STL

01 – Overview and Containers

Contents

neueda

Overview

01 – Overview and Containers

The STL – is a C++ library of

Container classes

`Vector`, Set, Map, List etc

Algorithms

for_each, transform, find_if

Iterators

Forward, backward,
bi-directional, inserters

Utilities

Adapters – stack, queue, priority_queue,
Binders, pair, dates & times

Function objects

generators, predicates

Memory Management

New, scoped allocators

Limts

System limits

Error Handling

Exceptions & assertions

String

Narrow, wide, unicode

Numerics

Maths, Random Numbers

Concurrency

Threads, Locks, Tasks, Atomics

A Few Others

Locales, RegEx,

Header Files & Naming convention

Header files in the STL do not have the .h extension

Historically this was the case, STL headers used to have a .h extension, however, these have been deprecated

However, you may still see examples of this in code

In such situations, make every effort to replace these older files with their standard versions

i.e **`<vector>`** instead of **`<vector.h>`**
`<iostream>` instead of **`<iostream.h>`** etc

All template classes, template functions in the STL are written in
`lower_snake_case`

neueda

The STD namespace

Every component of the STL

is enclosed in the std namespace

Needs to be scoped to this in order to compile.

There are 3 approaches to this.

1 – Use the entire namespace within a scope

```
void UsingEg1()
{
    using namespace std;
    vector<int> ints;
    /// stuff
}
```

2 - Only the whats needed within a scope

```
void UsingEg2()
{
    using std::vector;
    vector<int> ints;
    /// stuff
}
```

3 - Explicitly scope as needed

```
void UsingEg3()
{
    std::vector<int> ints;
    /// stuff
}
```

Typedefs

- The syntax for templates can become very complicated and difficult to read
- To simplify code, use a typedef to give an alias to complicated data types
- Use the least amount of scope required
- Become familiar with the inbuilt typedefs in the STL

```
void NoTypeDef()
{
    std::vector<int> ints;
    std::vector<int>::const_iterator itr;
    for (itr = ints.begin(); itr != ints.end(); ++itr)
    {
        int item = *itr;
        // Do stuff
    }
}
```

```
void TypedefDemo()
{
    typedef std::vector<int> intColl;
    intColl ints;
    intColl::const_iterator itr;
    for (itr = ints.begin(); itr != ints.end(); ++itr)
    {
        intColl::value_type item = *itr;
        // Do stuff
    }
}
```

Alias Declaration

Modern C++ recommends using an **alias declaration** to create programmer friendly versions of complicated types as opposed to typedefs.

Leads to much simpler code for more advanced C++ programmer and eliminates need for complex use of typename declarations

See E M C++ - Item 9. *(Effective Modern C++ - Scott Meyers)*

```
void NoTypeDef()
{
    std::vector<int> ints;
    std::vector<int>::const_iterator itr;
    for (itr = ints.begin(); itr != ints.end(); ++itr)
    {
        int item = *itr;
        // Do stuff
    }
}
```

```
void NamespaceALiasDemo()
{
    using intColl = std::vector<int>;
    intColl ints;
    intColl::const_iterator itr;
    for (itr = ints.begin(); itr != ints.end(); ++itr)
    {
        intColl::value_type item = *itr;
        // Do stuff
    }
}
```

A Simple Use Case

Demonstrate

A container – **`vector`**

An algorithm – **`copy`**

An iterator

A utility – **`ifstream`**, **`ofstream`**

Use Case

Build a vector of People objects in memory.

Write the vector to file.

Read the file into a vector

neueda

The Header File

```
// STL Headers
#include <vector>

// Application Headers
#include "Person.h"

// Aliases
using People = std::vector<Person>;

// Function Prototypes
void DemoPeople();
void BuildPeople(People& people);
void DumpPeopleToFile(const People& people);
void LoadPeopleFromFile(People& people);
```

Build an in-memory vector

```
void BuildPeople(People& people)
{
  people.clear();

  people.emplace_back("John", "Lennon");
  people.emplace_back("Paul", "McCartney");
  people.emplace_back("George", "Harrison");
  people.emplace_back("Ringo", "Starr");
}
```

Write to File & Read From File

```
void DumpPeopleToFile(const People& people) {
      std::ofstream store("beatles.txt", std::ofstream::out);
      for (const auto& person : people)
              store << person << std::endl;
      store.close();
}
```

- ```
  void LoadPeopleFromFile(People& people) {
        people.clear();
        std::ifstream store("beatles.txt", std::ios::in);

        std::copy( std::istream_iterator<Person>(store),
                   std::istream_iterator<Person>(),
                   std::back_insert_iterator<People>(people));
        store.close();
  }
  ```
  ```

STL Containers

01 – Overview and Containers

Contents

Overview

Sequence Containers & Examples

(Ordered) Associative Containers & Examples

Unordered Associative Containers & Examples

Categories of Containers

The STL containers are divided into one of 4 categories:

Sequence containers
array, vector, deque (pronounced deck), list, forward_list

(Ordered) Associative containers
map, multimap, set, multiset

Unordered Associative containers
unordered_map, unordered_multimap, unordered_set, unordered_multiset

Container adatpers
stack, queue, priority_queue

Some Theory

The STL literature uses the terms **first-class containers** and **near containers**.

A **first class container** is a pure maths / computer science term; when used with the STL it means

Sequence Container AND Ordered Associative AND Unordered Associative

A **near container**, another pure maths / computer science term

When applied to the STL, means other containers that can be used with some or all iterators and can be manipulated using some / all STL algorithms.

Examples of STL near containers

string, bitset, valarray

Container adapters are neither first class nor near containers.

neueda

Container Methods

All STL containers have a fairly easy to understand, common and uniform interface.

In general all Containers
- are first class copyable and moveable objects.
- can be constructed with an initial size (pre-allocation)
- automatically re-size when required (with the exception of arrays)

Have methods to generate **const** and **non-const** iterators for moving forward, backwards, random access, inserting and deletion.
- Allow **insertion** – insert an existing object into a container
- Allow **emplacement** – create a brand new object into a container
- Removal

At front, back or in the middle of a container.

Other common methods implemented by most containers
- **size**, **erase**, **clear**, **begin**, **end**, **front**

Common typedefs include
- **container::value_type**, **container::pointer**, **container::size_type**

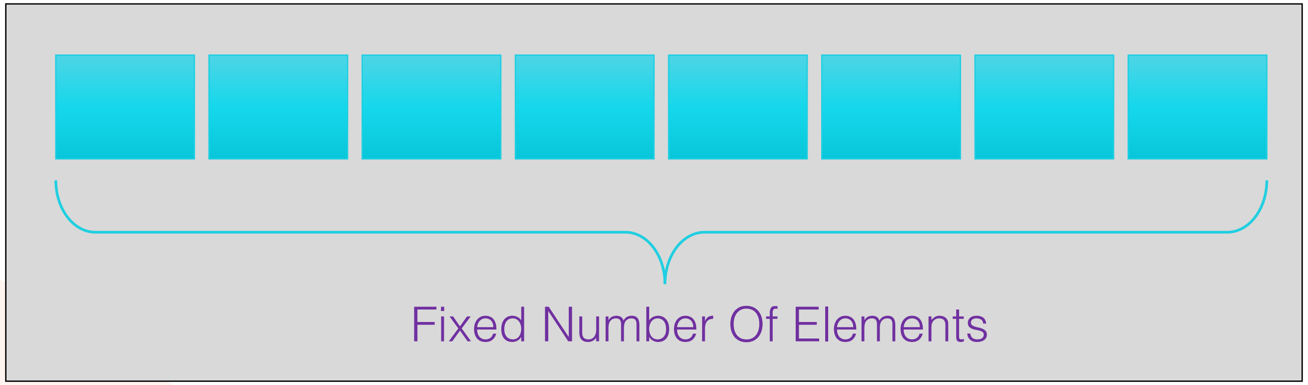
array

Contiguous block of memory

```
#include <array>
```

Encapsulates fixed sized arrays

Combines benefits of C-style array (speed, size, etc) with benefits of a standard container (use with algorithms, iterators etc)



neueda

array - Example Code

```
std::array<int, 10> arrInt { { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 } };

for (const auto& item : arrInt)
        std::cout << item << ' ' << std::endl;

arrInt[0] = 100; arrInt[2] = 100; arrInt[4] = 100; arrInt[6] = 100; arrInt[8] = 100;

Person john("John", "Lennon");
Person paul("Paul", "McCartney");
Person george("George", "Harrison");
Person ringo("Ringo", "Starr");

std::array<Person, 4> arrPeople { john, paul, george, ringo };

for (const auto& item : arrPeople)
   std::cout << item << std::endl;
```

Constructor here uses aggregate initialization
Double-braces “{{“ “}}” – Cx11

using a range based loop

aggregate initialization
single-braces only (C++14

using a range based loop

vector

Implemented as an array, a **contiguous** block of elements.

```
#include <vector>
```

Allows direct access to any element in the vector

Allows rapid insertions / deletions of items at the back of the vector

Automatically grows when required.

vector- Example Code

```
std::vector<Person> vecBeatles { john, paul, george, ringo };

std::cout <<"Number of Beatles: " << vecBeatles.size() << std::endl;

if (!vecBeatles.empty())
        vecBeatles.clear();

std::vector<Person> anotherVector;
anotherVector.push_back(john);
anotherVector.push_back(paul);

vecBeatles = anotherVector;

if (vecBeatles == anotherVector)
        vecBeatles.push_back(Person("Eric", "Clapton"));

std::cout << vecBeatles[1] << std::endl;
std::cout << vecBeatles.at(2) << std::endl;
```

Create a vector

How many elements

Checks if empty

Clears vector

push_back to add an item to end of vector

assignment

equivalence

Array-like syntax

At member function

list

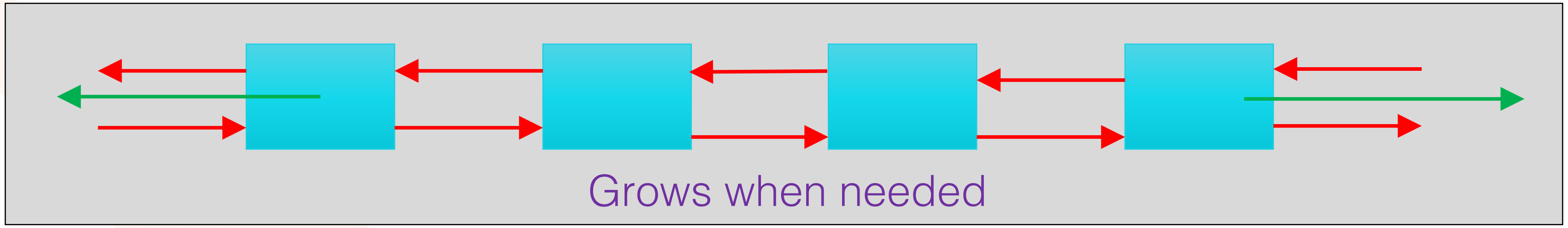
Implemented as a **doubly linked list**

```
#include <list>
```

Allows liner access to any element

Compatible with bi-directional iterators

Allows rapid insertions / deletions anywhere in the list



neueda

```cpp
std::vector<Person> vecBeatles { john, paul, george, ringo };

std::list<Person> listBeatles(vecBeatles.begin(),vecBeatles.end());

listBeatles.push_back(Person("Eric", "Clapton"));

listBeatles.push_front(Person("Bob", "Dylan"));

std::cout << listBeatles.front() << std::endl;
listBeatles.pop_front();
std::cout << listBeatles.front() << std::endl;

std::cout << listBeatles.back() << std::endl;
listBeatles.pop_back();
std::cout << listBeatles.back() << std::endl;
```

instantiate using the range constructor

Insert item at front of list

front and **pop_front**

back and **pop_back**

deque

Best thought of as a **linked list of arrays / vectors**

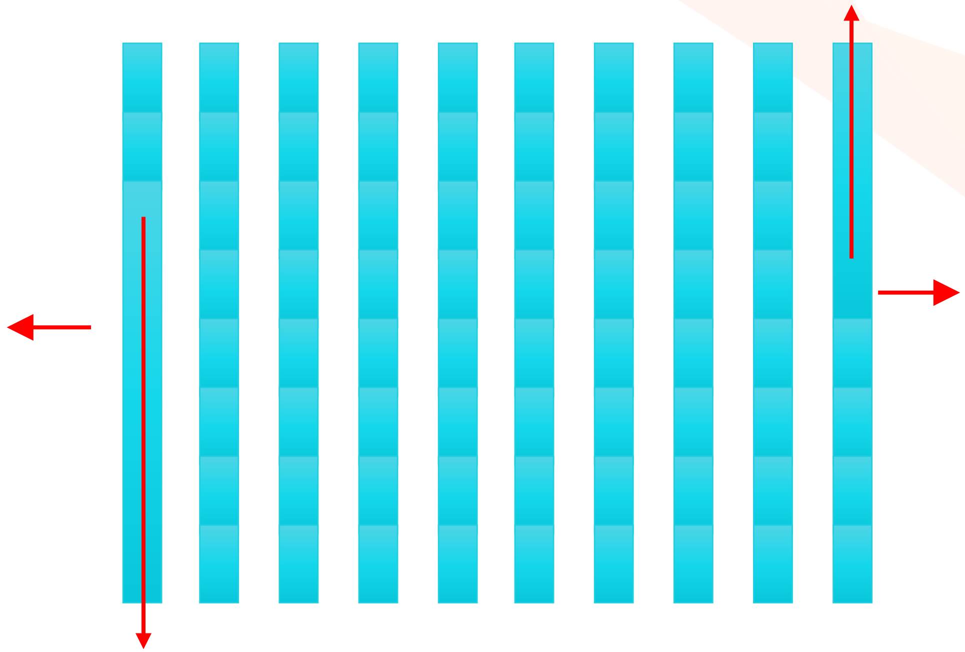
```
#include <deque>
```

Standard guarantees constant time access to elements

Efficiency is very much dependent on the STL implementation being used.

Allows rapid insertions / deletions of items at the back of the deque

Automatically grows when required.



Internal Structure of a Deque

deque – Example Code

```
std::deque<Person> deqBeatles;

deqBeatles.emplace_front(john);
deqBeatles.emplace_front( Person { "Paul", "McCartney" } );


deqBeatles.emplace_back(george);
deqBeatles.emplace_back( Person ( "Ringo", "Starr" ) );

for (const auto& item : deqBeatles)
        std::cout << item << ' ' << std::endl;


std::vector<Person> vecBeatles(begin(deqBeatles),end(deqBeatles));
```

Using emplacement to add items to front

Using emplacement to add items to back

using a range based constructor but using the free functions begin and end as opposed to the beginning and end member functions

neueda

forward_list

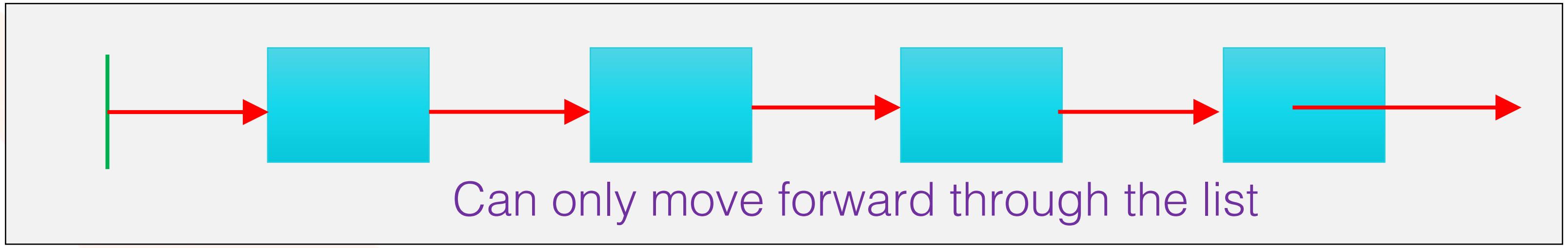
Implemented as a **singly linked list**

```
#include <forward_list>
```

Allows liner access to any element

Allows rapid insertions / deletions anywhere in the list

Provides more space efficient storage as bi-directional iteration not needed



neueda

forward_list - Example Code

```
typedef std::forward_list<Person> Beatles;

Beatles beatles;

// …

Beatles::value_type person = beatles.front();

std::cout          << "First Name = " << person.GetFirstName()
                   << " Last Name = " << person.GetLastName()
                   << std::endl;

Beatles::pointer pPerson = &beatles.front();

std::cout          << "First Name = " << pPerson->GetFirstName()
                   << " Last Name = " << pPerson->GetLastName()
                   << std::endl;
```

Using a typedef to simplify code later on

Beatles::value_type is a **Person**

Beatles::pointer is a **Person***

neueda

set

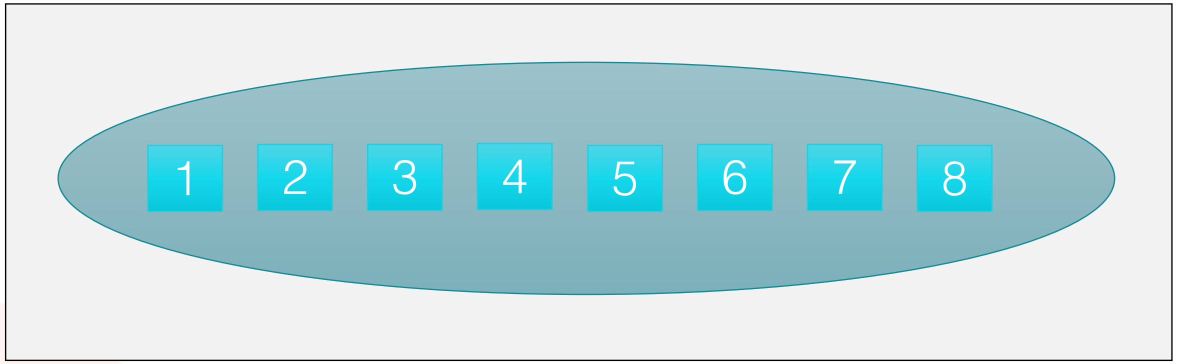
Implemented as a **balanced red/black binary tree**

```
#include <set>
```

Contain a **sorted** set of **unique** objects

No matter how often an item is inserted into a set, the set will only ever contain 1 instance of it.

Objects must be **comparable** (default is to allow < - less than) but comparison operation can be specialised if needed



neueda

set – Example Code

```
std::set<Person> setBeatles{ john, paul, george, ringo };

std::cout << "Number of Beatles " << setBeatles.size() << std::endl;

setBeatles.insert(john);
setBeatles.insert(paul);

if (setBeatles.insert(george).second)
        std::cout << "Added " << george << std::endl;
if (!setBeatles.insert(ringo).second)
        std::cout << "Did not add " << ringo << std::endl;


std::cout << "Number of Beatles " << setBeatles.size() << std::endl;
for (const auto& beatle : setBeatles)
        std::cout << beatle << ' ' << std::endl;
```

Add some more without testing for insertion

Add some more with testing for insertion
`set::insert` returns a **`std::pair`**
`.second` member of pair is a Boolean indicating whether insertion was successful or not

Still 4 Beatles

multiset

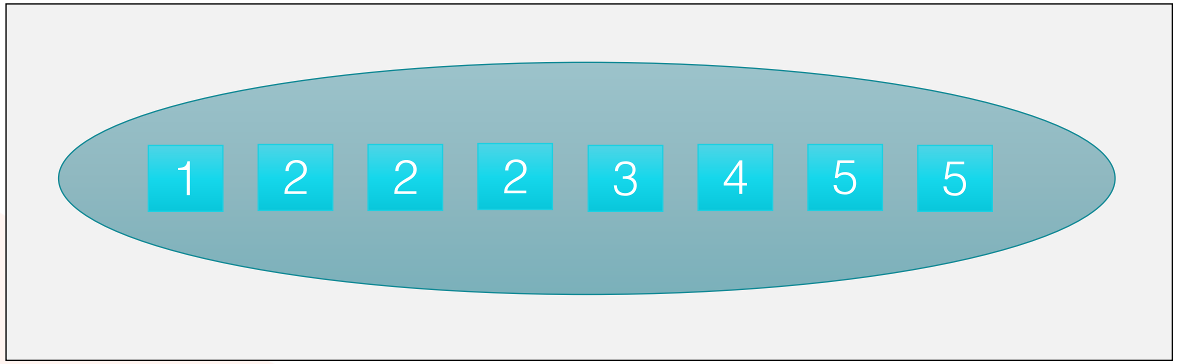
Implemented as a **balanced red/black binary tree**

```
#include <set>
```

Contain a **sorted** set of objects where multiple elements can have equivalent values

Resulting effect of this is that **duplicate items are allowed**

Objects must be **comparable** (default is to allow < - less than) but this can be customized if needed (same as set)



neueda

multiset - Example Code

```
std::multiset<Person> multisetBeatles { john, paul, george, ringo };

std::cout << "Number of Beatles " << multisetBeatles.size() << std::endl;

multisetBeatles.insert(john);    multisetBeatles.insert(paul);
multisetBeatles.insert(george); multisetBeatles.insert(ringo);

std::cout << "Number of Beatles " << multisetBeatles.size() << std::endl;

multisetBeatles.erase(paul);

std::cout << "Number of Beatles " << multisetBeatles.size() << std::endl;

multisetBeatles.clear();

std::cout << "Number of Beatles " << multisetBeatles.size() << std::endl;
```

4 Beatles

Add some more Beatles

8 Beatles

Both Paul McCartneys have been erased

6 Beatles

0 Beatles

And the Beatles broke up

neueda

map

Implemented as a **balanced red/black binary tree**

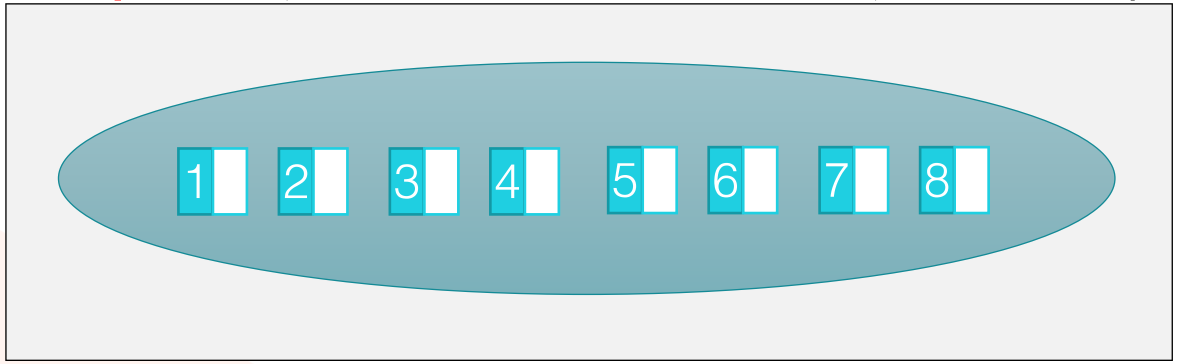
#include <map>

Contain a **sorted** collection of **key/value** pairs

Maps allow only **unique keys**

The types used for keys and values can differ (more often than not they do differ)

Items used as keys must be **comparable** (default is to allow < - less than) but can be specialised if needed



map – Example Code

```
using BeatleMap = std::map<const std::string, Person>;

BeatleMap mapBeatles;
// …

mapBeatles.insert( { "JL", john } );
mapBeatles.emplace("PMC", paul);
mapBeatles.insert(BeatleMap::value_type("GH", george));
mapBeatles.insert(std::pair<std::string, Person>("RS", ringo));
mapBeatles.insert(std::make_pair("BD", bob));
mapBeatles["EC"] = eric;

std::cout << "Number of Beatles " << mapBeatles.size() << std::endl;

for (const auto& beatle : mapBeatles)
        std::cout << beatle.first << " ==> " << beatle.second << std::endl;
```

Alias declaration as opposed to **`typedef`**

How to Add some beatles to a map

Using insert

Using emplace

Using **`value_type`**

Using a **`pair`** directly

Using a **`make_pair`**

Using array-like syntax

Print out key ==> value pairs of entire map.
Beatle is an iterator
First is the key
Second is the value

multimap

Implemented as a **balanced red/black binary tree**

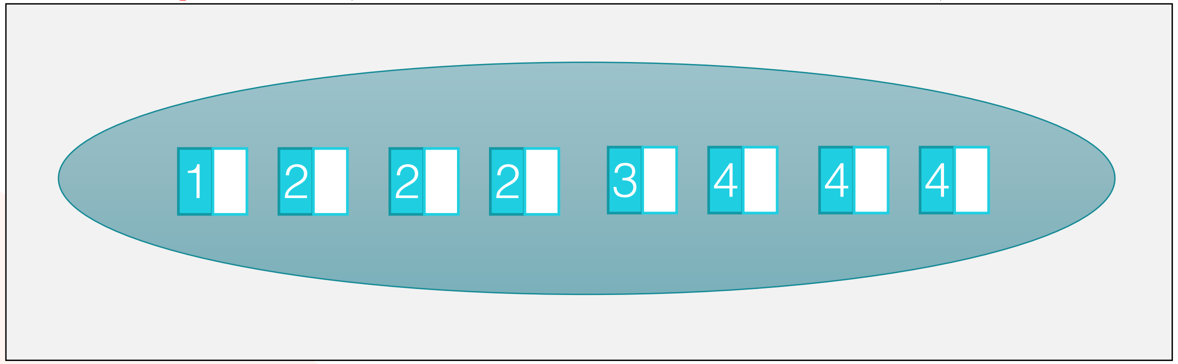
```
#include <map>
```

Contain a **sorted** collection of **key/value pairs**

multiaps allow **duplicate** keys

The types used for keys and values can differ (usually the case)

Items used as keys must be **comparable** (default is to allow < - less than) but can be specialised if needed



neueda

multimap – Example Code (1 of 2)

```
using BeatlesMultimap = std::multimap<std::string, Person>;

BeatlesMultimap multimapBealtes;

Person john("John", "Lennon");           Person paul("Paul", "McCartney");
Person george("George", "Harrison");     Person ringo("Ringo", "Starr");
Person julian("Julian", "Lennon");       Person yoko("Yoko", "Ono");
Person lynda("lynda", "McCartney");

multimapBealtes.insert(std::make_pair("Lennons", john));
multimapBealtes.insert(std::make_pair("McCartneys", paul));
multimapBealtes.insert(std::make_pair("Harrisons", george));
multimapBealtes.insert(std::make_pair("Starrs", ringo));
multimapBealtes.insert(std::make_pair("Lennons", julian));
multimapBealtes.insert(std::make_pair("Lennons", yoko));
multimapBealtes.insert(std::make_pair("McCartneys", lynda));
```

Create a multimap

Create some people

Add the people to the **`multimap`**

Note the duplicate keys – multiple **`Lennons`** and multiple **`McCartneys`**

neueda

`multimap` - Example Code (2 of 2)

```
std::cout << "Number of Beatles " << multimapBealtes.size() << std::endl;
// Print out key ==> value pairs of entire map

for (const auto& beatle : multimapBealtes)
        std::cout << beatle.first << " ==> " << beatle.second << std::endl;



for (auto& itr = multimapBealtes.lower_bound("Lennons");
                itr != multimapBealtes.upper_bound("Lennons");
                ++itr)

        std::cout        << itr->first << " ==> "
                          << (*itr).second
                         << std::endl;
```

Print out key ==> value pairs of entire map

Just the Lennons

NOTE the **`lower_bound`** and **`upper_bound`** member functions.

These return the first and last elements with the key "Lennon"

As iterator in maps and multimaps is a pointer to an **`std::pair`**, with member variable **`first`** and **`second`**
It can be used with **->** notation OR **(*).** notation

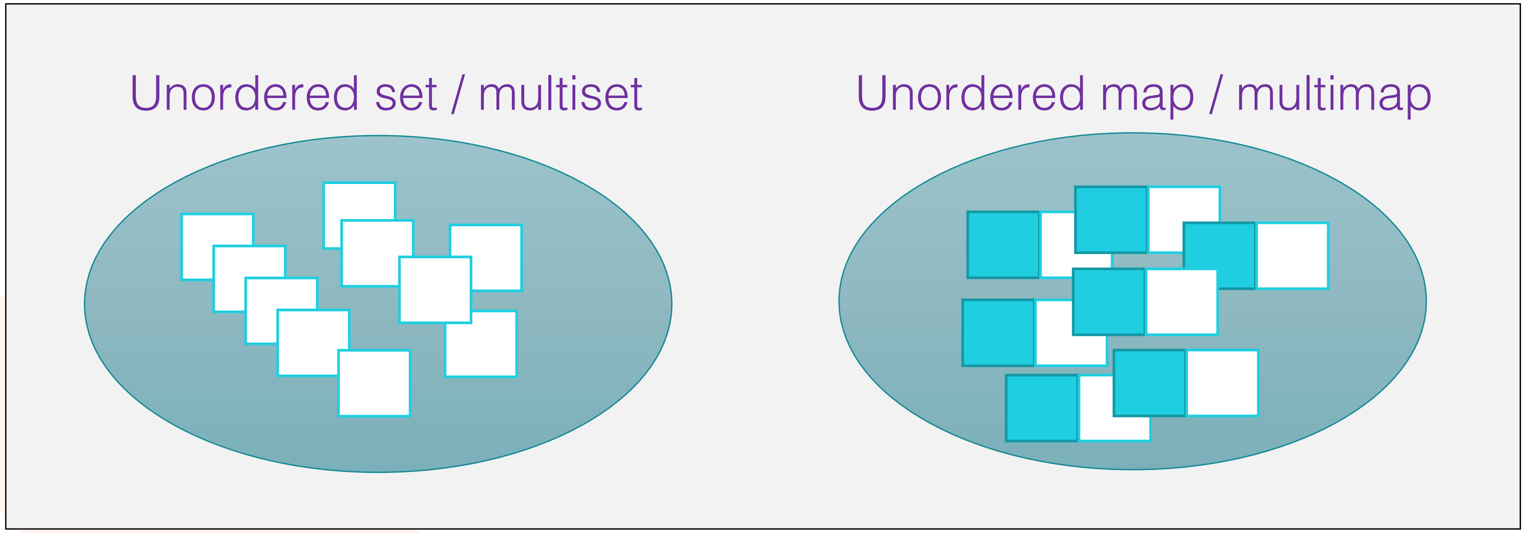
Unordered Containers

The unordered containers in the STL are hash table variants of set/multiset and map/multimap.

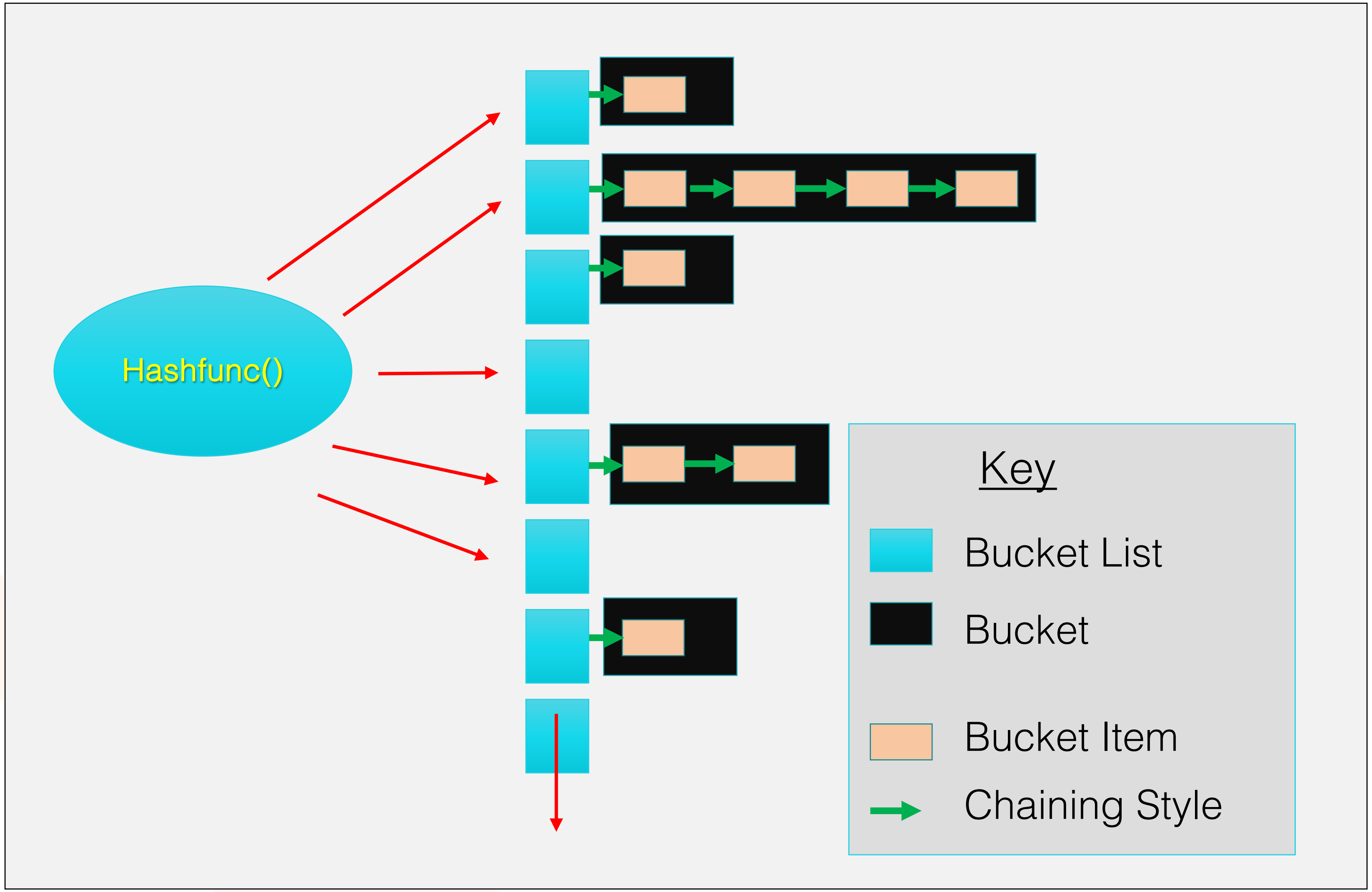
Conceptually, unordered containers contain the same elements as their ordered counterparts except they do not order them.

Unordered sets & multisets store single values, unordered maps and multimaps store key/value pairs.

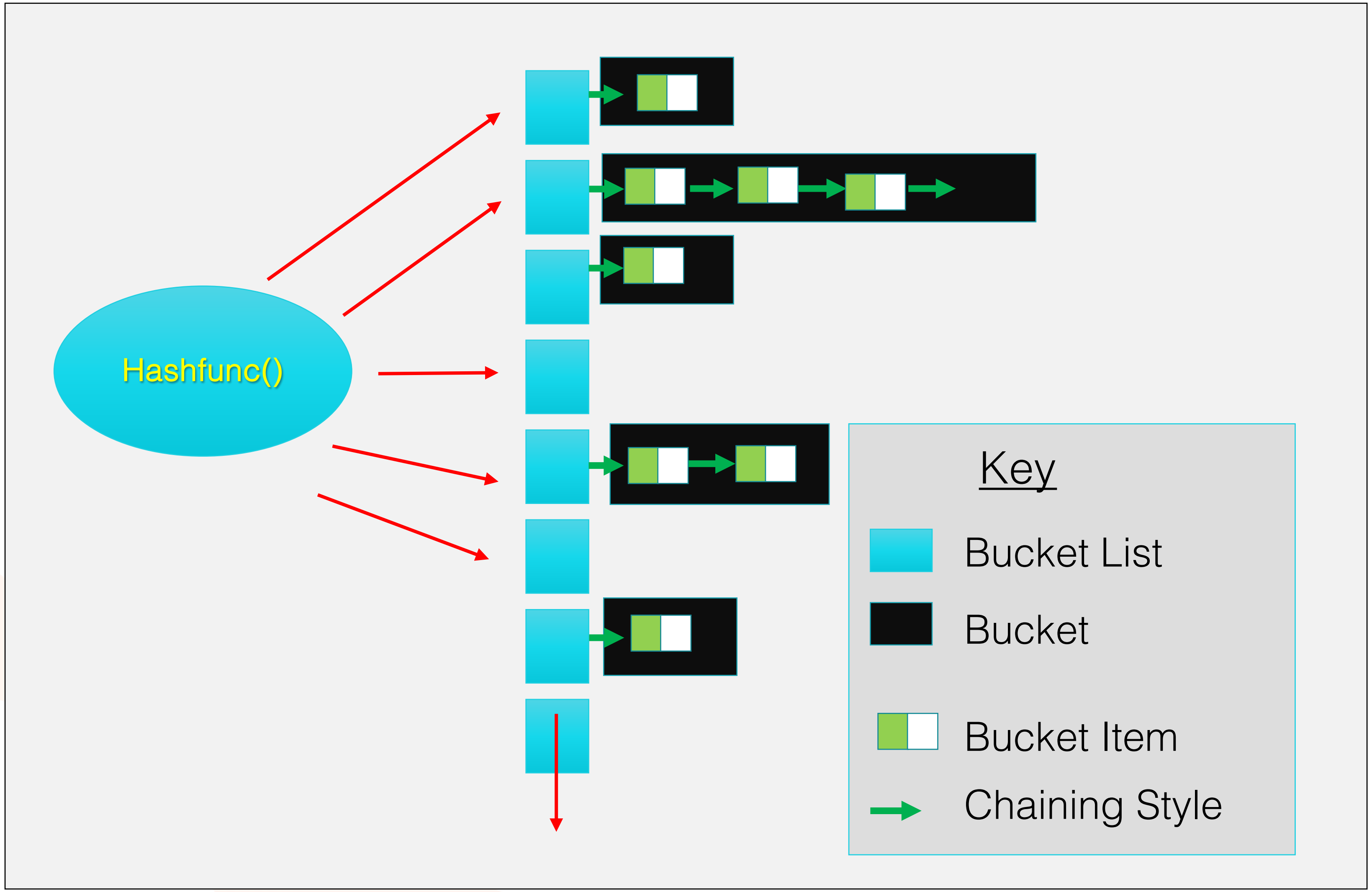
Unordered sets and maps do not allow duplicates, unordered multisets and multimaps do.



Internal Storage of unordered `set/multiset`



Internal Storage of unordered `map/multimap`



The Bucket Interface

The unordered containers expose the same sort of interfaces as their ordered counterparts.

In addition, unordered containers expose two further sets of methods

Buckets	Hash Policy
`bucket_count` Return the number of buckets	**`load_factor`** Return the load factor
`max_bucket_count` Return the maximum number of buckets	**`max_load_factor`** Get or Set maximum load factor
`bucket_size` Return the bucket size	**`rehash`** Set the number of Buckets
`bucket` Locate elements bucket	**`reserve`** Request a capacity change

Using The Bucket Interface

This example shows the behaviour of

creating a simple **`unordered_map`**
examining its internal structure
adding some new elements
examining its internal structure a second time to see the types of changes.

The output shows the internal default hashing function using the following compilers
VC++ 2015
LLVM (CLANG) running on Apple

Using The Bucket Interface – Driver Function

```
std::unordered_multimap<std::string, std::string> colours = {
        { "Red", "Richard" },     { "Orange", "Of" },       { "Yellow", "York" },
        { "Green", "Gave" },      { "Blue", "Battle" },     { "Indigo", "In" },
        { "Violet", "Vain" }
};

display_hash_table_state(colours);

colours.insert( {
        { "Red", "Run" },         { "Orange", "Off" },      { "Yellow", "You" },
        { "Green", "Great" },     { "Blue", "Big" },                { "Indigo", "Irish"
},
        { "Violet", "Vagabond" }
});

display_hash_table_state(colours);

colours.max_load_factor(0.7);

display_hash_table_state(colours);
```

Initial State

After insertions

After changing load factor

Using The Bucket Interface – Helper Templates

```
template <typename K, typename V>
std::ostream& operator << (std::ostream& os, const std::pair<K, V>& p)
{
        return os << "[" << p.first << "," << p.second << "]";
}
```


neueda

Using The Bucket Interface – Helper Templates

```
template <typename T>
void display_hash_table_state(const T& container)
{
    std::cout << "size:                                  " << container.size() <<
    std::endl;
    std::cout << "buckets:                                " << container.bucket_count()
    << std::endl;
    std::cout << "load factor:                   " << container.load_factor() <<
    std::endl;
    std::cout << "max load factor:               " << container.max_load_factor() <<
    std::endl;

    if ( typeid(typename std::iterator_traits
                <typename T::iterator>::iterator_category) ==
        typeid(std::bidirectional_iterator_tag)              )
        std::cout << "chaining style:   doubly-linked" << std::endl;
    else std::cout << "chaining style:singly-linked" << std::endl;

    std::cout << "data:" << std::endl;
    for (auto idx = 0; idx != container.bucket_count(); ++idx) {
        std::cout << "bucket [" << std::setw(2) << idx << "]:";
        for (auto pos = container.begin(idx); pos != container.end(idx); ++pos)
                std::cout << *pos << " ";
        std::cout << std::endl;
    }
}
```

How to determine if singly linked or doubly linked

Loop down each bucket

Loop across the elements of each bucket

neueda

The output – Initial State

MS VC++ - Vis Studio 2015

```
size:               7
buckets:            8
load factor:        0.875
max load factor:    1
chaining style:     doubly-linked
data:
        bucket [ 0]:
        bucket [ 1]:        [Yellow,York]
        bucket [ 2]:        [Violet,Vain]
        bucket [ 3]:        [Indigo,In] [Oran
        bucket [ 4]:        [Green,Gave] [R
        bucket [ 5]:        [Blue,Battle]
        bucket [ 6]:
        bucket [ 7]:
```

LLVM (on Apple Xcode 6.3)

```
size:               7
buckets:            11
load factor:        0.636364
max load factor:    1
chaining style:     singly-linked
data:
        bucket [ 0]:        [Indigo,In]
        bucket [ 1]:        [Green,Gave]
        bucket [ 2]:        [Blue,Battle]
        bucket [ 3]:
        bucket [ 4]:        [Yellow,York]
        bucket [ 5]:
        bucket [ 6]:
        bucket [ 7]:        [Orange,Of]
        bucket [ 8]:        [Red,Richard] [Violet,Vain]
        bucket [ 9]:
        bucket [10]:
```

In addition to differences highlights
Note the difference in ordering of items
And in bucket contents

neueda

The output – After Insertions

MS VC++ - Vis Studio 2015	LLVM (on Apple Xcode 6.3)

```
size:               14
buckets:            64
load factor:        0.21875
max load factor:  1
chaining style:   doubly-linked
data:
     .....
     bucket [11]:     [Orange,Of] [Orange,Off] [Indigo,In]
     [Indigo,Irish]
     .....
     bucket [18]:     [Violet,Vain] [Violet,Vagabond]
     .....
     bucket [28]:     [Green,Gave] [Green,Great]
     .....
     bucket [41]:     [Yellow,York] [Yellow,You]
     .....
     bucket [45]:     [Blue,Battle] [Blue,Big]
     .....
     bucket [60]:     [Red,Richard] [Red,Run]
     .....
```

```
size:           14
buckets:          23
load factor:      0.608696
max load factor:  1
chaining style:        singly-linked
data:
          .....
          bucket [ 2]:  [Red,Richard] [Red,Run]
          .....
          bucket [ 4]:  [Blue,Battle] [Blue,Big]
          .....
          bucket [ 6]:  [Violet,Vain] [Violet,Vagabond]
          .....
          bucket [10]: [Orange,Of] [Orange,Off]
          .....
          bucket [12]: [Yellow,York] [Yellow,You]
          .....
          bucket [20]: [Indigo,In] [Indigo,Irish]
          bucket [21]: [Green,Gave] [Green,Great]
          .....
```

The output – After Changing Load Factor

MS VC++ - Vis Studio 2015

```
size:               14
buckets:            64
load factor:        0.21875
max load factor:  0.7
chaining style:   doubly-linked
data:
      .....
      bucket [11]:        [Orange,Of] [Orange,Off] [Indigo,In]
      [Indigo,Irish]
      .....
      bucket [18]:        [Violet,Vain] [Violet,Vagabond]
      .....
      bucket [28]:        [Green,Gave] [Green,Great]
      .....
      bucket [41]:        [Yellow,York] [Yellow,You]
      .....
      bucket [45]:        [Blue,Battle] [Blue,Big]
      .....
      bucket [60]:        [Red,Richard] [Red,Run]
      .....
```

LLVM (on Apple Xcode 6.3)

```
size:             14
buckets:          23
load factor:      0.608696
max load factor:  0.7
chaining style:         singly-linked
data:
          .....
          bucket [ 2]:  [Red,Richard] [Red,Run]
          .....
          bucket [ 4]:  [Blue,Battle] [Blue,Big]
          .....
          bucket [ 6]:  [Violet,Vain] [Violet,Vagabond]
          .....
          bucket [10]: [Orange,Of] [Orange,Off]
          .....
          bucket [12]: [Yellow,York] [Yellow,You]
          .....
          bucket [20]: [Indigo,In] [Indigo,Irish]
          bucket [21]: [Green,Gave] [Green,Great]
          .....
```

?