterators important

- abstract traversal from the underlying data structure
- decouple algorithms implementation from data structure (later more)
- can implement algorithms based on its iterator category (later more)
  - containers just need to provide a begin/end sequence pair

Why are iterators important

- abstract traversal from the underlying data structure
- decouple algorithms implementation from data structure (later more)
- can implement algorithms based on its iterator category (later more)
  - containers just need to provide a begin/end sequence pair
- let you reduce implementation effort on algorithms

Introduction

- algorithms let you perform operations on your data

Introduction

- algorithms let you perform operations on your data
- STL algorithms design choices are a bit suprising at first

Algorithm design choices

- it was decided to implement algorithms as free functions, not as member functions

Algorithm design choices

- it was decided to implement algorithms as free functions, not as member functions
- it was decided that it would take `Iterator`s as input, instead of `Container`s

Algorithm design choices

- it was decided to implement algorithms as free functions, not as member functions
- it was decided that it would take `Iterator`s as input, instead of `Container`s
- consequence: algorithms are decoupled from containers

STL algorithms/iterators conventions

- algorithm specifications use half-open ranges for algorithms

STL algorithms/iterators conventions

- algorithm specifications use half-open ranges for algorithms
  - $[a, b)$

STL algorithms/iterators conventions

- algorithm specifications use half-open ranges for algorithms
  - $[a, b)$
  - a is included in the range. b is not included in the range

STL algorithms/iterators conventions

- algorithm specifications use half-open ranges for algorithms
  - $[a, b)$
  - a is included in the range. b is not included in the range
- containers provide `begin`, `end` and its variants

STL algorithms/iterators conventions

- algorithm specifications use half-open ranges for algorithms
  - $[a, b)$
  - a is included in the range. b is not included in the range
- containers provide `begin`, `end` and its variants
  - `begin` points to the first element of the container

STL algorithms/iterators conventions

- algorithm specifications use half-open ranges for algorithms
  - $[a, b)$
  - a is included in the range. b is not included in the range
- containers provide `begin`, `end` and its variants
  - `begin` points to the first element of the container
  - `end` points one past the last element of the container traversal

STL algorithms/iterators conventions

- algorithm specifications use half-open ranges for algorithms
  - $[a, b)$
  - a is included in the range. b is not included in the range
- containers provide `begin`, `end` and its variants
  - `begin` points to the first element of the container
  - `end` points one past the last element of the container traversal
- thus, passing `begin` and `end` to an algorithm will traverse the whole sequence

STL algorithms/iterators conventions

- algorithm specifications use half-open ranges for algorithms
  - $[a, b)$
  - a is included in the range. b is not included in the range
- containers provide `begin`, `end` and its variants
  - `begin` points to the first element of the container
  - `end` points one past the last element of the container traversal
- thus, passing `begin` and `end` to an algorithm will traverse the whole sequence
- some algorithms take three iterators: a range + the beginning of a second range where to put the output

STL algorithms/iterators conventions

- algorithm specifications use half-open ranges for algorithms
  - $[a, b)$
  - a is included in the range. b is not included in the range
- containers provide `begin`, `end` and its variants
  - `begin` points to the first element of the container
  - `end` points one past the last element of the container traversal
- thus, passing `begin` and `end` to an algorithm will traverse the whole sequence
- some algorithms take three iterators: a range + the beginning of a second range where to put the output
  - C++14: these algorithms, called "three-legged" algorithms, were added overloads to pass two ranges. Example: `std::equal`

STL algorithms/iterators conventions (2)

- no memory management in algorithms. Beware

STL algorithms/iterators conventions (2)

- no memory management in algorithms. Beware
  - you must have space in the target output range

STL algorithms/iterators conventions (2)

- no memory management in algorithms. Beware
  - you must have space in the target output range
    - example: `std::copy` must have enough space for the output

STL algorithms/iterators conventions (2)

- no memory management in algorithms. Beware
  - you must have space in the target output range
    - example: `std::copy` must have enough space for the output
  - `std::remove` algorithms do not resize your containers

STL algorithms/iterators conventions (2)

- no memory management in algorithms. Beware
  - you must have space in the target output range
    - example: `std::copy` must have enough space for the output
  - `std::remove` algorithms do not resize your containers
    - they just put the rubbish to remove at the end of your container. You must call `Container::erase`.

STL algorithms/iterators conventions (2)

- no memory management in algorithms. Beware
  - you must have space in the target output range
    - example: `std::copy` must have enough space for the output
  - `std::remove` algorithms do not resize your containers
    - they just put the rubbish to remove at the end of your container. You must call `Container::erase`.
- you can use iterator adaptors that make room for elements if you do not know how much space you will need. example: `back_insert_iterator`.

Extensibility and adaptation

- you can provide an algorithm based on iterator category: it will work on all existing containers that model at least that iterator category

Extensibility and adaptation

- you can provide an algorithm based on iterator category: it will work on all existing containers that model at least that iterator category
- if you provide your iterators for you container, it just works

Extensibility and adaptation

- you can provide an algorithm based on iterator category: it will work on all existing containers that model at least that iterator category
- if you provide your iterators for you container, it just works
- basic arrays also work with algorithms

Implementation effort reduction

- container-based implementation: N algorithms for M data structures

Implementation effort reduction

- container-based implementation: N algorithms for M data structures
  - $O(N \cdot M)$ implementations

Implementation effort reduction

- container-based implementation: N algorithms for M data structures
  - $O(N \cdot M)$ implementations
- iterator-based implementation: N algorithms independent of the data structure

Implementation effort reduction (2)

- STL $\approx$ 60 algorithms and 13 data structures (without counting adapters and native arrays)

Implementation effort reduction (2)

- STL $\approx$ 60 algorithms and 13 data structures (without counting adapters and native arrays)
  - container-based algorithms: $60 * 13 = 780$ implementations

Implementation effort reduction (2)

- STL $\approx$ 60 algorithms and 13 data structures (without counting adapters and native arrays)
  - container-based algorithms: $60 * 13 = 780$ implementations
  - iterator-based algorithms: 60 implementations, plus maybe some optimization overloads in cases where algorithms can be optimized for a specific iterator category

Implementation effort reduction (2)

- STL $\approx$ 60 algorithms and 13 data structures (without counting adapters and native arrays)
  - container-based algorithms: $60 * 13 = 780$ implementations
  - iterator-based algorithms: 60 implementations, plus maybe some optimization overloads in cases where algorithms can be optimized for a specific iterator category
    - remember that algorithms must be as good as hand-written algorithms for any use case

STL algorithm example

Utilities

Utilities include other useful components such as `std::pair`, `std::function`, `type_traits`, `functional` objects, `std::tuple`, `std::unique_ptr`, `std::shared_ptr`, etc.

Typical algorithm usage

1. Choose an algorithm.

Typical algorithm usage

1. Choose an algorithm.
2. Provide the iterators from your container(s). for the algorithm through `begin()` and `end()` functions

Typical algorithm usage

1. Choose an algorithm.
2. Provide the iterators from your container(s). for the algorithm through `begin()` and `end()` functions
3. Customize the behaviour of your algorithm through predicates/function objects.

Example exercise

Read all the data for the spanish fooball league. We store the data in a `league` class which is a collection of `football_match_result` records. The file is sorted by round. Every 10 matches it is a new round. Blank lines are accepted.

Example exercise

Read all the data for the spanish fooball league. We store the data in a `league` class which is a collection of `football_match_result` records. The file is sorted by round. Every 10 matches it is a new round. Blank lines are accepted.

File spanish_football_league2015-2016.txt

```
Real Madrid,2-1,Malaga
Rayo,0-2,Sevilla
...
```

Example exercise

Read all the data for the spanish fooball league. We store the data in a `league` class which is a collection of `football_match_result` records. The file is sorted by round. Every 10 matches it is a new round. Blank lines are accepted.

File spanish_football_league2015-2016.txt

```
Real Madrid,2-1,Malaga
Rayo,0-2,Sevilla
...
```

We will:

1. Get all matches for a given team

Example exercise

Read all the data for the spanish fooball league. We store the data in a `league` class which is a collection of `football_match_result` records. The file is sorted by round. Every 10 matches it is a new round. Blank lines are accepted.

File spanish_football_league2015-2016.txt

```
Real Madrid,2-1,Malaga
Rayo,0-2,Sevilla
...
```

We will:

1. Get all matches for a given team
2. Calculate the classification table

Example exercise

Read all the data for the spanish fooball league. We store the data in a `league` class which is a collection of `football_match_result` records. The file is sorted by round. Every 10 matches it is a new round. Blank lines are accepted.

File spanish_football_league2015-2016.txt

```
Real Madrid,2-1,Malaga
Rayo,0-2,Sevilla
...
```

We will:

1. Get all matches for a given team
2. Calculate the classification table
3. Calculate the matches a given team won as a visitor

Current status of future standardization

- added parallel algorithms to C++17
  - essentially, you do `std::sort(parallel::par, begin(v), end(v))` and enjoy performance gains automatically
- ranges-based STL, by Eric Niebler → Ranges v3 experimental implementation
  - you can do `std::sort(v)` and much more
  - lazy evaluation and composition
    - `v | view::transform([](auto const & elem) {...}) | view::remove_if([]...)`
  - library includes, besides ranges, `view`s and `action`s, but these are not proposed for standardization yet
- there is a TS for Concepts support in the language: Concepts TS latest draft
  - implementation available in GCC 6 with -fconcepts flag

Thank you