C++ Programming II

Introduction to Standard Template Library
Overview STL Containers

C++ Programming II
September 16, 2018

Prof. Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- **Standard Template Library**

Agenda

- Standard Template Library
- Containers

Agenda

- ▶ **Standard Template Library**
- ▶ **Containers**
- ▶ **Sequential Containers**
  - Vector
  - Deque
  - List
  - Array

Agenda

- **Standard Template Library**
- **Containers**
- **Sequential Containers**
  - Vector
  - Deque
  - List
  - Array
- **Associative Containers**
  - Set / Multiset
  - Map / Multimap

Agenda

- Standard Template Library
- Containers
- Sequential Containers
  - Vector
  - Deque
  - List
  - Array
- Associative Containers
  - Set / Multiset
  - Map / Multimap
- Unordered Containers
  - Unordered Set / Unordered Multiset

Agenda

- **Standard Template Library**
- **Containers**
- **Sequential Containers**
  - Vector
  - Deque
  - List
  - Array
- **Associative Containers**
  - Set / Multiset
  - Map / Multimap
- **Unordered Containers**
  - Unordered Set / Unordered Multiset
- **Container Adapters**

Agenda

- Standard Template Library
- Containers
- Sequential Containers
  - Vector
  - Deque
  - List
  - Array
- Associative Containers
  - Set / Multiset
  - Map / Multimap
- Unordered Containers
  - Unordered Set / Unordered Multiset
- Container Adapters
- Algorithms

Standard Template Library

STL

Motivation

```
int count_a(void *buf)
{
    int count;
    char *bufr;
    count = 0;
    bufr = (char*)buf;
    for(int i = 0; i < bufr != '/0'; ++i)
    {
        if(bufr[i] = 'a')
        {
            ++count;
        }
    }
    return count;
}
```

- From this ...

STL
Motivation

```
size_t count_a(const std::vector<char> &vec)
{
    size_t count;



    for(const auto c : vec)
    {
        if(c == 'a')
        {
            ++count;
        }
    }
    return count;
}
```

- ... over this ...

STL

Motivation

```
size_t count_a(const std::vector<char> &vec)
{
    return std::count(vec.begin(), vec.end(), 'a');
}
```

- .. to finally this

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

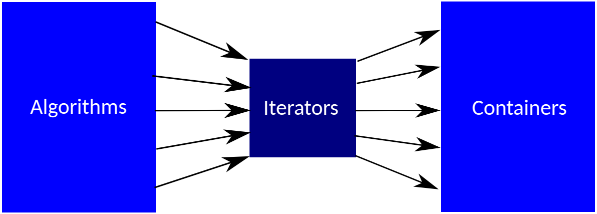
Intro

- 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

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 $+$ M implementations

STL

Example of STL Work Flow

```cpp
#include <iostream>
#include <vector>

// Everything in STL Library is defined in the namespace std
using namespace std;

int main()
{
    // Container
    vector<int> vec;
    vec.push_back(4);
    vec.push_back(1);
    vec.push_back(8); // vec{4, 1, 8}

    // Iterator
    vector<int>::iterator itr1 = vec.begin(); // Points to first element
    vector<int>::iterator itr2 = vec.end(); // Points to the spot afer the last element!

    for(vector<int>::iterator itr = itr1; itr!=itr2; ++itr)
        cout << *itr << " ";    // Print out: 4 1 8

    // Algorithms work on iterators
    sort(itr1,itr2); // vec: {1, 4, 8}

    return 0;
}
```

STL

Example of STL Work Flow

- The majority of our C++ programs should be using STL!
- **Code reuse**, no need to re-invent the wheel
- **Efficiency** Modern C++ compilers are tuned to optimize for C++ STL code
- **Bullet proove**
- **less buggy**
- **clean and readable**
- Best use of data structure and algorithms under the hood

Containers

STL - Containers

Types

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

STL - Containers

Sequential Containers (array and linked list)

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

STL - Containers

Sequential Containers (array and linked list)

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 (array and linked list)

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 (array and linked list)

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 (array and linked list)

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

Sequential Containers (array and linked list)

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
- `std::array` Fixed size array with the performance and accessibility of a C-style array with the benefits of a standard container, such as knowing its own size, supporting assignment, random access iterators, etc.

STL - Containers

Associative Containers (binary trees)

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

STL - Containers

Associative Containers (binary trees)

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 $\mathcal{O}(\log n)$

STL - Containers
Associative Containers (binary trees)

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::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 (binary trees)

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::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::map` - Stores key-value pairs sorted by their unique keys in a container with logarithmic complexity $\mathcal{O}(\log n)$

STL - Containers

Associative Containers (binary trees)

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::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::map` - Stores key-value pairs sorted by their unique keys in a container with logarithmic complexity $\mathcal{O}(\log n)$

STL - Containers

Associative Containers (binary trees)

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::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::map` - Stores key-value pairs sorted by their unique keys in a container with logarithmic complexity $\mathcal{O}(\log n)$
- `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

Unordered Containers (hash tables)

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

STL - Containers

Unordered Containers (hash tables)

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

- `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

Unordered Containers (hash tables)

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

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

Unordered Containers (hash tables)

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

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

Unordered Containers (hash tables)

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

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

STL - Containers
Headers

In order to use STL containers you have to use the following headers:

```
#include <vector>
#include <deque>
#include <list>
#include <set>              // set and multiset
#include <map>              // map and multimap
#include <unordered_set>    // unordered set/multiset
#include <unordered_map>    // unordered map/multimap
#include <queue>            // queue / priority_queue
#include <stack>
#include <iterator>
#include <algorithm>
#include <numeric>          // some numeric algorithms
#include <functional>
```

Sequential Containers

STL - Vector
Initializing Vectors

```
std::vector<int> vec1;                  // empty vector of ints
std::vector<int> vec2(3);               // 3 ints
std::vector<int> vec2(3,10);            // 3 ints with value 10
std::vector<int> vec3{1,2,3,4};         // 4 ints: 1,2,3,4

std::vector<int> vec4(vec2.begin(),vec2.end()); // via vec2
std::vector<int> vec5(vec3);            // a copy of vec3

int myInt[] = {1,2,3};                  // construct from arrays
std::vector<int> vec6(myInt, myInt + sizeof(myInt)/sizeof(int));
```

STL - Vector

Using Vectors

```
vector<int> vec;      // vec.size() -> 0
vec.push_back(4);
vec.push_back(1);
vec.push_back(8);     // vec{4, 1, 8}: vec.size() -> 3

// Vector specific (random access)
cout << vec[2];       // 8 (no range check)
cout << vec.at(2);    // 8 (throws exception out of range)

for(int i = 0; i < vec.size(); ++i)
    cout << vec[i] << " ";   // Random access possible

vector<int>::iterator itr; // Create iterator
for(itr = vec.begin(); itr!=vec.end(); itr++)
    cout << *itr << " ";     // Recommended to use iterators:
                             // 1) Faster than random access;
                             // 2) Universal way of container
                             //    traversing

for(auto elem : vec)         // C++ 11 - Most convinient
    cout << elem << " ";

// Vector is a dynamically allocated contigous array in memory
int* p = &vec[0];
```

STL - Vector
Using Vectors

```cpp
// Common member functions for all containers

if(vec.empty())
    cout << "no elements ";

vec.size();                  // 3

vector<int> vec2(vec);       // Copy Contructr, vec2: {4, 1, 8}

vec.clear();                 // Remove all items: vec.size() -> 0

vec2.swap(vec);              // vec2 becomes empty, vec has 3 elements
```

Properties of STL-Vector:

- fast insert/remove at end: $\mathcal{O}(1)$
- slow insert/remove at the beginning and middle: $\mathcal{O}(n)$
- slow search: $\mathcal{O}(n)$

STL - Vector

Using Vectors

vector

STL - Vector

Performance

Vectors grow dynamically, and every vector has a specific size. When we add a new element to a vector, the computer reallocates memory and may even copy all of the vector elements into this new memory! This can cause a performance hit.

STL - Vector
Performance

Vectors grow dynamically, and every vector has a specific size. When we add a new element to a vector, the computer reallocates memory and may even copy all of the vector elements into this new memory! This can cause a performance hit.

- `capacity()` - returns the capacity of the vector, *i.e.* the number of elements that a vector can hold before a program must allocate more memory

STL - Vector
Performance

Vectors grow dynamically, and every vector has a specific size. When we add a new element to a vector, the computer reallocates memory and may even copy all of the vector elements into this new memory! This can cause a performance hit.

- `capacity()` - returns the capacity of the vector, *i.e.* the number of elements that a vector can hold before a program must allocate more memory
- `size()` - returns the currently filled level, which is always equal to or less than the capacity

STL - Vector
Performance

Vectors grow dynamically, and every vector has a specific size. When we add a new element to a vector, the computer reallocates memory and may even copy all of the vector elements into this new memory! This can cause a performance hit.

- `capacity()` - returns the capacity of the vector, *i.e.* the number of elements that a vector can hold before a program must allocate more memory
- `size()` - returns the currently filled level, which is always equal to or less than the capacity
- `reserve()` - increases the capacity of a vector to the number supplied as an argument

STL - Vector

Performance

Vectors grow dynamically, and every vector has a specific size. When we add a new element to a vector, the computer reallocates memory and may even copy all of the vector elements into this new memory! This can cause a performance hit.

- `capacity()` - returns the capacity of the vector, *i.e.* the number of elements that a vector can hold before a program must allocate more memory
- `size()` - returns the currently filled level, which is always equal to or less than the capacity
- `reserve()` - increases the capacity of a vector to the number supplied as an argument

```
vector<int> v;
for(int i = 0; i < 1000; ++i)
    v.push_back(i);
```

- Requires 2-18 reallocations!

STL - Vector

Performance

Vectors grow dynamically, and every vector has a specific size. When we add a new element to a vector, the computer reallocates memory and may even copy all of the vector elements into this new memory! This can cause a performance hit.

- `capacity()` - returns the capacity of the vector, *i.e.* the number of elements that a vector can hold before a program must allocate more memory
- `size()` - returns the currently filled level, which is always equal to or less than the capacity
- `reserve()` - increases the capacity of a vector to the number supplied as an argument

```
vector<int> v;
for(int i = 0; i < 1000; ++i)
    v.push_back(i);
```

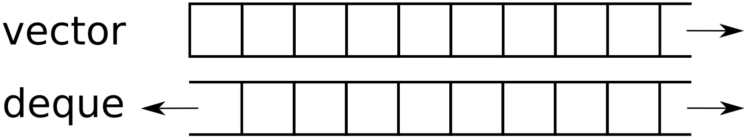
- Requires 2-18 reallocations!

```
vector<int> v;
v.reserve(1000);

for(int i = 0; i < 1000; ++i)
    v.push_back(i);
```

- Reduces cost

STL - Deque - Double-ended-queue

Usage



STL - Deque - Double-ended-queue

Usage

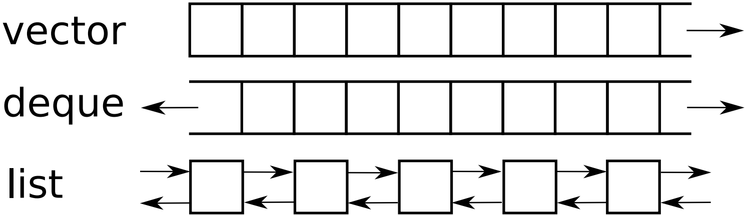
```cpp
deque<int> deq = {4, 6, 7};        // C++11 container initialisation
deq.push_front(2);                 // deq: {2, 4, 6, 7}
deq.push_back(3);                  // deq: {2, 4, 6, 7, 3}

// Similar interface with vector
cout << deq[1];                    // 4
```

Properties of STL-Deque:

- fast insert/remove at beginning and end: $\mathcal{O}(1)$
- slow insert/remove at the middle: $\mathcal{O}(n)$
- slow search: $\mathcal{O}(n)$
- Not contigous in memory

STL - List

Usage



STL - List

Usage

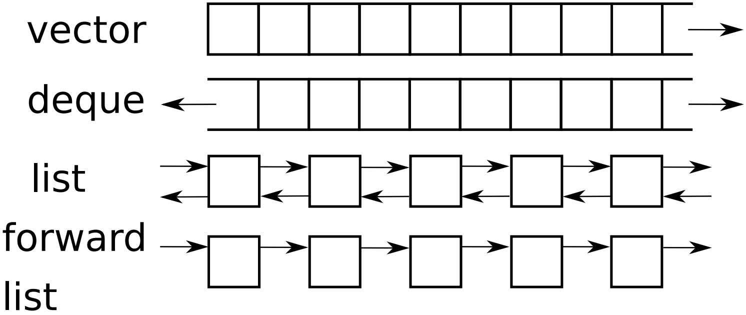
```
list<int> l = {5, 2, 9} ;        // C++11 container initialisation
l.push_back(6);                  // l: {5, 2, 9, 6}
l.push_front(4);                 // l: {4, 5, 2, 9, 6}

list<int>::iterator itr = find(l.begin(), l.end(), 2); // itr -> 2
l.insert(itr,8);                 // l: {4, 5, 8, 2, 9, 6}
                                 // O(1), faster than vector
itr++;                           // itr -> 9
l.erase(itr);                    // l: {4, 5, 8, 2, 6} O(1)

// Main reason to use list:
myIist1.splice(itr, myList2, itr_a, itr_b);  // O(1)
```

Properties of STL-List:

- fast insert/remove at any place: $\mathcal{O}(1)$
- slow search: $\mathcal{O}(n)$, slower than vector!
- no random access, no [] operator
- Not contigous in memory
- Unique killer feature is splice $\mathcal{O}(1)$

STL - List

Usage



STL - Array
Usage

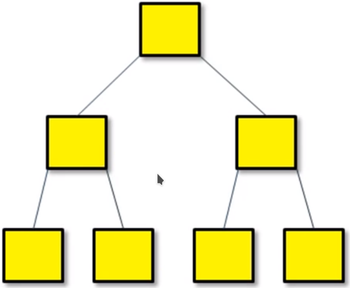
```
int a[3] = {3,4,5};
a.begin();   // not existing!
a.end();     // not existing!
a.size();    // not existing!
a.swap();    // not existing!

// Solution
array<int,3> b =  {3,4,5};
b.end();
b.size();
b.swap();
```

Properties of STL-Array:

- Thin layer around C-Array
- Provides STL functionalty
- Size is fixed
- Size is deduced in the type: `array<int,3>` is not the same type as `array<int,4>`

Associative Containers

STL - Set / Multiset



- Implemented with binray tree (red-black tree [1])
- Always sorted, default criteria is $<$, but can be freely choosen

[1] https://en.wikipedia.org/wiki/Red-black_tree

STL - Set / Multiset

```cpp
set<int> myset;
myset.insert(3);     // myset: {3}
myset.insert(1);     // myset: {1, 3}
myset.insert(7);     // myset: {1, 3, 7}, O(log(n))

// Note: find function
set<int>::iterator it;
it = myset.find(7);  // O(log(n)), it points to 7

// Check insertion
pair<set<int>::iterator, bool> ret;
ret = myset.insert(3); // no new element inserted

if (ret.second==false)
   it=ret.first;        // "it" now points to element 3
```

Properties of STL-Set:

- Sorting at insertion
- No dublicates allowed
- `insert()` ($\mathcal{O}(\log n)$)
- insertion checkable
- Fast `find()` function availabe ($\mathcal{O}(\log n)$)

STL - Set / Multiset

```
// Insert with hint: O(log(n)) => O(1)
myset.insert(it, 9);  // myset:  {1, 3, 7, 9}
                      // it points still to 3

myset.erase(it);           // myset:  {1, 7, 9}

// Note: erease by element
myset.erase(7);    // myset:  {1, 9}
```

Advanced properties of STL-Set:

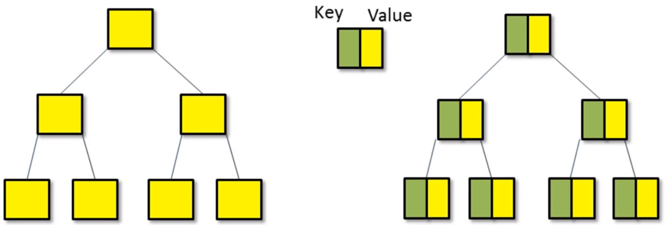
- Fast insertion with hint ($\mathcal{O}(1)$))
- `erase()` by iterator
- `erase()` by value!
- Note: `set<int>::iterator` are read only!
- Traversing is slow comp. to `vector & deque`
- No random access with `[]` operator

STL-Multiset

`Multiset` works the same as `set`, but allows dublicated items

STL - Map / Multimap

Usage



- Have **key** / **value** pairs
- Implemented with binray tree (red-black tree [1])
- Always sorted, default criteria is $<$, but can be freely choosen
- Items are sorted by **key**

[1] https://en.wikipedia.org/wiki/Red-black_tree

STL - Map / Multimap

Usage

```
map<char,int> mymap;

// Init
mymap.insert( pair<char,int>('a',100) );
mymap.insert( make_pair('z',200) ); // types detected

map<char,int>::iterator it = mymap.begin();
mymap.insert(it, pair<char,int>('b',300));   // "it" is a hint

it = mymap.find('z');   // O(log(n))

// showing contents:
for ( it=mymap.begin() ; it != mymap.end(); it++ )
cout << (*it).first << " => " << (*it).second << endl;
```

Properties of STL-Map:

- Sorting at insertion
- No dublicated **keys** allowed
- `insert()` ($\mathcal{O}(\log n)$)
- insertion checkable
- Fast `find()` function availabe ($\mathcal{O}(\log n)$)

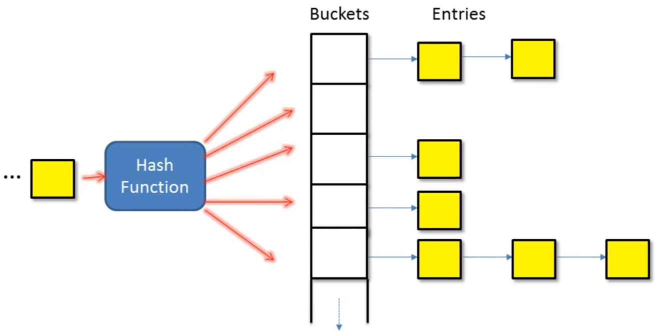
STL-Multimap

`Multimap` works the same as `map`, but allows dublicated `keys`

Unordered Containers

STL - Unordered Set / Unordered Multiset

Usage



- The order of elements is not defined and may change over time
- Implemented with *hash tables* (which is an array (*buckets*) of linked-list (*entries*))
- The hash function is used for insert and search
- Default hash function defined for fundamental types and string
- Finding an element is very fast/ fastest among all containers ($\mathcal{O}(1)$)

STL - Unordered Set / Unordered Multiset

Usage

```
unordered_set<string> myset = { "red","green","blue" };
unordered_set<string>::const_iterator itr = myset.find("green");
// Note: O(1) -fastest search possible!

if (itr != myset.end())   // Important check
    cout << *itr << endl;
myset.insert("yellow");  // O(1)

vector<string> vec = {"purple", "pink"};
myset.insert(vec.begin(), vec.end());
```

Properties of STL-Unordered Set:

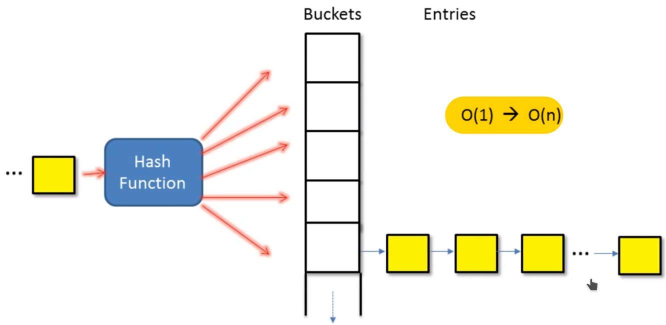
- Sorting at insertion
- No subscript `operator[]` or `at()`
- No `push_back()`, `push_front()`
- Order of element values can not be changed

STL-Unordered Multiset

`Unordered Multiset` works the same as `unorderd set`, but allows dublicated `values`

STL - Unordered Set / Unordered Multiset

Hash Collision



- In the worst case, all elements are inserted into one bucket!
- Search performance degrades from $\mathcal{O}(1) \rightarrow \mathcal{O}(n)$

STL - Unordered Set / Unordered Multiset

Hash Collision

- Hash specific API's

```
// Hash table specific APIs:
cout << "load_factor = " << myset.load_factor() << endl;

string x = "red";
cout << x << " is in bucket #" << myset.bucket(x) << endl;
cout << "Total bucket #" << myset.bucket_count() << endl;
```

STL - Unordered Map Example

Associative Array

```cpp
unordered_map<char,string> day = {{'S',"Sunday"},{'M',"Monday"}};

cout << day['S'] << endl;     // No range check
cout << day.at('S') << endl; // Has range check

vector<int> vec = {1, 2, 3};
vec[5] = 5;    // Compile Error

day['W'] = "Wednesday";   // Inserting {'W', "Wednesday}
day.insert(make_pair('F', "Friday"));   // Insert {'F',"Friday"}

day.insert(make_pair('M', "MONDAY"));   // Fail to modify
day['M'] = "MONDAY";                    // Succeed to modify
```

- Associative array can be implemented using `map` or `multimap`
    1. test
    2. Search time:
       `unordered_map`: $\mathcal{O}(1)$
       `map`: $\mathcal{O}(log(n))$
    3. `unordered_map` may degrade to $\mathcal{O}(n)$, `map` guarantees $\mathcal{O}(log(n))$!
- To modify elements use subscript operator []
- This means the subscript operator provides a write access to the container

STL - Unordered Map Example

Associative Array

```
void print(const unordered_map<char, string>& m)
{
    m['S'] = "SUNDAY";        // compile error
    cout << m['S'] << endl; // compile error

    auto itr = m.find('S');

    if (itr != m.end())
        cout << *itr << endl;
}
print(day);
```

- Use iterators to access read-only elements

Container Adapters

STL - Container Adapters
For special needs

Container Adapters are implemented using fundamental container classes, providing a restricted interface for special needs:

- `stack`: LIFO, `push()`, `pop()`, `top()`
- `queue`: FIFO, `push()`, `pop()`, `front()`, `back()`
- `priority queue`: first item always has the greatest priority, `push()`, `pop()`, `top()`

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. 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

Thank You

Questions

???