# C++ Programming I

## C++ 11 Standard Template Library - STL

*C++ Programming*
*May 24, 2018*

Dr. P. Arnold
Bern University of Applied Sciences

# Agenda

- Standard Template Library

# Agenda

- Standard Template Library
- Containers

# Agenda

- Standard Template Library
- Containers
- Iterators

# Agenda

- Standard Template Library
- Containers
- Iterators
- Algorithms

# Agenda

- **Standard Template Library**
- **Containers**
- **Iterators**
- **Algorithms**
- **Exercise**
  - File I/O

# Agenda

- Standard Template Library
- Containers
- Iterators
- Algorithms
- Exercise
  - File I/O
- Last Lecture

# Standard Template Library

# STL
## Intro

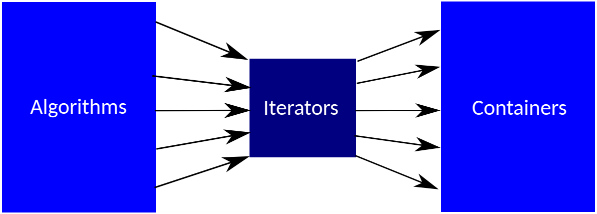
- The standard template library (STL) is a set of template classes and functions that supply the programmer with
    1. **Containers** for storing information
    2. **Iterators** for accessing the information stored
    3. **Algorithms** for manipulating the content of the containers

# STL
## Intro

- The standard template library (STL) is a set of template classes and functions that supply the programmer with
  1. **Containers** for storing information
  2. **Iterators** for accessing the information stored
  3. **Algorithms** for manipulating the content of the containers


- N algorithms, M containers $\rightarrow$ N $\cdot$ M implementations

- The standard template library (STL) is a set of template classes and functions that supply the programmer with
  1. **Containers** for storing information
  2. **Iterators** for accessing the information stored
  3. **Algorithms** for manipulating the content of the containers




- N algorithms, M containers $\rightarrow$ N + M implementations

# Containers

# STL - Containers
## Types

- Containers are STL classes that are used to store data. STL supplies two types of container classes:
  1. **Sequential containers**
  2. **Associative containers**
- In addition classes called *container adapters* with reduced functionality are provided

# STL - Containers

## Sequential Containers

Sequential containers are characterized by a **fast insertion time**, but are relatively **slow in find operations**.

# STL - Containers

## Sequential Containers

Sequential containers are characterized by a **fast insertion time**, but are relatively **slow in find operations**.

- `std::vector` - Operates like a dynamic array and grows only at the end

# STL - Containers
## Sequential Containers

Sequential containers are characterized by a **fast insertion time**, but are relatively **slow in find operations**.

- `std::vector` - Operates like a dynamic array and grows only at the end
- `std::deque` - Similar to `std::vector` except that it allows for new elements to be inserted or removed at the beginning, too

# STL - Containers

## Sequential Containers

Sequential containers are characterized by a **fast insertion time**, but are relatively **slow in find operations**.

- `std::vector` - Operates like a dynamic array and grows only at the end
- `std::deque` - Similar to `std::vector` except that it allows for new elements to be inserted or removed at the beginning, too
- `std::list` - Operates like a double linked list. Like a chain where an object is a link in the chain. You can add or remove links, *i.e.* objects at any position

# STL - Containers
## Sequential Containers

Sequential containers are characterized by a **fast insertion time**, but are relatively **slow in find operations**.

- `std::vector` - Operates like a dynamic array and grows only at the end
- `std::deque` - Similar to `std::vector` except that it allows for new elements to be inserted or removed at the beginning, too
- `std::list` - Operates like a double linked list. Like a chain where an object is a link in the chain. You can add or remove links, *i.e.* objects at any position
- `std::forward_list` Similar to a `std::list` except that it is a singly linked list of elements that allows you to iterate only in one direction

# STL - Containers

## Associative Containers

Associative containers store data in a sorted fashion. This results in **slower insertion times**, but **optimized search performance**

# STL - Containers

## Associative Containers

Associative containers store data in a sorted fashion. This results in **slower insertion times**, but **optimized search performance**

- `std::set` - Stores unique values sorted on inse in a container featuring logarithmic complexity $\mathcal{O}(\log n)$

# STL - Containers

## Associative Containers

Associative containers store data in a sorted fashion. This results in **slower insertion times**, but **optimized search performance**

- `std::set` - Stores unique values sorted on insertion in a container featuring logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_set` - Stores unique values sorted on insertion in a container featuring near constant complexity $\mathcal{O}(1)$. Available starting C++ 11

# STL - Containers

## Associative Containers

Associative containers store data in a sorted fashion. This results in **slower insertion times**, but **optimized search performance**

- `std::set` - Stores unique values sorted on insertion in a container featuring logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_set` - Stores unique values sorted on insertion in a container featuring near constant complexity $\mathcal{O}(1)$. Available starting C++ 11
- `std::map` - Stores key-value pairs sorted by their unique keys in a container with logarithmic complexity $\mathcal{O}(\log n)$

# STL - Containers
## Associative Containers

Associative containers store data in a sorted fashion. This results in **slower insertion times**, but **optimized search performance**

- `std::set` - Stores unique values sorted on insertion in a container featuring logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_set` - Stores unique values sorted on insertion in a container featuring near constant complexity $\mathcal{O}(1)$. Available starting C++ 11
- `std::map` - Stores key-value pairs sorted by their unique keys in a container with logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_map` - Stores key-value pairs sorted by their unique keys in a container with near constant complexity $\mathcal{O}(1)$(since C++ 11)

# STL - Containers
## Associative Containers

Associative containers store data in a sorted fashion. This results in **slower insertion times**, but **optimized search performance**

- `std::set` - Stores unique values sorted on insertion in a container featuring logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_set` - Stores unique values sorted on insertion in a container featuring near constant complexity $\mathcal{O}(1)$. Available starting C++ 11
- `std::map` - Stores key-value pairs sorted by their unique keys in a container with logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_map` - Stores key-value pairs sorted by their unique keys in a container with near constant complexity $\mathcal{O}(1)$(since C++ 11)
- `std::multiset` - Like `set`. Additionally, supports the ability to store multiple items having the same value; that is, the value doesn't need to be unique $\mathcal{O}(\log n)$

# STL - Containers

## Associative Containers

Associative containers store data in a sorted fashion. This results in **slower insertion times**, but **optimized search performance**

- `std::set` - Stores unique values sorted on insertion in a container featuring logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_set` - Stores unique values sorted on insertion in a container featuring near constant complexity $\mathcal{O}(1)$. Available starting C++ 11
- `std::map` - Stores key-value pairs sorted by their unique keys in a container with logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_map` - Stores key-value pairs sorted by their unique keys in a container with near constant complexity $\mathcal{O}(1)$(since C++ 11)
- `std::multiset` - Like `set`. Additionally, supports the ability to store multiple items having the same value; that is, the value doesn't need to be unique $\mathcal{O}(\log n)$
- `std::unordered_multiset` - Like `unordered_set`. Additionally, supports the ability to store multiple items having the same value; that is, the value doesn't need to be unique $\mathcal{O}(1)$( since C++ 11)

# STL - Containers

## Associative Containers

Associative containers store data in a sorted fashion. This results in **slower insertion times**, but **optimized search performance**

- `std::set` - Stores unique values sorted on insertion in a container featuring logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_set` - Stores unique values sorted on insertion in a container featuring near constant complexity $\mathcal{O}(1)$. Available starting C++ 11
- `std::map` - Stores key-value pairs sorted by their unique keys in a container with logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_map` - Stores key-value pairs sorted by their unique keys in a container with near constant complexity $\mathcal{O}(1)$(since C++ 11)
- `std::multiset` - Like `set`. Additionally, supports the ability to store multiple items having the same value; that is, the value doesn't need to be unique $\mathcal{O}(\log n)$
- `std::unordered_multiset` - Like `unordered_set`. Additionally, supports the ability to store multiple items having the same value; that is, the value doesn't need to be unique $\mathcal{O}(1)$( since C++ 11)
- `std::multimap` - Like `map` . Additionally, supports the ability to store key-value pairs where keys don't need to be unique. $\mathcal{O}(\log n)$

# STL - Containers
## Associative Containers

Associative containers store data in a sorted fashion. This results in **slower insertion times**, but **optimized search performance**

- `std::set` - Stores unique values sorted on insertion in a container featuring logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_set` - Stores unique values sorted on insertion in a container featuring near constant complexity $\mathcal{O}(1)$. Available starting C++ 11
- `std::map` - Stores key-value pairs sorted by their unique keys in a container with logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_map` - Stores key-value pairs sorted by their unique keys in a container with near constant complexity $\mathcal{O}(1)$(since C++ 11)
- `std::multiset` - Like `set`. Additionally, supports the ability to store multiple items having the same value; that is, the value doesn't need to be unique $\mathcal{O}(\log n)$
- `std::unordered_multiset` - Like `unordered_set`. Additionally, supports the ability to store multiple items having the same value; that is, the value doesn't need to be unique $\mathcal{O}(1)$( since C++ 11)
- `std::multimap` - Like `map` . Additionally, supports the ability to store key-value pairs where keys don't need to be unique. $\mathcal{O}(\log n)$
- `std::unordered_multimap` - Like `unordered_map`. Additionally, supports the ability to store key-value pairs where keys don't need to be unique $\mathcal{O}(1)$ (since C++ 11)

# STL - Containers

## Container Adapters

Container adapters are variants of sequential and associative containers that have limited functionality and are intended to fulfill a particular purpose

# STL - Containers

## Container Adapters

Container adapters are variants of sequential and associative containers that have limited functionality and are intended to fulfill a particular purpose

- `std::stack` - Stores elements in a LIFO (last-in-first-out) fashion, allowing elements to be inserted (pushed) and removed (popped) at the top

# STL - Containers
## Container Adapters

Container adapters are variants of sequential and associative containers that have limited functionality and are intended to fulfill a particular purpose

- `std::stack` - Stores elements in a LIFO (last-in-first-out) fashion, allowing elements to be inserted (pushed) and removed (popped) at the top
- `std::queue` - Stores elements in FIFO (first-in-first-out) fashion, allowing the first element to be removed in the order they're inserted

# STL - Containers
## Container Adapters

Container adapters are variants of sequential and associative containers that have limited functionality and are intended to fulfill a particular purpose

- `std::stack` - Stores elements in a LIFO (last-in-first-out) fashion, allowing elements to be inserted (pushed) and removed (popped) at the top
- `std::queue` - Stores elements in FIFO (first-in-first-out) fashion, allowing the first element to be removed in the order they're inserted
- `std::priority_queue` - Stores elements in a sorted order, such that the one whose value is evaluated to be the highest is always first in the queue

# Iterators

# STL - Iterators
## Operators

- An iterator is an object that can traverse (iterate over) a container class without the user having to know how the container is implemented
- An iterator is best visualized as a pointer to a given element in the container, with a set of overloaded operators to provide a set of well-defined functions
  1. `operator*` - Dereferencing the iterator returns the element that the iterator is currently pointing at
  2. `operator++` - Moves the iterator to the next element in the container. Most iterators also provide `operator--` to move to the previous element
  3. `operator==` and `operator!=` - Basic comparison operators to determine if two iterators point to the same element. To compare the values that two iterators are pointing at, dereference the iterators first, and then use a comparison operator
  4. `operator=` - Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator points at, dereference the iterator first, then use the assign operator

**STL-Iterators**

Iterators are the bridge that allow the STL-algorithms to work with STL-containers

# STL - Iterators

## Common Members

- Each container includes four basic member functions:
  1. `begin()` - returns an iterator representing the beginning of the elements in the container
  2. `end()` - returns an iterator representing the element just past the end of the elements
  3. `cbegin()` - returns a const (read-only) iterator representing the beginning of the elements in the container
  4. `cend()` - returns a const (read-only) iterator representing the element just past the end of the elements
- Note: `end()` points to the element just past the end! This is done primarily to make looping easy

# STL - Iterators

## Common Members

- Each container includes four basic member functions:
  1. `begin()` - returns an iterator representing the beginning of the elements in the container
  2. `end()` - returns an iterator representing the element just past the end of the elements
  3. `cbegin()` - returns a const (read-only) iterator representing the beginning of the elements in the container
  4. `cend()` - returns a const (read-only) iterator representing the element just past the end of the elements
- Note: `end()` points to the element just past the end! This is done primarily to make looping easy
- All containers provide (at least) two types of iterators:
  1. `container::iterator` provides a read/write iterator
  2. `container::const_iterator` provides a read-only iterator
- See examples ...

# STL - Iterators

## Example - vector

```
#include <iostream>
#include <vector>
int main()
{
    using namespace std;

    vector<int> vect;
    for (int i=0; i < 6; i++)
    {
        vect.push_back(i);
    }

    vector<int>::const_iterator it; // read-only iterator
    it = vect.begin(); // assign it to the start of the vector
    while (it != vect.end()) // while not at end
    {
        cout << *it << " "; // print value it points to
        ++it; // and iterate to the next element
    }

    cout << endl; // Output: 0 1 2 3 4 5
}
```

# STL - Iterators

## Example - `list`

```cpp
#include <iostream>
#include <list>
int main()
{
    using namespace std;

    list<int> li;
    for (int i=0; i < 6; i++)
    {
        li.push_back(i);
    }

    list<int>::const_iterator it; // declare an iterator
    it = li.begin(); // assign it to the start of the list
    while (it != li.end()) // while not at end
    {
        cout << *it << " "; // print the value it points to
        ++it; // and iterate to the next element
    }

    cout << endl; // Output: 0 1 2 3 4 5
}
```

Note: The code is almost identical to the vector case, even though vectors and lists have almost completely different internal implementations!

# STL - Iterators

## Example - set

```
#include <iostream>
#include <set>
int main()
{
    using namespace std;

    set<int> myset;
    myset.insert(7);
    myset.insert(2);
    myset.insert(-6);
    myset.insert(8);
    myset.insert(1);
    myset.insert(-4);

    set<int>::const_iterator it; // declare an iterator
    it = myset.begin(); // assign it to the start of the set
    while (it != myset.end()) // while not at end
    {
        cout << *it << " "; // print the value it points to
        ++it; // and iterate to the next element
    }

    cout << endl; // Output: -6 -4 1 2 7 8
}
```

Note: Besides the creation, the code used to iterate through the elements of the set is essentially identical as before!

# STL - Iterators

## Example - map

```
#include <iostream>
#include <map>
#include <string>
int main()
{
    using namespace std;

    map<int, string> mymap;
    mymap.insert(make_pair(4, "apple"));
    mymap.insert(make_pair(2, "orange"));
    mymap.insert(make_pair(1, "banana"));
    mymap.insert(make_pair(3, "grapes"));
    mymap.insert(make_pair(6, "mango"));
    mymap.insert(make_pair(5, "peach"));

    map<int, string>::const_iterator it; // declare an iterator
    it = mymap.begin(); // assign it to the start of the vector
    while (it != mymap.end()) // while not at end
    {
        cout << it->first << "=" << it->second << " "; // print
        ++it; // and iterate to the next element
    }
    cout << endl;
    // Output: 1=banana 2=orange 3=grapes 4=apple 5=peach 6=mango
}
```

Note: Iterators make it easy to step through each of the elements of the container. You don't have to care at all how map stores its data!

# Algorithms

# STL Algorithms
## Standard Programming Requirements

STL algorithms supplies the programmer with the most common used requirements

# STL Algorithms

## Standard Programming Requirements

STL algorithms supplies the programmer with the most common used requirements

- `std::find` - Finds a value in a collection

# STL Algorithms

## Standard Programming Requirements

STL algorithms supplies the programmer with the most common used requirements

- `std::find` - Finds a value in a collection
- `std::find_if` - Finds a value in a collection on the basis of a specific user-defined predicate

# STL Algorithms

## Standard Programming Requirements

STL algorithms supplies the programmer with the most common used requirements

- `std::find` - Finds a value in a collection
- `std::find_if` - Finds a value in a collection on the basis of a specific user-defined predicate
- `std::reverse` - Reverses a collection

# STL Algorithms

## Standard Programming Requirements

STL algorithms supplies the programmer with the most common used requirements

- `std::find` - Finds a value in a collection
- `std::find_if` - Finds a value in a collection on the basis of a specific user-defined predicate
- `std::reverse` - Reverses a collection
- `std::remove_if` - Removes an item from a collection on the basis of a user-defined predicate

# STL Algorithms

## Standard Programming Requirements

STL algorithms supplies the programmer with the most common used requirements

- `std::find` - Finds a value in a collection
- `std::find_if` - Finds a value in a collection on the basis of a specific user-defined predicate
- `std::reverse` - Reverses a collection
- `std::remove_if` - Removes an item from a collection on the basis of a user-defined predicate
- `std::transform` - Applies a user-defined transformation function to elements in a container

**STL Algorithms**

To use those algorithms include the standard header `<algorithm>`

# STL Algorithms

## Example - `find`

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
    // A dynamic array of integers
    vector<int> intArray;
    intArray.push_back(50);
    intArray.push_back(2991);
    intArray.push_back(23);
    intArray.push_back(9999);

    // Find an element (say 2991) using the 'find' algorithm
    vector<int>::iterator elFound = find(intArray.begin(),
         intArray.end(), 2991);
    // Check if value was found
    if (elFound != intArray.end())
    {
        // Determine position of element using std::distance
        int elPos = distance(intArray.begin(), elFound);
        cout << "Value "<< *elFound;
        cout << " found in the vector at position: " << elPos <<
             endl;
    }
    return 0;
}
```

# STL Algorithms

## Example - `find`

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
    // A dynamic array of integers
    vector<int> intArray;
    intArray.push_back(50);
    intArray.push_back(2991);
    intArray.push_back(23);
    intArray.push_back(9999);

    // Use auto for convenience
    auto elFound = find(intArray.begin(), intArray.end(), 2991);
    // Check if value was found
    if (elFound != intArray.end())
    {
        // Determine position of element using std::distance
        int elPos = distance(intArray.begin(), elFound);
        cout << "Value "<< *elFound;
        cout << " found in the vector at position: " << elPos <<
            endl;
    }
    return 0;
}
```

# STL

## Choosing the right Container

- If you're developping a new application, your requirements might be satisfied by more than one STL container. Nevertheless, the wrong choice could result in performance issues and scalability bottlenecks
- Refer to the companion book to find a comprehensive list (p. 429)

| Container | Advantages | Disadvantages |
|---|---|---|
| std::unordered_ multiset (Associative Container) | Should be preferred over an unordered_set when you need to contain nonunique values too.<br><br>Performance is similar to unordered_set, namely, constant average time for search, insertion, and removal of elements, independent of size of container. | Elements are weakly ordered, so one cannot rely on their relative position within the container. |
| std::map (Associative Container) | Key-value pairs container that offers search performance proportional to the logarithm of number of elements in the container and hence often significantly faster than sequential containers. | Elements (pairs) are sorted on insertion, hence insertion will be slower than in a sequential container of pairs. |
| std::unordered_map. (Associative Container) | Offers advantage of near constant time search, insertion, and removal of elements independent of the size of the container. | Elements are weakly ordered and hence not suited to cases where order is important. |
| std::multimap. (Associative Container) | To be selected over std::map when requirements | Insertion of elements will be slower than in a sequential |

# Exercise

# Exercise 11

## Letter Frequency

In exercise 11 you'll have to

- Read and write files with `fstream`
- Use a container to count, search and sort values

# File I/O

## File Output

- Writing a file using the `ofstream` class is straight forward:

```
#include <fstream>
#include <iostream>

int main()
{
    using namespace std;

    // ofstream is used for writing files
    // We'll make a file called Sample.dat
    ofstream out("Sample.txt");

    // If we couldn't open the output file stream for writing
    if(!out)
    {
        // Print an error and exit
        cout << "Open Sample.txt failed!" << endl;
        return 1;
    }

    // We'll write two lines into this file
    out << "This is line 1" << endl;
    out << "This is line 2" << endl;

    return 0;
}
```

# File I/O

## File Input

- Reading a file using the `ifstream` class
- Note that `ifstream` returns a 0 if we've reached the end of the file (EOF)

```cpp
#include <fstream>
#include <iostream>
#include <string>

int main()
{
    using namespace std;
    ifstream in("Sample.txt");
    // If we couldn't open the output file stream for reading
    if(!in)
    {
        // Print an error
        cout << "Open Sample.txt failed" << endl;
        return 1;
    }

    // While there's still stuff left to read
    while(in)
    {
        string strInput;
        in >> strInput; // read one string from the stream!
        cout << strInput << endl;
    }
    return 0;
}
```

# File I/O

## File Input

- Use `getline()` to read entire lines

```
#include <fstream>
#include <iostream>
#include <string>

int main()
{
    using namespace std;
    ifstream in("Sample.txt");

    // If we couldn't open the input file stream for reading
    if (!in)
    {
        // Print an error
        cout << "Open Sample.txt failed" << endl;
        exit(1);
    }

    // While there's still stuff left to read
    while(in)
    {
        std::string strInput;
        getline(in, strInput); // read line!
        cout << strInput << endl;
    }
    return 0;
}
```

# File I/O
## File Input

- Reading a file using the `ifstream` class
- Note that `ifstream` returns a 0 if we've reached the end of the file (EOF)

```
#include <fstream>
#include <iostream>
#include <string>

int main()
{
    using namespace std;
    ifstream in("Sample.txt");
    // If we couldn't open the output file stream for reading
    if(!in)
    {
        // Print an error
        cout << "Open Sample.txt failed" << endl;
        return 1;
    }

    // While there's still stuff left to read
    while(in)
    {
        string strInput;
        in >> strInput; // read one string from the stream!
        cout << strInput << endl;
    }
    return 0;
}
```

# Last Lecture

# Last Lecture

## Topics

- Exercise 11
- Topics to repeat
- Outlook and contents of C++ Programming II
- Application Demos

# Thank You

## Questions

???