The C++ Standard Template Library (STL)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
The C++ Standard Template Library: Presentation Outline

• What is STL?
• Generic programming: Why use STL?
• Overview of STL concepts & features
  - e.g., helper class & function templates, containers, iterators, generic algorithms, function objects, adapters
• A complete STL example
• References for more information on STL
What is the C++ Standard Template Library (STL)?

STL is a subset of the Standard C++ library that provides a set of well-structured generic C++ components that work together in a seamless way.

See en.wikipedia.org/wiki/Standard_Template_Library
What is the C++ Standard Template Library (STL)?

- A collection of composable class & function templates
  - Helper class & function templates: operators, pair
  - Container & iterator class templates
  - Generic algorithms that operate over *iterators*
  - Functors (function objects)
  - adapters

Functors act on elements, holds data & does actions on them

Iterators are passed to perform a function

Container of data elements

Iterators are passed to perform an algorithm

Algorithm acts on elements

Functors are sometimes passed to algorithms & containers to direct their actions
What is the C++ Standard Template Library (STL)?

- Enables *generic programming* in C++

See [en.wikipedia.org/wiki/Generic_programming](en.wikipedia.org/wiki/Generic_programming)
What is the C++ Standard Template Library (STL)?

- Enables **generic programming** in C++
  - A programming paradigm in which algorithms are written in terms of **generic** types, which are instantiated when needed for **specific** types provided as parameters

```cpp
template <typename InputIterator, typename T>
InputIterator find
  (InputIterator first,
   InputIterator last,
   const T& value) {
  while (first != last
      && *first != value)
    ++first;
  return first;
}
```

```cpp
class vector<int> v {1, 2, 3, 4};
auto i = find(v.begin(), v.end(), 3);
```
What is the C++ Standard Template Library (STL)?

- Enables **generic programming** in C++
  - A programming paradigm in which algorithms are written in terms of *generic* types, which are instantiated when needed for *specific* types provided as parameters
  - Each generic algorithm can operate over *any iterator for which the necessary operations are provided*

```cpp
template <typename InputIterator, typename T>
InputIterator find
(InputIterator first, InputIterator last,
 const T& value) {
    while (first != last && *first != value)
        ++first;
    return first;
}
...
int a[] = {1, 2, 3, 4};
int *e = a + sizeof(a)/sizeof(*a);
auto i = find(a, e, 3);
```
What is the C++ Standard Template Library (STL)?

- Enables *generic programming* in C++
  - A programming paradigm in which algorithms are written in terms of *generic* types, which are instantiated when needed for *specific* types provided as parameters
  - Each generic algorithm can operate over *any iterator for which the necessary operations are provided*
  - Extensible: can support new algorithms, containers, iterators
Generic Programming: Why Use STL?

• Reuse: “write less, do more”
Generic Programming: Why Use STL?

- **Reuse: “write less, do more”**
  - STL hides complex, tedious, & error-prone details
    - e.g., dynamic memory management, complex data structures & algorithms, many optimizations, etc.

Programmers can thus focus on the (business) problem at hand
Generic Programming: Why Use STL?

- **Reuse: “write less, do more”**
  - STL hides complex, tedious, & error-prone details
  - Ensures *type-safe* plug compatibility between STL components

```cpp
vector<int> v{2, 4, 3, 5, 1};
list<int> l{5, 2, 1, 2, 3};

sort(v.begin(), v.end()); // Ok
sort(l.begin(), l.end()); // Error
```

C++ STL performs static type checking to enhance correctness & performance
Generic Programming: Why Use STL?

- Flexibility
Generic Programming: Why Use STL?

• **Flexibility**
  – Iterators decouple algorithms from containers

```cpp
vector<int> v{1, 2, 3, 4, 5};
list<int> l{5, 4, 3, 2, 1};

auto vi = find(v.begin(), v.end(), 5);
auto li = find(l.begin(), l.end(), 5);
```
Generic Programming: Why Use STL?

- **Flexibility**
  - Iterators decouple algorithms from containers
  - Unanticipated combinations easily supported
  - e.g., via adapters

```cpp
vector<int> v{1, 2, 3, 4, 5};
list<int> l{5, 4, 3, 2, 1};

template<typename T>
struct greater_than_5 {
    bool operator()(const T &i) {
        return i > 5;
    }
};

auto i = find_if(v.begin, v.end(), greater_than_5<>());
auto vi = find_if(v.begin(), v.end(),
                 bind(greater<>(), _1, 5));
auto li = find_if(l.begin(), l.end(),
                 not_fn(bind(greater<>(), _1, 5)));
```
Generic Programming: Why Use STL?

- Efficiency
Generic Programming: Why Use STL?

- **Efficiency**
  - Templates & inlining avoids virtual function overhead

```cpp
vector<int> v{2, 4, 3, 5, 1};
list<int> l{5, 2, 1, 2, 3};

sort(v.begin(), v.end());
l.sort();
```
Generic Programming: Why Use STL?

- **Efficiency**
  - Templates & inlining avoids virtual function overhead
  - Strict attention to time complexity of algorithms
Generic Programming with C++ Function & Class Templates

- C++ STL makes heavy use of function templates & class templates

See en.wikipedia.org/wiki/Template_(C++)
C++ STL makes heavy use of function templates & class templates

- **Function templates** are C++ functions that can operate on different data types without separate code for each of them

```cpp
template <typename InputIterator, typename T>
InputIterator find
(InputIterator first, InputIterator last,
 const T& value) {
    while (first != last && *first != value)
        ++first;
    return first;
}
```
C++ STL makes heavy use of function templates & class templates

- **Function templates** are C++ functions that can operate on different data types without separate code for each of them.
- **Class templates** define a family of classes parameterized by type or values.
- If a set of functions have the same functionality for different data types, this becomes a good class template.

```cpp
template <typename T, typename Container = deque<T>>
class stack {
    public:
        explicit stack(const Container&);
        bool empty() const;
        size_type size() const;
        value_type& top();
        const value_type& top() const;
        void push(const value_type& t);
        void pop();
    private:
        Container container_;  
        // ...
};
```