Standard Template Library (STL)

Dr. Anwar Shah, PhD, MBA(HR)

Assistant Professor in CS

FAST National University of Computer and Emerging Sciences CFD

Section Overview

The Standard Template Library

- What is the STL
- Generic programming/ Meta-programming
  - Preprocessor macros
  - Function templates
  - Class templates
- STL Containers
- STL Iterators
- STL Algorithms

- Array
- Vector
- Deque
- List and Forward List
- Set and Multi Set
- Map and Multi Map
- Stack and Queue
- Priority Queue
- Algorithms

What is STL?

What is the STL?

- A library of powerful, reusable, adaptable, generic classes and functions
- Implemented using C++ templates
- Implements common data structures and algorithms
- Huge class library!!
- Alexander Stepanov (1994)

Why use the STL?

- Assortment of commonly used containers
- Known time and size complexity
- Tried and tested – Reusability!!!
- Consistent, fast, and type-safe
- Extensible

Elements of the STL

- Containers
  - Collections of objects or primitive types
    (array, vector, deque, stack, set, map, etc.)

- Algorithms
  - Functions for processing sequences of elements from containers
    (find, max, count, accumulate, sort, etc.)

- Iterators
  - Generate sequences of element from containers
    (forward, reverse, by value, by reference, constant, etc.)

Elements of the STL

A simple example

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

std::vector<int> v {1,5,3};
```

Elements of the STL

A simple example – sort a vector

```
std::sort(v.begin(), v.end());

for (auto elem: v)
    std::cout << elem << std::endl;
```

```
1
3
5
```

Elements of the STL

A simple example – reverse a vector

```
std::reverse(v.begin(), v.end());

for (auto elem: v)
    std::cout << elem << std::endl;
```

```
5
3
1
```

Elements of the STL

A simple example - accumulate

```
int sum{};

sum = std::accumulate(v.begin(), v.end(), 0);
std::cout << sum << std::endl;

9    // 1+3+5
```

Types of Containers

- Sequence containers
  - array, vector, list, forward_list, deque
- Associative containers
  - set, multi set, map, multi map
- Container adapters
  - stack, queue, priority queue

Types of Iterators

- Input iterators – from the container to the program
- Output iterators – from the program to the container
- Forward iterators – navigate one item at a time in one direction
- Bi-directional iterators – navigate one item at a time both directions
- Random access iterators – directly access a container item

Types of Algorithms

- About 60 algorithms in the STL
- Non-modifying
- Modifying

Generic Macros

The Standard Template Library

Generic Programming with macros

- Generic programming
  "Writing code that works with a variety of types as arguments, as long as those argument types meet specific syntactic and semantic requirements", Bjarne Stroustrup

- Macros ***** beware *****

- Function templates

- Class templates

The Standard Template Library

Macros (`#define`)

- C++ preprocessor directives
- No type information
- Simple substitution

```
#define MAX_SIZE 100

#define PI  3.14159
```

The Standard Template Library

Macros (`#define`)

```
#define MAX_SIZE 100

#define PI 3.14159

if (num > MAX_SIZE)
    std::cout << "Too big";

double area = PI * r * r;
```

The Standard Template Library

Macros (#define)

```
//#define MAX_SIZE 100      // removed

//#define PI 3.14159        // removed

if (num > 100)
    std::cout << "Too big");

double area = 3.14159 * r * r;
```

The Standard Template Library

max function

- Suppose we need a function to determine the max of 2 integers

```
int max(int a, int b) {
   return (a > b) ? a : b;
}

int x = 100;
int y = 200;
std::cout << max(x, y);      // displays 200
```

The Standard Template Library

max function

- Now suppose we need to determine the max of 2 doubles, and 2 chars

```
int max(int a, int b) {
    return (a > b) ? a : b;
}

double max(double a, double b) {
    return (a > b) ? a : b;
}

char max(char a, char b) {
    return (a > b) ? a : b;
}
```

The Standard Template Library

Macros with argument s (`#define`)

- We can write a generic macro with arguments instead

```
#define MAX(a, b) ((a > b) ? a : b)

std::cout << MAX(10,20)     << std::endl;   // 20
std::cout << MAX(2.4, 3.5) << std::endl;   // 3.5
std::cout << MAX('A', 'C') << std::endl;   // C
```

The Standard Template Library

Macros with argument s (`#define`)

- We have to be careful with macros

```
#define SQUARE(a) a*a

result = SQUARE(5);          // Expect 25
result = 5*5;                // Get 25

result = 100/SQUARE(5);      // Expect 4
result = 100/5*5             // Get 100!
```

The Standard Template Library

Macros with argument s (`#define`)

```
#define SQUARE(a) ((a)*(a)) // note the parenthesis

result = SQUARE(5);          // Expect 25
result = ((5)*(5));          // Still Get 25

result = 100/SQUARE(5);      // Expect 4
result = 100/((5)*(5));      // Now we get 4!!
```

Generic Function Templates

The Standard Template Library

Generic Programming with function templates

What is a C++ Template?

- Blueprint
- Function and class templates
- Allow **plugging-in** any data type
- Compiler generates the appropriate function/class from the blueprint
- Generic programming / meta-programming

The Standard Template Library

Generic Programming with function templates

- Let's revisit the max function from the last lecture

```
int max(int a, int b) {
    return (a > b) ? a : b;
}

int x = 100;
int y = 200;
std::cout << max(x, y);      // displays 200
```

The Standard Template Library

max function

- Now suppose we need to determine the max of 2 doubles, and 2 chars

```
int max(int a, int b) {
    return (a > b) ? a : b;
}

double max(double a, double b) {
    return (a > b) ? a : b;
}

char max(char a, char b) {
    return (a > b) ? a : b;
}
```

The Standard Template Library

max function as a template function

- We can replace type we want to generalize with a name, say T
- But now this won't compile

```
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

The Standard Template Library

max function as a template function

- We need to tell the compiler this is a template function
- We also need to tell it that T is the template parameter

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

The Standard Template Library

max function as a template function

- We may also use **`class`** instead of **`typename`**

```
template <class T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

The Standard Template Library

max function as a template function

- Now the compiler can generate the appropriate function from the template
- Note, this happens at compile-time!

```
int a {10};
int b {20};

std::cout << max<int>(a, b);
```

The Standard Template Library

max function as a template function

- Many times the compiler can deduce the type and the template parameter is not needed
- Depending on the type of a and b, the compiler will figure it out

```
std::cout << max<double>(c, d);

std::cout << max(c, d);
```

The Standard Template Library

max function as a template function

- And we can use **almost** any type we need

```
char a {'A'};
char b {'Z'};

std::cout << max(a, b) << std::endl;
```

The Standard Template Library

max function as a template function

- Notice the type MUST support the > operator either natively or as an overloaded operator **(operator>)**

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

The Standard Template Library

max function as a template function

- The following will not compile unless `Player` overloads `operator>`

```
Player p1{"Hero", 100, 20};
Player p2{"Enemy", 99, 3};

std::cout << max<Player>(p1, p2);
```

The Standard Template Library

multiple types as template parameters

- We can have multiple template parameters
- And their types can be different

```
template <typename T1, typename T2>
void func(T1 a, T2 b) {
    std::cout << a << " " << b;
}
```

The Standard Template Library

multiple types as template parameters

- When we use the function we provide the template parameters
- Often the compiler can deduce them

```
func<int,double>(10, 20.2);

func('A', 12.4);
```

Generic Class Templates

The Standard Template Library

Generic Programming with class templates

What is a C++ **Class** Template?

- Similar to function template, but at the class level
- Allows **plugging-in** any data type
- Compiler generates the appropriate class from the blueprint

The Standard Template Library

Generic Programming with class templates

- Let's say we want a class to hold Items where the item has a name and an integer

```
class Item {
private:
    std::string name;
    int value;
public:
    Item(std::string name, int value)
       : name{name}, value{value}
    {}
    std::string get_name() const {return name; }
    int get_value() const { return value; }
};
```

The Standard Template Library

Generic Programming with class templates

- But we'd like our `Item` class to be able to hold any type of data in addition to the string
- We can't overload class names
- We don't want to use dynamic polymorphism

The Standard Template Library

Generic Programming with class templates

```
class Item {
private:
    std::string name;
    T value;
public:
    Item(std::string name, T value)
       : name{name}, value{value}
    {}
    std::string get_name() const {return name; }
    T get_value() const { return value; }
};
```

The Standard Template Library

Generic Programming with class templates

```
template <typename T>
   class Item {
   private:
        std::string name;
        T value;
   public:
        Item(std::string name, T value)
           : name{name}, value{value}
        {}
        std::string get_name() const {return name; }
        T get_value() const { return value; }
   };
```

The Standard Template Library

Generic Programming with class templates

```
Item<int> item1 {"Larry", 1};

Item<double> item2 {"House", 1000.0};

Item<std::string> item3 {"Frank", "Boss"};

std::vector<Item<int>> vec;
```

The Standard Template Library

Multiple types as template parameters

- We can have multiple template parameters
- An their types can be different

```
template <typename T1, typename T2>
struct My_Pair {
    T1 first;
    T2 second;
};
```

The Standard Template Library

Multiple types as template parameters

```
My_Pair <std::string, int> p1 {"Frank", 100};

My_Pair <int, double> p2 {124, 13.6};

std::vector<My_Pair<int, double>> vec;
```

The Standard Template Library

std::pair

```
#include <utility>

std::pair<std::string, int> p1 {"Frank", 100};

std::cout << p1.first;      // Frank
std::cout << p1.second;     // 100
```

STL Containers

The Standard Template Library

Containers

- Data structures that can store object of *almost* any type
  - Template-based classes
- Each container has member functions
  - Some are specific to the container
  - Others are available to all containers
- Each container has an associated header file
  - `#include <container_type>`

The Standard Template Library

Containers – common

Function	Description
`Default constructor`	Initializes an empty container
`Overloaded constructors`	Initializes containers with many options
`Copy constructor`	Initializes a container as a copy of another container
`Move constructor`	Moves existing container to new container
`Destructor`	Destroys a container
`Copy assignment (operator=)`	Copy one container to another
`Move assignment (operator=)`	Move one container to another
`size`	Returns the number of elements in the container
`empty`	Returns boolean – is the container empty?
`insert`	Insert an element into the container

The Standard Template Library

Containers – common

Function	Description
`operator< and operator<=`	Returns boolean - compare contents of 2 containers
`operator> and operator>=`	Returns boolean - compare contents of 2 containers
`operator== and operator!=`	Returns boolean - are the contents of 2 containers equal or not
`swap`	Swap the elements of 2 containers
`erase`	Remove element(s) from a container
`clear`	Remove all elements from a container
`begin and end`	Returns iterators to first element or end
`rbegin and rend`	Returns reverse iterators to first element or end
`cbegin and cend`	Returns constant iterators to first element or end
`crbegin and crend`	Returns constant reverse iterators to first element or end

The Standard Template Library

Container elements

What types of elements can we store in containers?

- A **copy** of the element will be stored in the container
  - All primitives OK

- Element should be
  - Copyable and assignable (copy constructor / copy assignment)
  - Moveable for efficiency (move Constructor / move Assignment)

- Ordered associative containers must be able to compare elements
  - `operator<, operator==`

STL Iterators

The Standard Template Library

Iterators

- Allows abstracting an arbitrary container as a sequence of elements
- They are objects that work like pointers by design
- Most container classes can be traversed with iterators

The Standard Template Library

Declaring iterators

- Iterators must be declared based on the container type they will iterate over

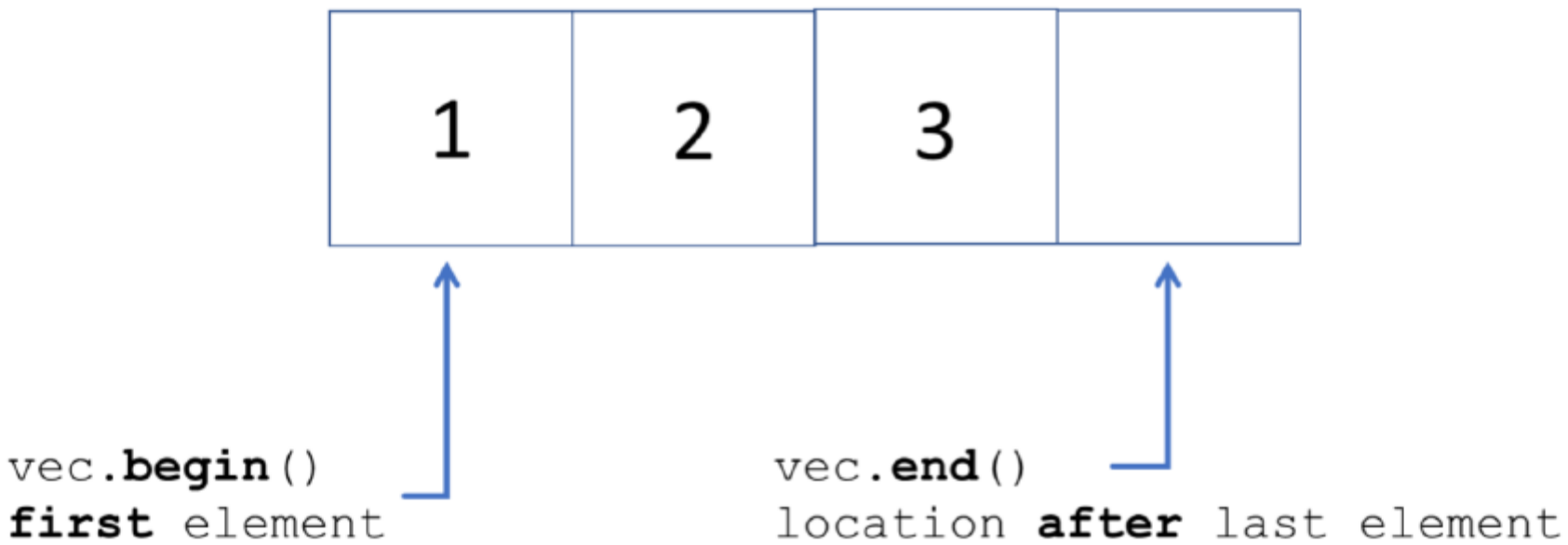
container_type::iterator_type *iterator_name*;

```
std::vector<int>::iterator it1;

std::list<std::string>::iterator it2;

std::map<std::string, std::string>::iterator it3;

std::set<char>::iterator it4;
```

The Standard Template Library

iterator begin and end methods

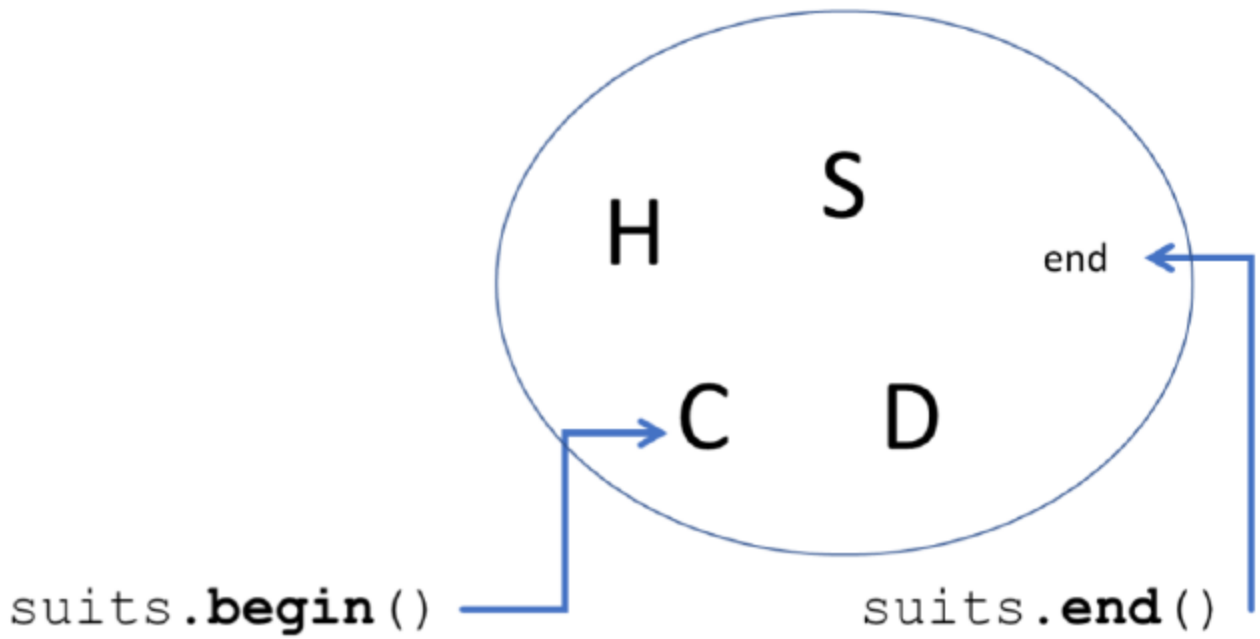
```
std::vector<int> vec {1,2,3};
```



The Standard Template Library

Declaring iterators

```
std::set<char> suits {'C','H','S','D'};
```



The Standard Template Library

Initializing iterators

```
std::vector<int> vec {1,2,3};

std::vector<int>::iterator it = vec.begin();

or

auto it = vec.begin();
```

The Standard Template Library

Operations with iterators `(it)`

Operation	Description	Type of Iterator
`++it`	Pre-increment	All
`it++`	Post-increment	All
`it = it1`	Assignment	All
`*it`	Dereference	Input and Output
`it->`	Arrow operator	Input and Output
`it == it1`	Comparison for equality	Input
`it != it1`	Comparison for inequality	Input
`--it`	Pre-decrement	Bidirectional
`it--`	Post-decrement	Bidirectional
`it + i, it += i` `it - i, it -= i`	Increment and decrement	Random Access
`it < it1, it <= it1` `it > it1, it >= it1`	Comparison	Random Access

The Standard Template Library

Using iterators – std::vector

```
std::vector<int> vec {1,2,3};

std::vector<int>::iterator it = vec.begin();

while (it != vec.end()) {
    std::cout << *it << " ";
    ++it;
}
// 1 2 3
```

The Standard Template Library

Using iterators – std::vector

```
std::vector<int> vec {1,2,3};

for (auto it = vec.begin(); it != vec.end(); it++) {
    std::cout << *it << " ";
}
// 1 2 3
```

- This is how the range-based for loop works

The Standard Template Library

Using iterators – std::set

```
std::set<char> suits {'C', 'H', 'S', 'D'};

auto it = suits.begin();

while (it != suits.end()) {
    std::cout << *it << " " << std::end;
    ++it;
}
// C H S D
```

The Standard Template Library

Reverse iterators

- Works in reverse
- Last element is the first and first is the last
- ++ moves backward, -- moves forward

```
std::vector<int> vec {1,2,3};
std::vector<int>::reverse_iterator it = vec.begin();
while (it != vec.end()) {
    std::cout << *it << " ";
    ++it;
}
// 3 2 1
```

The Standard Template Library

Other iterators

- `begin()` and `end()`
  - `iterator`
- `cbegin()` and `cend()`
  - `const_iterator`
- `rbegin()` and `rend()`
  - `reverse_iterator`
- `crbegin()` and `crend()`
  - `const_reverse_iterator`

STL Algorithms

The Standard Template Library

Algorithms

- STL algorithms work on sequences of container elements provided to them by an iterator
- STL has many common and useful algorithms
- Too many to describe in this section
  - http://en.cppreference.com/w/cpp/algorithm
- Many algorithms require extra information in order to do their work
  - Functors (function objects)
  - Function pointers
  - Lambda expressions (C++11)

The Standard Template Library

Algorithms and iterators

- `#include <algorithm>`

- Different containers support different types of iterators
  - Determines the types of algorithms supported

- All STL algorithms expect iterators as arguments
  - Determines the sequence obtained from the container

The Standard Template Library

Iterator invalidation

- Iterators point to container elements
- It's possible iterators become invalid during processing
- Suppose we are iterating over a vector of 10 elements
  - And we clear() the vector while iterating? What happens?
  - Undefined behavior – our iterators are pointing to invalid locations

The Standard Template Library

Example algorithm – `find` with primitive types

- The `find` algorithm tries to locate the first occurrence of an element in a container
- Lots of variations
- Returns an iterator pointing to the located element or `end()`

```
std::vector<int> vec {1,2,3};

auto loc = std::find(vec.begin(), vec.end(), 3);

if (loc != vec.end())                    // found it!
   std::cout << *loc << std::endl;  // 3
```

The Standard Template Library

Example algorithm – `find` with user-defined types

- Find needs to be able to compare object

- `operator==` is used and must be provided by your class

```
std::vector<Player> team { /* assume initialized */ }
Player p {"Hero", 100, 12};

auto loc = std::find(team.begin(), team.end(), p);

if (loc != vec.end())                    // found it!
   std::cout << *loc << std::endl;       // operator<< called
```

The Standard Template Library

Example algorithm – `for_each`

- `for_each` algorithm applies a function to each element in the iterator sequence

- Function must be provided to the algorithm as:
  - Functor (function object)
  - Function pointer
  - Lambda expression (C++11)

- Let's square each element

The Standard Template Library

`for_each` - using a functor

```
struct Square_Functor {
      void operator()(int x) {    // overload () operator
            std::cout << x * x << " ";
      }
};
Square_Functor square;      // Function object

std::vector<int> vec {1, 2, 3, 4};

std::for_each(vec.begin(), vec.end(), square);
// 1 4 9 16
```

The Standard Template Library

`for_each` - using a function pointer

```
void square(int x) {                    // function
      std::cout << x * x << " ";
}

std::vector<int> vec {1, 2, 3, 4};

std::for_each(vec.begin(), vec.end(), square);
// 1 4 9 16
```

The Standard Template Library

for_each - using a lambda expression

```
std::vector<int> vec {1, 2, 3, 4};

std::for_each(vec.begin(), vec.end(),
      [](int x) { std::cout << x * x << " "; }) // lambda

// 1 4 9 16
```

STL Arrays

The Standard Template Library

`std::array  (C++11)`

```
#include <array>
```

- Fixed size
  - Size must be known at compile time
- Direct element access
- Provides access to the underlying raw array
- Use instead of raw arrays when possible
- All iterators available and do not invalidate

The Standard Template Library

`std::array` – initialization and assignment

```
std::array<int, 5> arr1 { {1,2,3,4,5} };
```

C++11 vs. C++14

```
std::array<std::string, 3> stooges {
      std::string("Larry"},
      "Moe",
      std::string("Curly"}
};

arr1 = {2,4,6,8,10};
```

The Standard Template Library

`std::array` – common methods

```
std::array<int, 5> arr {1,2,3,4,5};

std::cout << arr.size();        // 5

std::cout << arr.at(0);         // 1
std::cout << arr[1];            // 2

std::cout << arr.front();       // 1
std::cout << arr.back();        // 5
```

The Standard Template Library

`std::array` – common methods

```
std::array<int, 5> arr {1,2,3,4,5};
std::array<int, 5> arr1 {10,20,30,40,50};

std::cout << arr.empty();      // 0 (false)
std::cout << arr.max_size();   // 5

arr.swap(arr1);                // swaps the 2 arrays
int *data = arr.data();        // get raw array address
```

STL Vectors

The Standard Template Library

`std::vector`

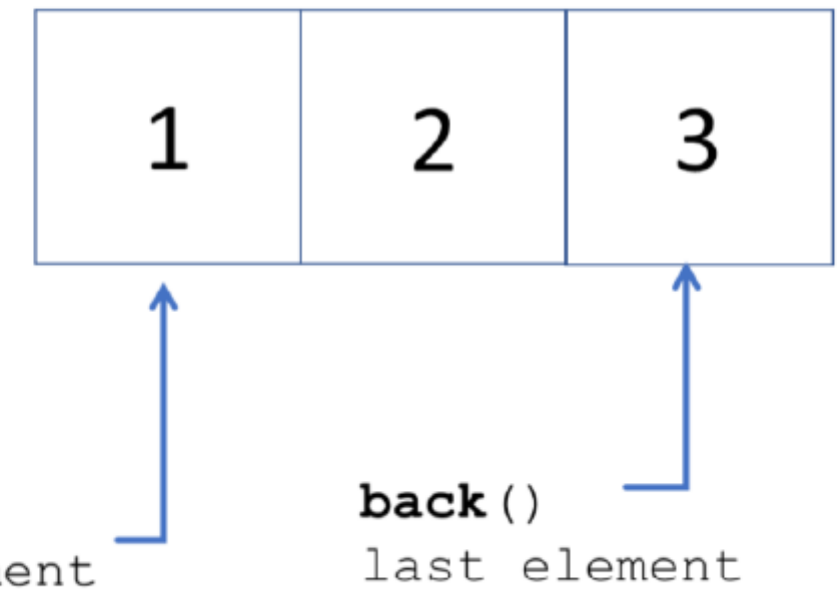
```
#include <vector>
```

- Dynamic size
  - Handled automatically
  - Can expand and contract as needed
  - Elements are stored in contiguous memory as an array
- Direct element access (constant time)
- Rapid insertion and deletion at the back (constant time)
- Insertion or removal of elements (linear time)
- All iterators available and may invalidate

The Standard Template Library

`std::vector`

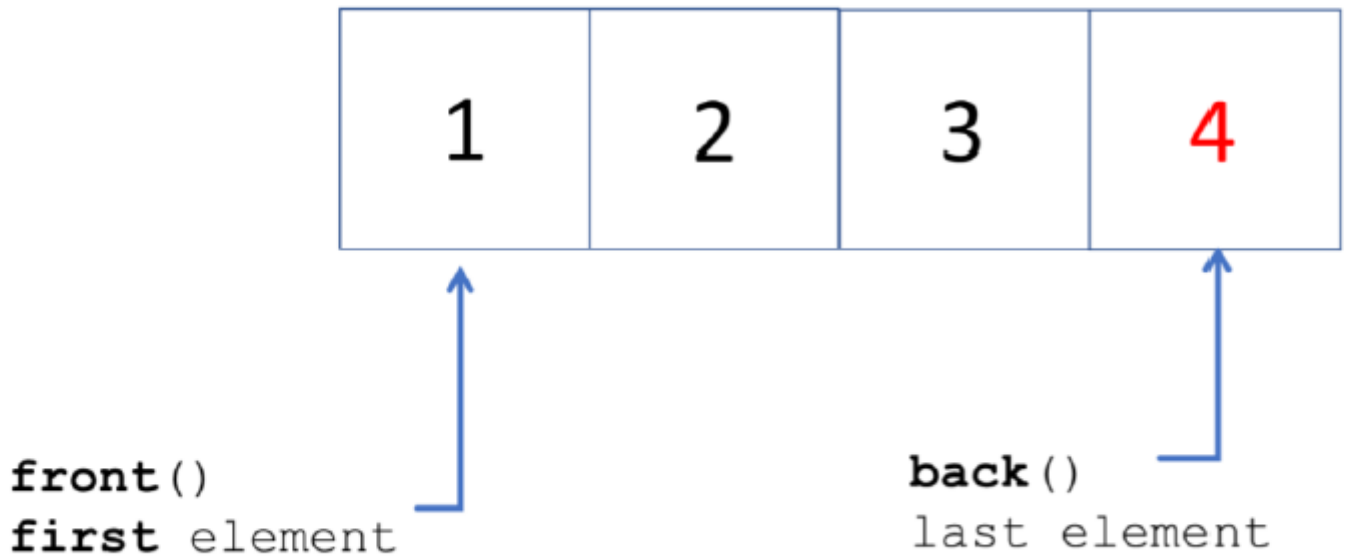
```
std::vector<int> vec {1,2,3};
```



The Standard Template Library

`std::vector`

```
vec.push_back(4)
```



The Standard Template Library

`std::vector` – initialization and assignment

```
std::vector<int> vec {1,2,3,4,5};
std::vector<int> vec1 (10,100);  // ten 100s

std::vector<std::string> stooges {
      std::string{"Larry"},
      "Moe",
      std::string{"Curly"}
};

vec1 = {2,4,6,8,10};
```

The Standard Template Library

`std::vector` – common methods

```
std::vector<int> vec {1,2,3,4,5};

std::cout << vec.size();          // 5
std::cout << vec.capacity();      // 5
std::cout << vec.max_size;        // a very large number

std::cout << vec.at(0);           // 1
std::cout << vec[1];              // 2

std::cout << vec.front();         // 1
std::cout << vec.back();          // 5
```

The Standard Template Library

`std::vector` – common methods

```
Person p1 {"Larry", 18};
std::vector<Person> vec;

vec.push_back(p1);          // add p1 to the back
vec.pop_back();             // remove p1 from the back

vec.push_back(Person{"Larry", 18});

vec.emplace_back("Larry", 18);      // efficient!!
```

The Standard Template Library

`std::vector` – common methods

```
std::vector<int> vec1 {1,2,3,4,5};
std::vector<int> vec2 {10,20,30,40,50};

std::cout << vec1.empty();     // 0 (false)
vec1.swap(vec2);               // swaps the 2 vector

std::sort(vec1.begin(), vec1.end());
```

The Standard Template Library

`std::vector` – common methods

```
std::vector<int> vec1 {1,2,3,4,5};
std::vector<int> vec2 {10,20,30,40,50};

auto it = std::find(vec1.begin(), vec1.end(), 3);
vec1.insert(it, 10);      // 1,2,10,3,4,5

it = std::find(vec1.begin(), vec1.end(), 4);
vec1::insert(it, vec2.begin(), vec2.end());
     // 1,2,10,3,10,20,30,40,50,4,5
```

STL Deque

The Standard Template Library

`std::deque (double ended queue)`

```
#include <deque>
```

- Dynamic size
  - Handled automatically
  - Can expand and contract as needed
  - Elements are NOT stored in contiguous memory
- Direct element access (constant time)
- Rapid insertion and deletion at the front **and** back (constant time)
- Insertion or removal of elements (linear time)
- All iterators available and may invalidate

The Standard Template Library

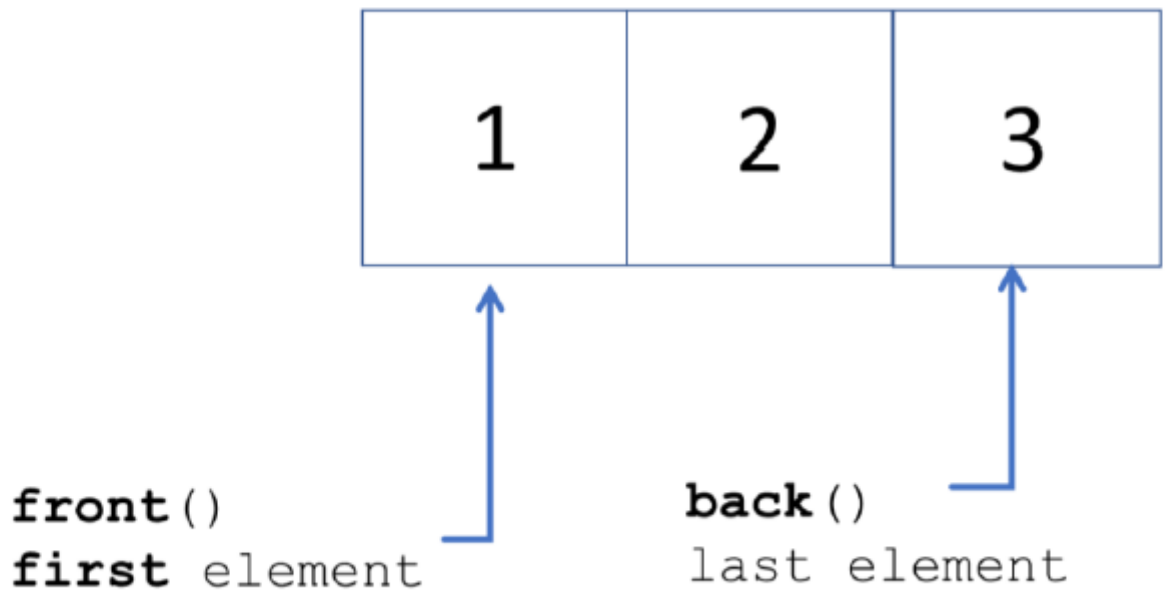
`std::deque` – initialization and assignment

```
std::deque<int> d{1,2,3,4,5};
std::deque<int> d1(10,100);       // ten 100s

std::deque<std::string> stooges {
      std::string("Larry"},
      "Moe",
      std::string("Curly"}
};

d = {2,4,6,8,10};
```

The Standard Template Library

`std::deque`

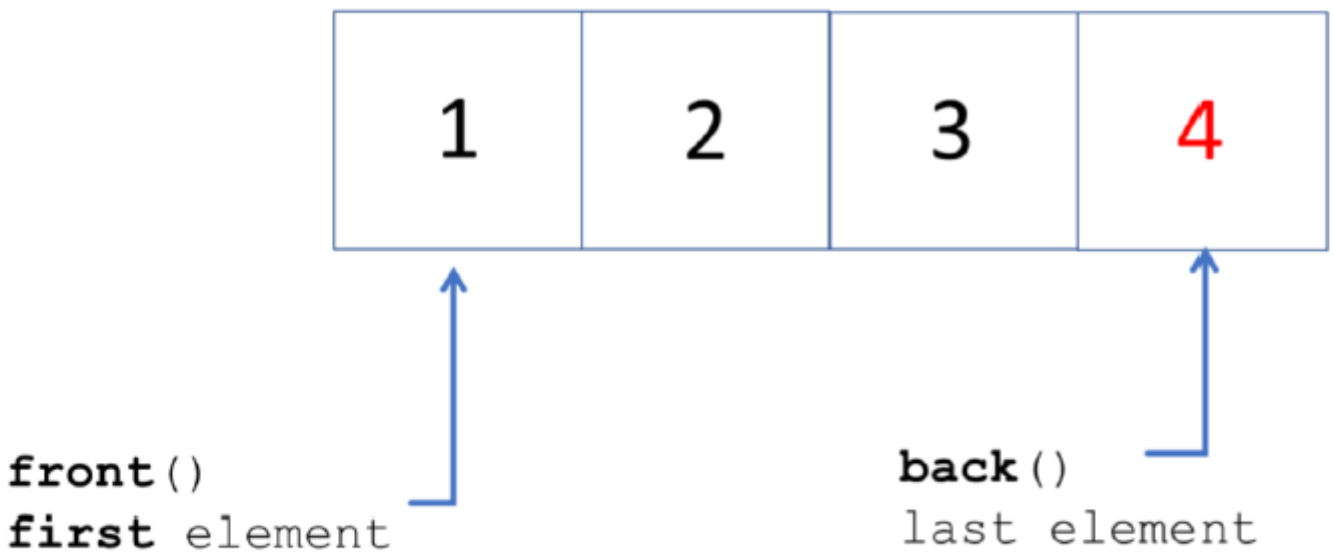
```
std::deque<int> d{1,2,3};
```



The Standard Template Library

`std::deque`

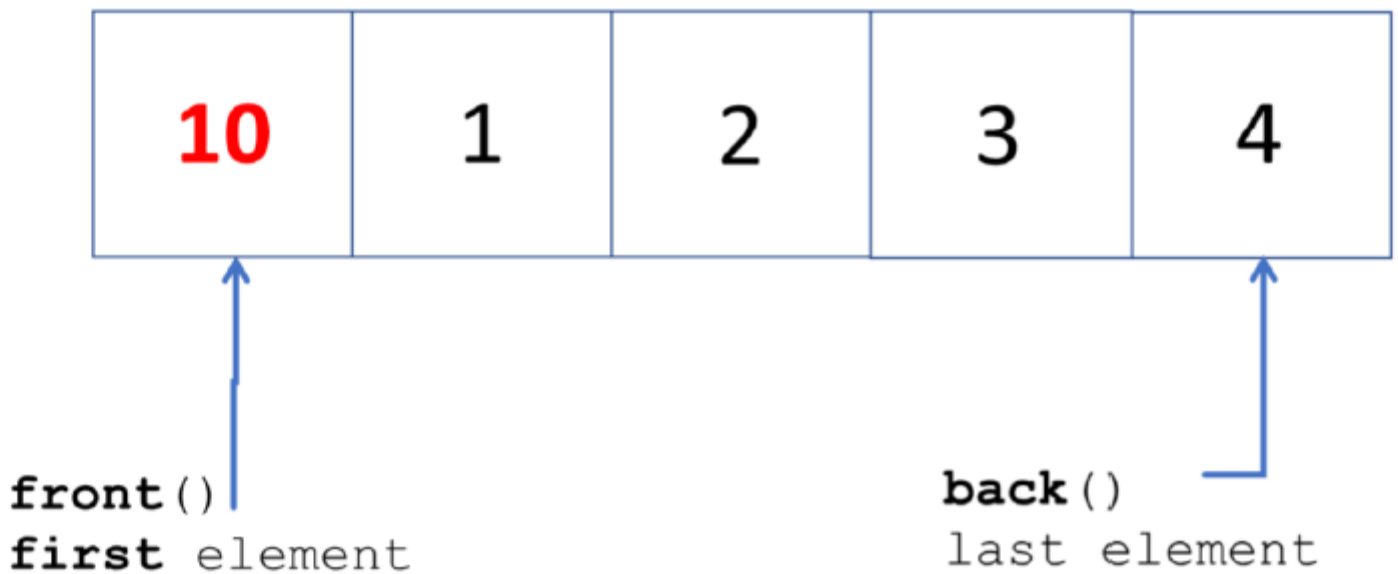
```
d.push_back(4)
```



The Standard Template Library

`std::deque`

```
d.push_front(10)
```



The Standard Template Library

`std::deque`

The Standard Template Library

`std::deque` – common methods

```
std::deque<int> d {1,2,3,4,5};

std::cout << d.size();          // 5
std::cout << d.max_size();      // a very large number

std::cout << d.at(0);           // 1
std::cout << d[1];              // 2

std::cout << d.front();         // 1
std::cout << d.back();          // 5
```

The Standard Template Library

`std::deque` – common methods

```
Person p1 {"Larry", 18};
std::deque<Person> d;

d.push_back(p1);                    // add p1 to the back
d.pop_back();                       // remove p1 from the  back

d.push_front(Person{"Larry", 18});
d.pop_front();                      // remove element from the front

d.emplace_back("Larry", 18);        // add to back efficient!!
d.emplace_front("Moe", 24);         // add to front
```

STL Lists

The Standard Template Library

`std::list and std::forward_list`

- Sequence containers
- Non-contiguous in memory
- No direct access to elements
- Very efficient for inserting and deleting elements once an element is found

The Standard Template Library

`std::list`

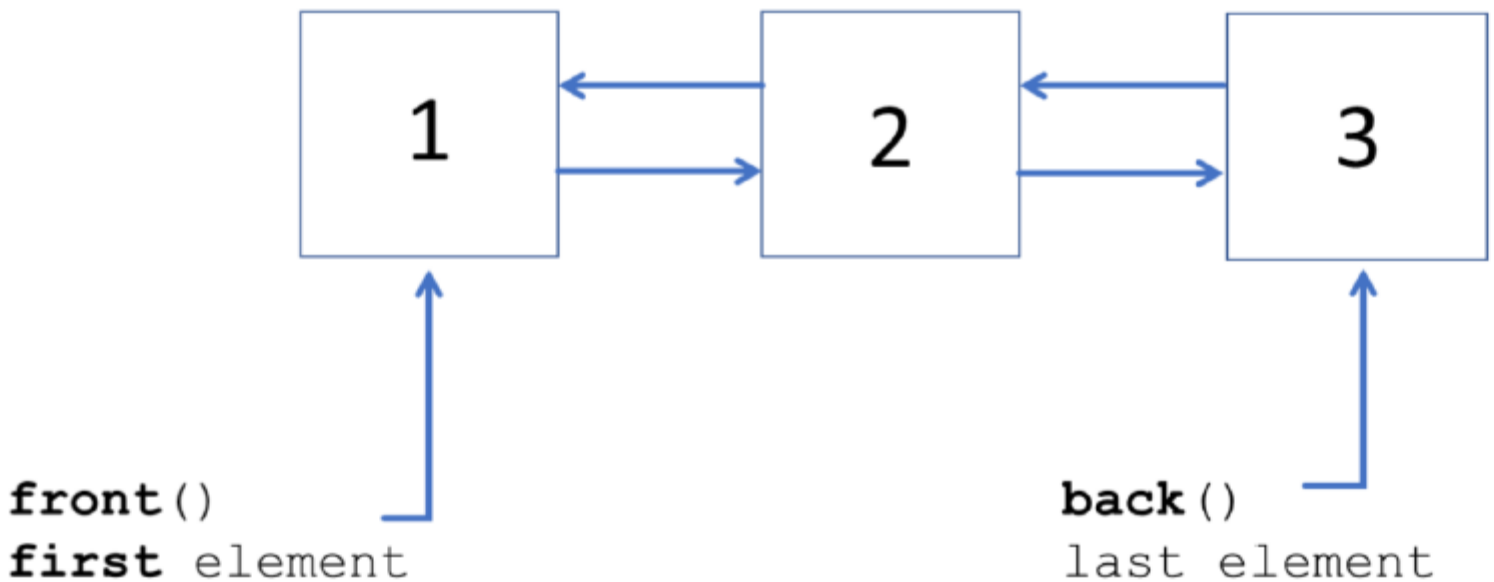
```
#include <list>
```

- Dynamic size
  - Lists of elements
  - `list` is bidirectional (doubly-linked)
- Direct element access is NOT provided
- Rapid insertion and deletion of elements anywhere in the container (constant time)
- All iterators available and invalidate when corresponding element is deleted

The Standard Template Library

`std::list`

```
std::list<int> l{1,2,3};
```



The Standard Template Library

`std::list` – initialization and assignment

```
std::list<int> l {1,2,3,4,5};
std::list<int> ll(10,100); // ten 100s

std::list<std::string> stooges {
      std::string{"Larry"},
      "Moe",
      std::string{"Curly"}
};

l = {2,4,6,8,10};
```

The Standard Template Library

`std::list` – common methods

```
std::list<int> l {1,2,3,4,5};

std::cout << l.size();          // 5
std::cout << l.max_size();      // a very large number

std::cout << 1.front();         // 1
std::cout << 1.back();          // 5
```

The Standard Template Library

`std::list` – common methods

```
Person p1 {"Larry", 18};
std::list<Person> l;

l.push_back(p1);                // add p1 to the back
l.pop_back();                   // remove p1 from the  back

l.push_front(Person{"Larry", 18});
l.pop_front();                  // remove element from the front

l.emplace_back("Larry", 18);     // add to back efficient!!
l.emplace_front("Moe", 24);      // add to front
```

The Standard Template Library

`std::list` – methods that use iterators

```
std::list<int> l {1,2,3,4,5};
auto it = std::find(l.begin(), l.end(), 3);

l.insert(it, 10);              // 1 2 10 3 4 5

l.erase(it);                   // erases the 3,  1 2 10 4 5

l.resize(2);                   // 1 2

l.resize(5);                   // 1 2 0 0 0
```

The Standard Template Library

`std::list` – common methods

```
// traversing the list (bi-directional)

std::list<int> l {1,2,3,4,5};
auto it = std::find(l.begin(), l.end(), 3);

std::cout << *it;                // 3
it++;
std::cout << *it;                // 4
it--;
std::cout << *it;                // 3
```

The Standard Template Library

`std::forward_list`

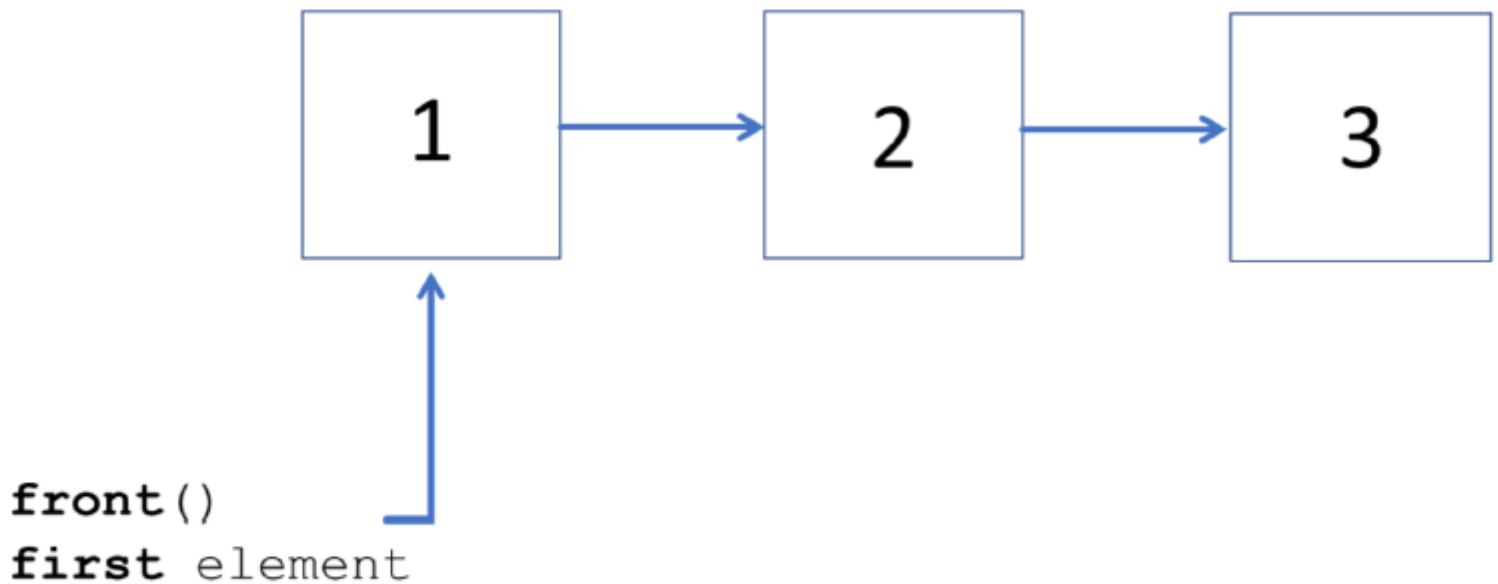
```
#include <forward_list>
```

- Dynamic size
  - Lists of elements
  - `list` uni-directional (singly-linked)
  - Less overhed than a `std::list`
- Direct element access is NOT provided
- Rapid insertion and deletion of elements anywhere in the container (constant time)
- Reverse iterators not available. Iterators invalidate when corresponding element is deleted

The Standard Template Library

`std::forward_list`

```
std::forward_list<int> l{1,2,3};
```



The Standard Template Library

`std::forward_list` – common methods

```
std::forward_list<int> l {1,2,3,4,5};

std::cout << l.size();          // Not available
std::cout << l.max_size;        // a very large number

std::cout << l.front();         // 1
std::cout << l.back();          // Not available
```

The Standard Template Library

`std::forward_list` - common methods

```
Person p1 {"Larry", 18};
std::forward_list<Person> l;

l.push_front(p1);          // add p1 to the front
l.pop_front();             // remove p1 from the front


l.emplace_front("Moe", 24};          // add to front
```

The Standard Template Library

`std::forward_list` – methods that use iterators

```
std::forward_list<int> l {1,2,3,4,5};
auto it = std::find(l.begin(), l.end(), 3);

l.insert_after(it, 10);     // 1 2 3 10 4 5
l.emplace_after(it, 100);   // 1 2 3 100 10 4 5

l.erase_after(it);          // erases the 100,  1 2 3 10 4 5

l.resize(2);                // 1 2

l.resize(5);                // 1 2 0 0 0
```

STL Sets

The Standard Template Library

The STL Set containers

- Associative containers
  - Collection of stored objects that allow fast retrieval using a key
  - STL provides Sets and Maps
  - Usually implemented as a balanced binary tree or hashsets
  - Most operations are very efficient

- Sets
  - `std::set`
  - `std::unordered_set`
  - `std::multiset`
  - `std::unordered_multiset`

The Standard Template Library

`std::set`

```
#include <set>
```

- Similar to a mathematical set
- Ordered by key
- No duplicate elements
- All iterators available and invalidate when corresponding element is deleted

The Standard Template Library

`std::set` – initialization and assignment

```
std::set<int> s {1,2,3,4,5};

std::set<std::string> stooges {
      std::string{"Larry"},
      "Moe",
      std::string{"Curly"}
};

s = {2,4,6,8,10};
```

The Standard Template Library

`std::set` – common methods

```
std::set<int> s {4,1,1,3,3,2,5};   // 1 2 3 4 5

std::cout << s.size();             // 5
std::cout << s.max_size;           // a very large number
```

- No concept of front and back

```
s.insert(7);                       // 1 2 3 4 5 7
```

The Standard Template Library

`std::set` – common methods

```
Person p1 {"Larry", 18};
Person p2 {"Moe", 25};

std::set<Person> stooges;

stooges.insert(p1);                    // adds p1 to the set
auto result = stooges.insert(p2);    // adds p2 to the set
```

- uses `operator<` for ordering!
- returns a `std::pair<iterator, bool>`
  - `first` is an iterator to the inserted element or to the duplicate in the set
  - `second` is a boolean indicating success or failure

The Standard Template Library

`std::set` – common methods

```
std::set<int> s {1,2,3,4,5};



s.erase(3);                    // erase the 3 : 1 2 4 5

auto it = s.find(5);
if (it != s.end())
   s.erase(it);                // erase the 5: 1 2 4
```

The Standard Template Library

`std::set` – common methods

```
std::set<int> s {1,2,3,4,5};

int num = s.count(1);     // 0 or 1

s.clear();                // remove all elements

s.empty();                // true or false
```

The Standard Template Library

`std::multi_set`

```
#include <set>
```

- Sorted by key
- Allows duplicate elements
- All iterators are available

The Standard Template Library

`std::unordered_set`

```
#include <unordered_set>
```

- Elements are unordered
- No duplicate elements allowed
- Elements cannot be modified
  - Must be erased and new element inserted
- No reverse iterators are allowed

The Standard Template Library

`std::unordered_multiset`

```
#include <unordered_set>
```

- Elements are unordered
- Allows duplicate elements
- No reverse iterators are allowed

STL Maps

The Standard Template Library

The STL Map containers

- Associative containers
  - Collection of stored objects that allow fast retrieval using a key
  - STL provides Sets and Maps
  - Usually implemented as a balanced binary tree or hashsets
  - Most operations are very efficient

- Maps
  - `std::map`
  - `std::unordered_map`
  - `std::multimap`
  - `std::unordered_multimap`

The Standard Template Library

`std::map`

```
#include <map>
```

- Similar to a dictionary
- Elements are stored as Key, Value pairs (`std::pair`)
- Ordered by key
- No duplicate elements (keys are unique)
- Direct element access using the key
- All iterators available and invalidate when corresponding element is deleted

The Standard Template Library

`std::map` – initialization and assignment

```
std::map<std::string, int> m1 {
    {"Larry", 18},
    {"Moe", 25}
};

std::map<std::string, std::string> m2 {
    {"Bob", "Butcher"},
    {"Anne", "Baker"},
    {"George", "Candlestick maker"}
};
```

The Standard Template Library

`std::map` – common methods

```
std::map<std::string, std::string> m {
    {"Bob", "Butcher"},
    {"Anne", "Baker"},
    {"George", "Candlestick maker"}
};

std::cout << m.size();          // 3
std::cout << m.max_size;        // a very large number
```

- No concept of front and back

The Standard Template Library

`std::map` – common methods

```
std::map<std::string, std::string> m {
    {"Bob", "Butcher"},
    {"Anne", "Baker"},
    {"George", "Candlestick maker"}
};

std::pair<std::string, std::string> p1 {"James", "Mechanic"};

m.insert(p1);

m.insert(std::make_pair("Roger", "Ranger"));
```

The Standard Template Library

`std::map` – common methods

```
std::map<std::string, std::string> m {
    {"Bob", "Butcher"},
    {"Anne", "Baker"},
    {"George", "Candlestick maker"}
};

m["Frank"] = "Teacher";         // insert

m["Frank"] = "Instructor";      // update value
m.at("Frank") = "Professor";    // update value
```

The Standard Template Library

`std::map` – common methods

```
std::map<std::string, std::string> m {
   {"Bob", "Butcher"},
   {"Anne", "Baker"},
   {"George", "Candlestick maker"}
};

m.erase("Anne");                          // erase Anne

if (m.find("Bob") != m.end())             // find Bob
       std::cout << "Found Bob!";

auto it = m.find("George");
if (it != m.end())
   m.erase(it);                           // erase George
```

The Standard Template Library

`std::map` – common methods

```
std::map<std::string, std::string> m {
    {"Bob", "Butcher"},
    {"Anne", "Baker"},
    {"George", "Candlestick maker"}
};

int num = m.count("Bob");   // 0 or 1

m.clear();                  // remove all elements

m.empty();                  // true or false
```

The Standard Template Library

`std::multi_map`

```
#include <map>
```

- Ordered by key
- Allows duplicate elements
- All iterators are available

The Standard Template Library

`std::unordered_map`

```
#include <unordered_map>
```

- Elements are unordered
- No duplicate elements allowed
- No reverse iterators are allowed

The Standard Template Library

`std::unordered_multimap`

```
#include <unordered_map>
```

- Elements are unordered
- Allows duplicate elements
- No reverse iterators are allowed

STL Stack

The Standard Template Library

`std::stack`

- Last-in First-out (LIFO) data structure
- Implemented as an adapter over other STL container
  Can be implemented as a vector, list, or deque
- All operations occur on one end of the stack (top)
- No iterators are supported

The Standard Template Library

`std::stack`

```
#include <stack>
```

- `push` – insert an element at the top of the stack
- `pop` – remove an element from the top of the stack
- `top` – access the top element of the stack
- `empty` – is the stack empty?
- `size` – number of elements in the stack

The Standard Template Library

`std::stack` – initialization

```
std::stack<int> s;                          // deque

std::stack<int, std::vector<int>> s1;       // vector

std::stack<int, std::list<int>> s2;         // list

std::stack<int, std::deque<int>> s3;        // deque
```

The Standard Template Library

`std::stack` – common methods

```
std::stack<int> s;
```

```
s.push(10);
```


```
s.push(20);
```


```
s.push(30);
```

The Standard Template Library

`std::stack` – common methods

```
std::cout << s.top();
```


```
// 30
```

```
s.pop();
```


```
// 30 is removed
```

```
s.pop();
```


```
// 20 is removed
std::cout << s.size();
// 1
```

STL Queue

The Standard Template Library

`std::queue`

- First-in First-out (FIFO) data structure
- Implemented as an adapter over other STL container
  Can be implemented as a list or deque
- Elements are pushed at the back and
  popped from the front
- No iterators are supported

The Standard Template Library

`std::queue`

```
#include <queue>
```

- `push` – insert an element at the back of the queue
- `pop` – remove an element from the front of the queue

- `front` – access the element at the front
- `back` – access the element at the back

- `empty` – is the queue empty?
- `size` – number of elements in the queue

The Standard Template Library

`std::queue` – initialization

```
std::queue<int> q;                          // deque

std::queue<int, std::list<int>> q2;         // list

std::queue<int, std::deque<int>> q3;        // deque
```

The Standard Template Library

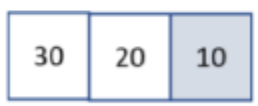
`std::queue` – common methods

```
std::queue<int> q;
```

```
q.push(10);
```


```
q.push(20);
```


```
q.push(30);
```



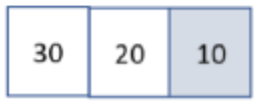
The Standard Template Library

`std::queue` – common methods

```
std::cout << q.front();
```


```
// 10
```

```
std::cout << q.back();
```




```
// 30
```

```
q.pop();
```


```
// remove 10
```

```
q.pop();
```


```
// remove 20
```

```
std::cout << q.size();
// 1
```

STL Priority Queue

The Standard Template Library

`std::priority_queue`

- Allows insertions and removal of elements in order from the front of the container
- Elements are stored internally as a vector by default
- Elements are inserted in ***priority*** order
  (largest value will always be at the front)
- No iterators are supported

The Standard Template Library

`std::priority_queue`

```
#include <queue>
```

- `push` – insert an element into sorted order
- `pop` – removes the top element (greatest)

- `top` – access the top element (greatest)

- `empty` – is the queue empty?
- `size` – number of elements in the queue

The Standard Template Library

`std::priority_queue` – initialization

```
std::priority_queue<int> pq;                    // vector

pq.push(10);
pq.push(20);
pq.push(3);
pq.push(4);

std::cout << pq.top();                          // 20 (largest)
pq.pop();                                       // remove 20
pq.top();                                       /  10 (largest)
```

Good Luck!