Imperial College London

Advanced Programming

Standard Template Library

Containers, Operators, Iterators

Adriana Paluszny

[Many of the slides are adapted from the original Bjarne Stroustrup slides on C++]

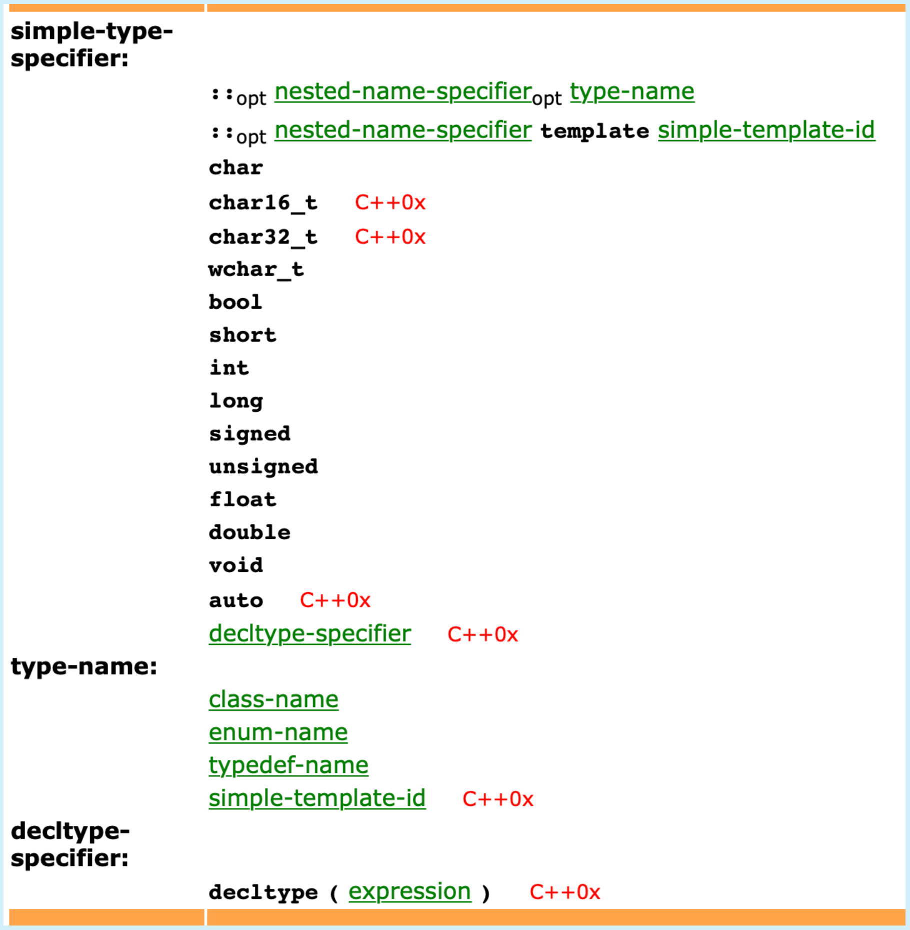
Objects

- These are ways of creating new types beyond the native types

- Create objects:
  - class
  - enum
  - typedef
  - And… templates

These are "typename"s.

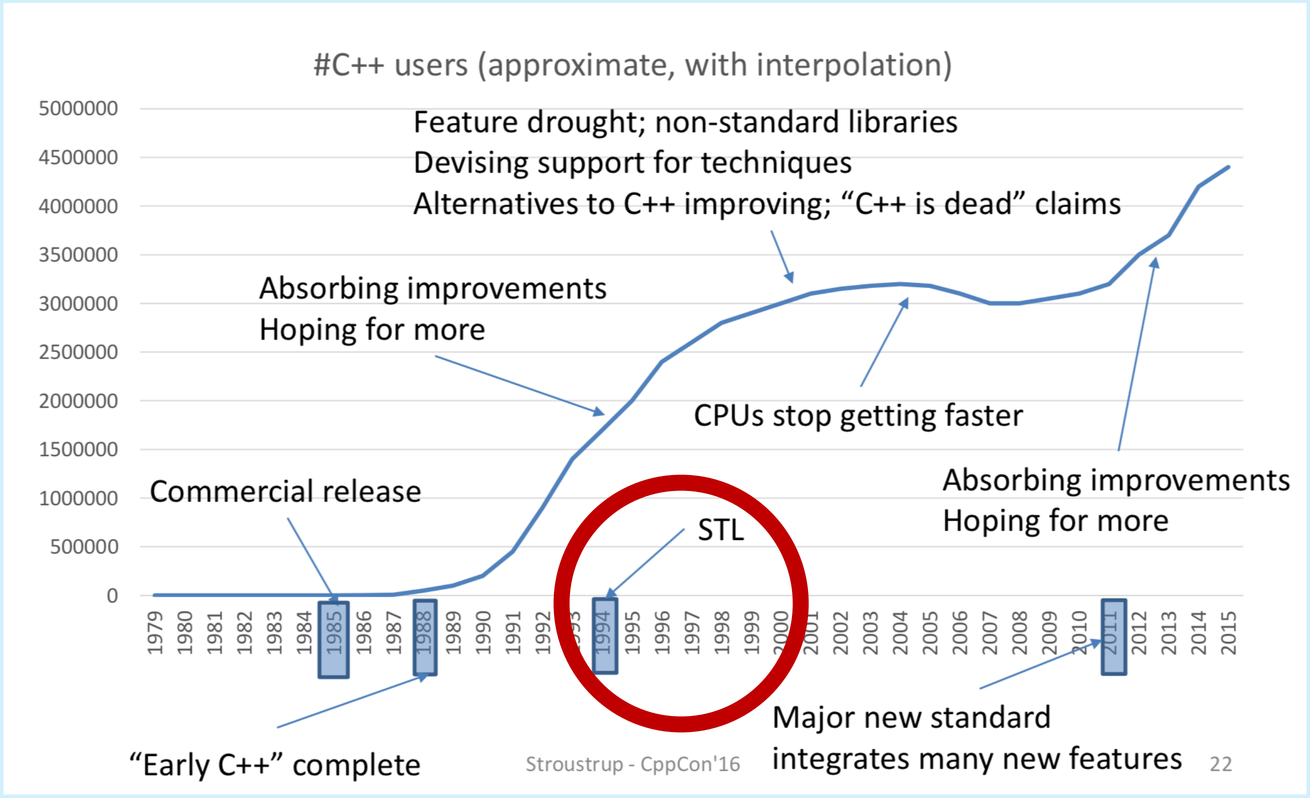
from the c++ grammar



```
simple-type-specifier:
        ::opt nested-name-specifieropt type-name
        ::opt nested-name-specifier template simple-template-id
        char
        char16_t    C++0x
        char32_t    C++0x
        wchar_t
        bool
        short
        int
        long
        signed
        unsigned
        float
        double
        void
        auto    C++0x
        decltype-specifier    C++0x
type-name:
        class-name
        enum-name
        typedef-name
        simple-template-id    C++0x
decltype-specifier:
        decltype ( expression )    C++0x
```

http://www.nongnu.org/hcb/

“C++ Success” by Bjarne Stroustrup



Imperial College
London

Common tasks

- Collect data into containers
- Organize data
  - For printing
  - For fast access
- Retrieve data items
  - By index (e.g., get the **N**th element)
  - By value (e.g., get the first element with the value **"Chocolate"**)
  - By properties (e.g., get the first elements where "**age<64**")
- Add data
- Remove data
- Sorting and searching
- Simple numeric operations

Imperial College London

Aims

- We aim to write code that is
  - Easy to read
  - Easy to modify
  - Regular
  - Short
  - Fast
- Uniform access to data
  - Independently of how it is stored
  - Independently of its type
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast
  - Retrieval of data
  - Addition of data
  - Deletion of data
- Standard versions of the most common algorithms
  - Copy, find, search, sort, sum, …

Imperial College
London

Generic programming

- Generalize algorithms
  - Sometimes called “lifting an algorithm”
- The aim (for the end user) is
  - Increased correctness
    - Through better specification
  - Greater range of uses
    - Possibilities for re-use
  - Better performance
    - Through wider use of tuned libraries
    - Unnecessarily slow code will eventually be thrown away
- Go from the concrete to the more abstract
  - The other way most often leads to bloat

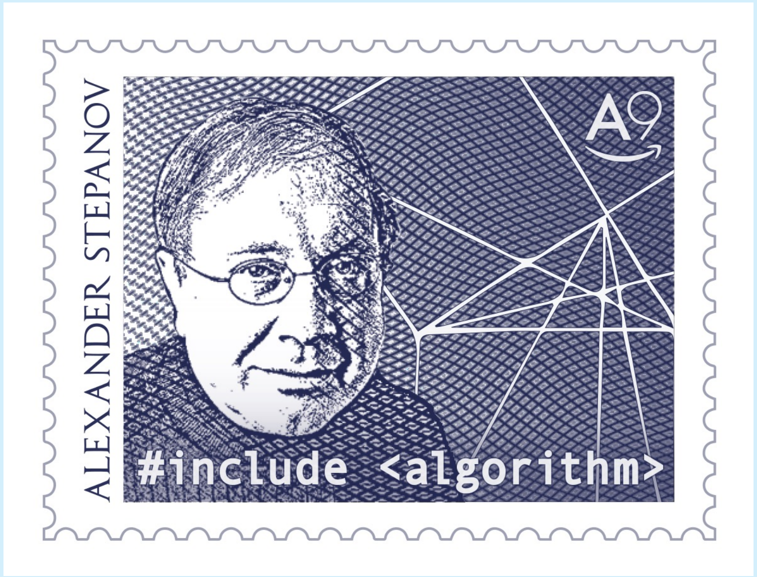
The STL

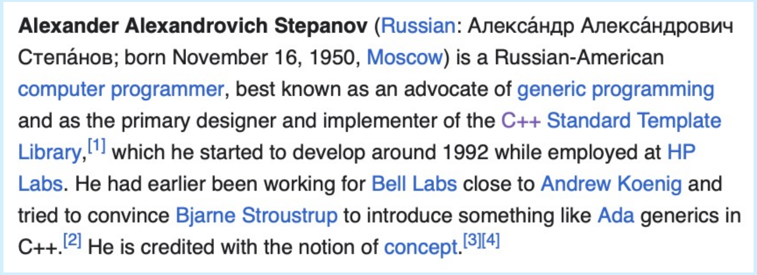
- Part of the ISO C++ Standard Library
- Mostly non-numerical
  - Only 4 standard algorithms specifically do computation
    - Accumulate, inner_product, partial_sum, adjacent_difference
  - Handles textual data as well as numeric data
    - E.g. string
  - Deals with organization of code and data
    - Built-in types, user-defined types, and data structures
- Optimizing disk access was among its original uses
  - Performance was always a key concern

Imperial College London

The STL

- Designed by Alex Stepanov
- General aim: The most general, most efficient, most flexible representation of concepts (ideas, algorithms)
  - Represent separate concepts separately in code
  - Combine concepts freely wherever meaningful
- General aim to make programming "like math"
  - or even "Good programming *is* math"
  - works for integers, for floating-point numbers, for polynomials, for …





Alexander Alexandrovich Stepanov (Russian: Алекса́ндр Алекса́ндрович Степа́нов; born November 16, 1950, Moscow) is a Russian-American computer programmer, best known as an advocate of generic programming and as the primary designer and implementer of the C++ Standard Template Library,[1] which he started to develop around 1992 while employed at HP Labs. He had earlier been working for Bell Labs close to Andrew Koenig and tried to convince Bjarne Stroustrup to introduce something like Ada generics in C++.[2] He is credited with the notion of concept.[3][4]

The Standard Template Library, or STL

STL is a C++ library of container classes, algorithms, and iterators;

it provides many of the basic algorithms and data structures of computer science.

The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template.

You should make sure that you understand how templates work in C++ before you use the STL.

-SGI Introduction to the Standard Template Library

STL Library

- Code reuse, abstraction and optimization of containers and algorithms
- Efficient (fast and less resources)
- Accurate (less bugs)
- Readable code
- Standardised
- A good example of what a library should be

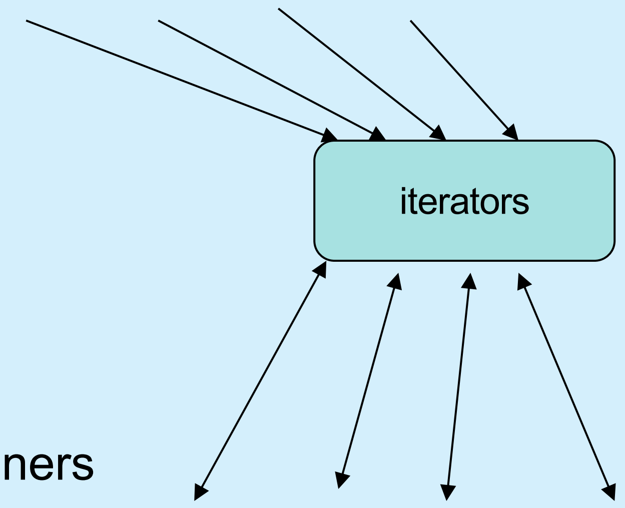
Types of Containers

- Sequence containers (arrays and linked lists)
  - Vector, deque, list, forward list
- Associative containers (binary tree)
  - Set, multiset
  - Map, multimap
- Unordered containers (hash table)
  - Unordered set/multiset
  - Unordered map/multimap

Basic model

- Algorithms

  sort, find, search, copy, …



- Containers

  vector, list, map, unordered_map, …

- Separation of concerns
  - Algorithms manipulate data, but don’t know about containers
  - Containers store data, but don’t know about algorithms
  - Algorithms and containers interact through iterators
    - Each container has its own iterator types

Imperial College London

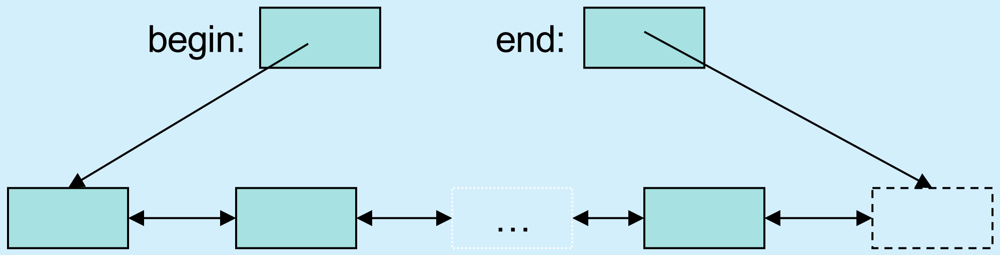
The STL

- An ISO C++ standard framework of about 10 containers and about 60 algorithms connected by iterators
  - Other organizations provide more containers and algorithms in the style of the STL
    - Boost.org, Microsoft, …
- Probably the currently best known and most widely used example of generic programming

Imperial College London

Basic model

- A pair of iterators defines a sequence
  - The beginning (points to the first element – if any)
  - The end (points to the one-beyond-the-last element)



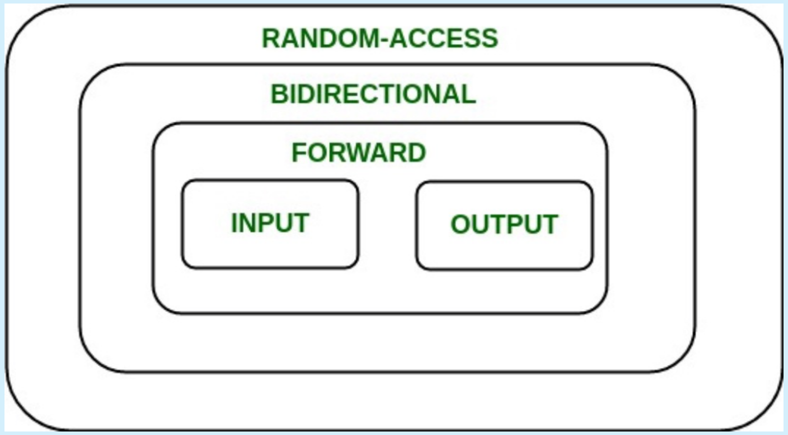
- An iterator is a type that supports the "iterator operations"
  - ++ Go to next element
  - * Get value
  - == Does this iterator point to the same element as that iterator?
- Some iterators support more operations (e.g. --, +, and [ ])

Imperial College London

Iterators

"An iterator is an object that can iterate over elements in a C++ Standard Library container and provide access to individual elements. The C++ Standard Library containers all provide iterators so that algorithms can access their elements in a standard way without having to be concerned with the type of container the elements are stored in."

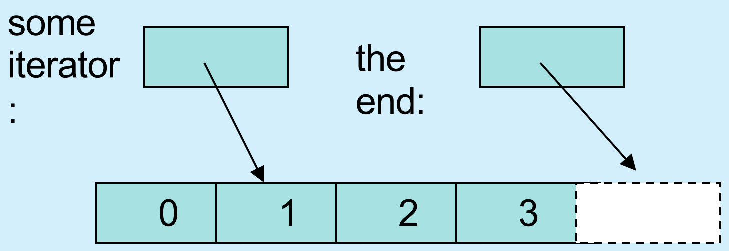
https://docs.microsoft.com/en-us/cpp/standard-library/iterators?view=msvc-170

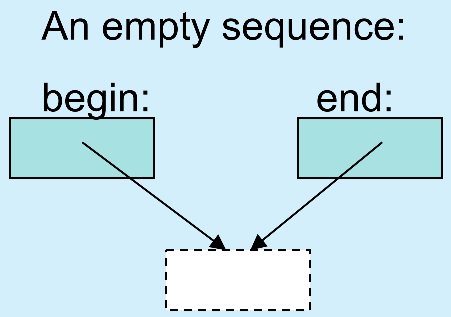


Imperial College London

How iterators work

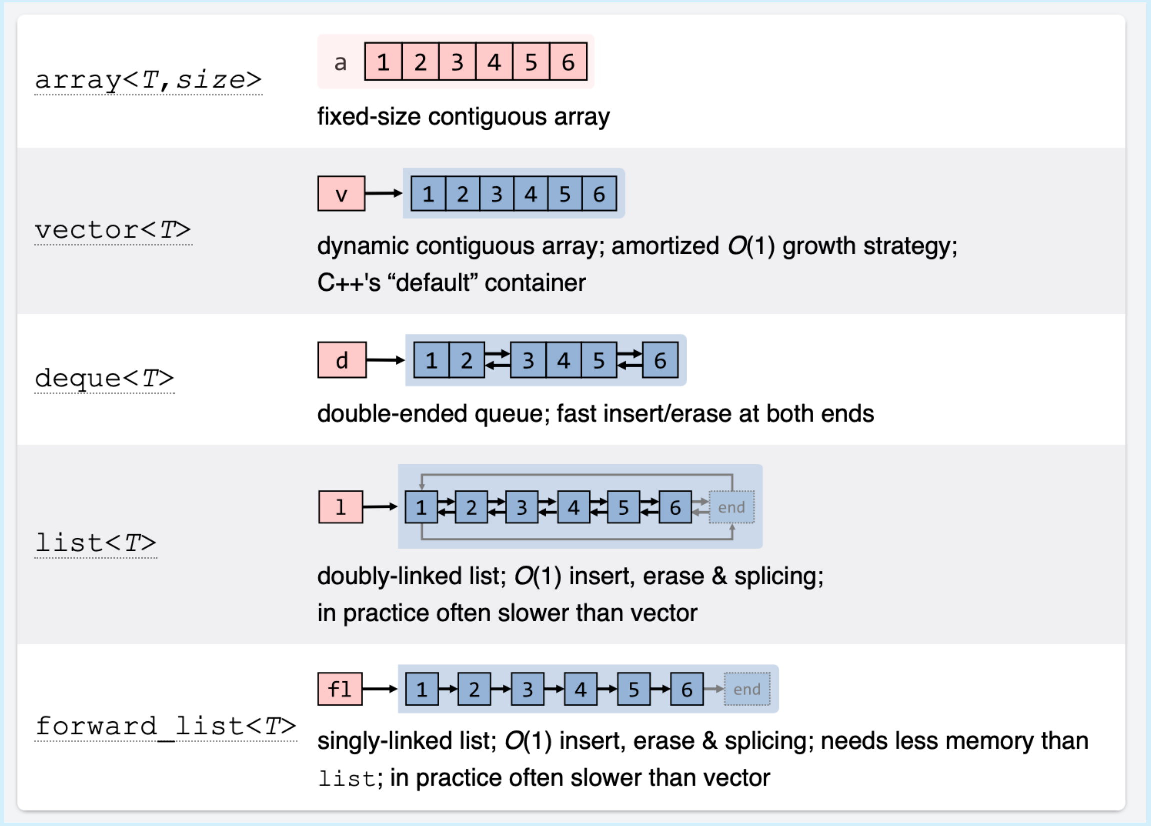
- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is “one past the last element”
  - ***not*** “the last element”
  - That’s necessary to elegantly represent an empty sequence
  - One-past-the-last-element isn’t an element
    - You can compare an iterator pointing to it
    - You can’t dereference it (read its value)
- Returning the end of the sequence is the standard idiom for “not found” or “unsuccessful”





Imperial College London

Sequence Containers (hold sequences in difference ways)



`array<T,size>`	fixed-size contiguous array
`vector<T>`	dynamic contiguous array; amortized *O*(1) growth strategy; C++'s "default" container
`deque<T>`	double-ended queue; fast insert/erase at both ends
`list<T>`	doubly-linked list; *O*(1) insert, erase & splicing; in practice often slower than vector
`forward_list<T>`	singly-linked list; *O*(1) insert, erase & splicing; needs less memory than `list`; in practice often slower than vector

Imperial College
London

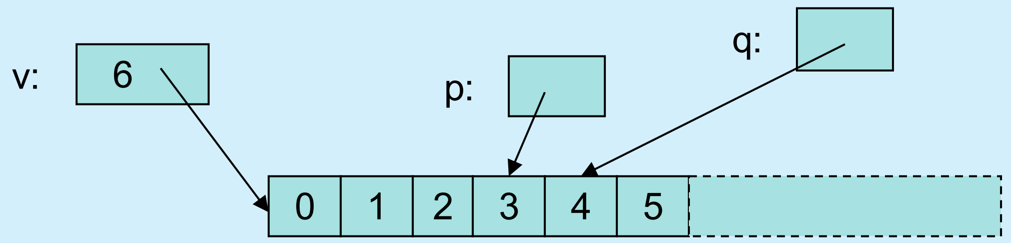
vector

```
template<class T> class vector {
    T* elements;
    // …
    using value_type = T;
    using iterator = ???;     // the type of an iterator is implementation defined
                // and it (usefully) varies (e.g. range checked iterators)
                // a vector iterator could be a pointer to an element
    using const_iterator = ???;

    iterator begin();         // points to first element
    const_iterator begin() const;
    iterator end();      // points to one beyond the last element
    const_iterator end() const;

    iterator erase(iterator p);         // remove element pointed to by p
    iterator insert(iterator p, const T& v);// insert a new element v before p
};
```

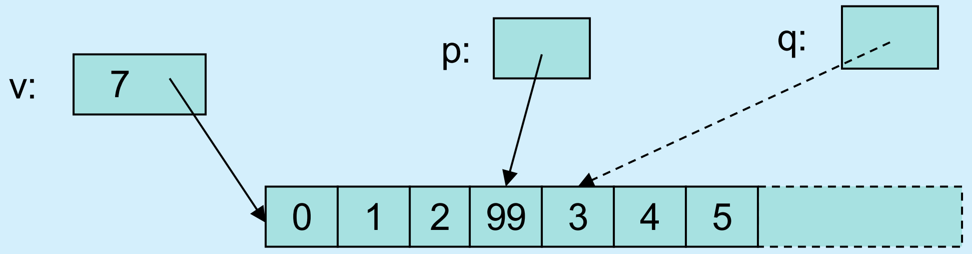
Imperial College London

`insert()` into vector

```
vector<int>::iterator p = v.begin(); ++p; ++p; ++p;
vector<int>::iterator q = p; ++q;
```



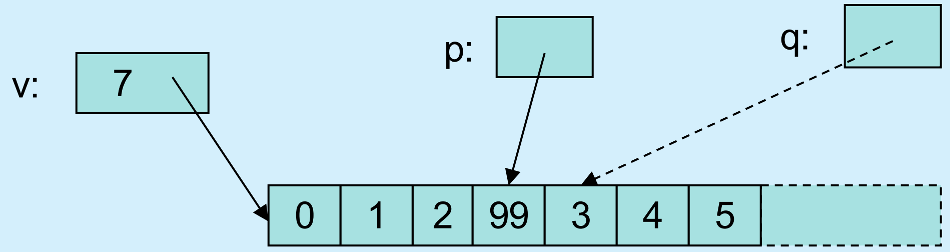
`p=v.insert(p,99);` // leaves p pointing at the inserted element



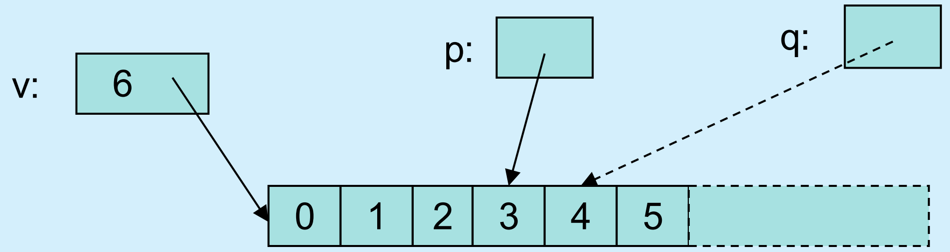
- Note: q is invalid after the insert()
- Note: Some elements moved; all elements could have moved

Imperial College London

erase() from vector



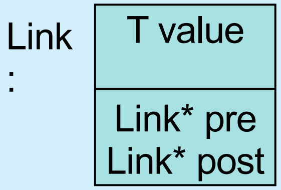
```
p = v.erase(p);     // leaves p pointing at the element after the erased one
```



- vector elements move when you insert() or erase()
- Iterators into a vector are invalidated by insert() and erase()

Imperial College
London

list

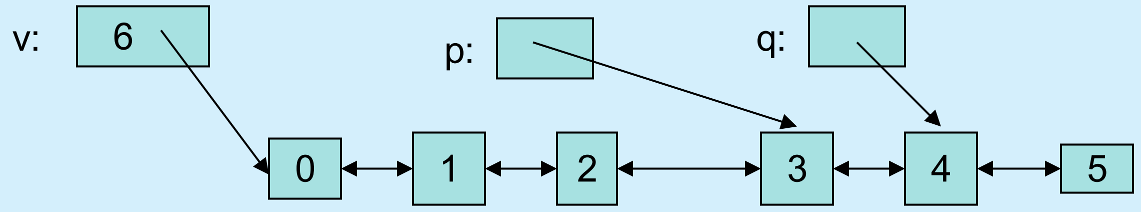


```
template<class T> class list {
    Link* elements;
    // …
    using value_type = T;
    using iterator = ???;     // the type of an iterator is implementation defined
                // and it (usefully) varies (e.g. range checked iterators)
                // a list iterator could be a pointer to a link node
    using const_iterator = ???;

    iterator begin();         // points to first element
    const_iterator begin() const;
    iterator end();      // points one beyond the last element
    const_iterator end() const;

    iterator erase(iterator p);         // remove element pointed to by p
    iterator insert(iterator p, const T& v);// insert a new element v before p
};
```

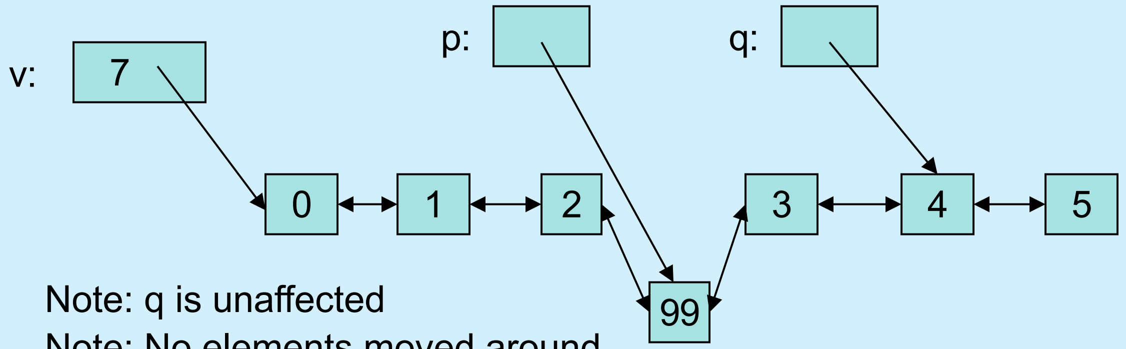
Imperial College London

What happens when you insert() into a list?

```
list<int>::iterator p = v.begin(); ++p; ++p; ++p;
list<int>::iterator q = p; ++q;
```



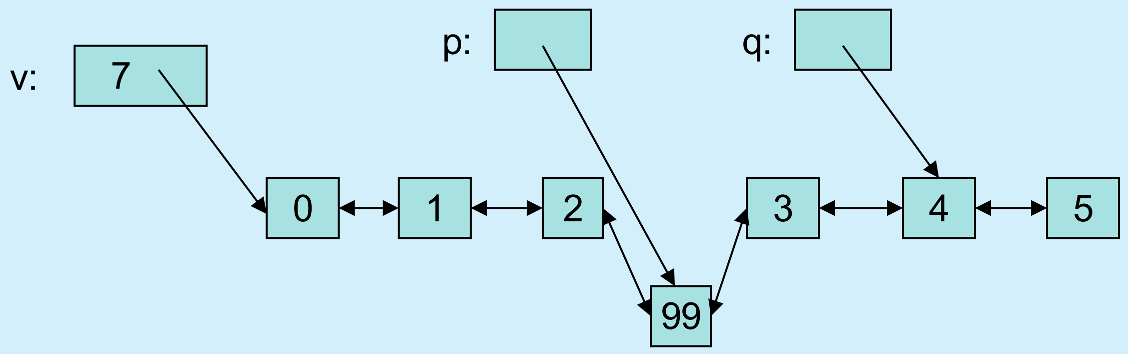
```
v = v.insert(p,99); // leaves p pointing at the inserted element
```



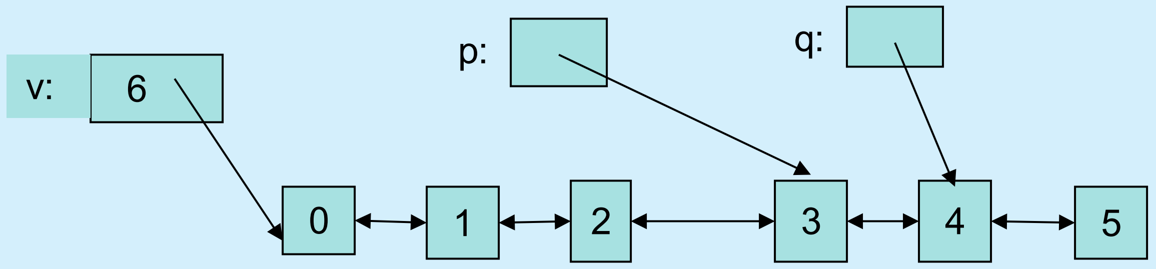
- Note: q is unaffected
- Note: No elements moved around

Imperial College
London

What happens when you erase() from a list?



```
p = v.erase(p);     // leaves p pointing at the element after the erased one
```



- Note: list elements do not move when you insert() or erase()

Imperial College London

Methods to traverse a vector

```
for(int i = 0; i<v.size(); ++i)           // why int?
     …  // do something with v[i]

for(vector<T>::size_type i = 0; i<v.size(); ++i)   // longer but always correct
     …  // do something with v[i]

for(vector<T>::iterator p = v.begin(); p!=v.end(); ++p)
    …// do something with *p
```

- Know both ways (iterator and subscript)
  - The subscript style is used in essentially every language
  - The iterator style is used in C (pointers only) and C++
  - The iterator style is used for standard library algorithms
  - The subscript style doesn’t work for lists (in C++ and in most languages)
- Use either way for vectors
  - There are no fundamental advantages of one style over the other
  - But the iterator style works for all sequences
  - Prefer **`size_type`** over plain **`int`**
    - pedantic, but quiets compiler and prevents rare errors

Imperial College
London

Methods to traverse a vector

```
for(vector<T>::iterator p = v.begin(); p!=v.end(); ++p)
    …// do something with *p

for(vector<T>::value_type x : v)
    …// do something with x

for(auto& x : v)
    …// do something with x
```

- "Range **for**"
  - Use for the simplest loops
    - Every element from **begin()** to **end()**
  - Over one sequence
  - When you don't need to look at more than one element at a time
  - When you don't need to know the position of an element

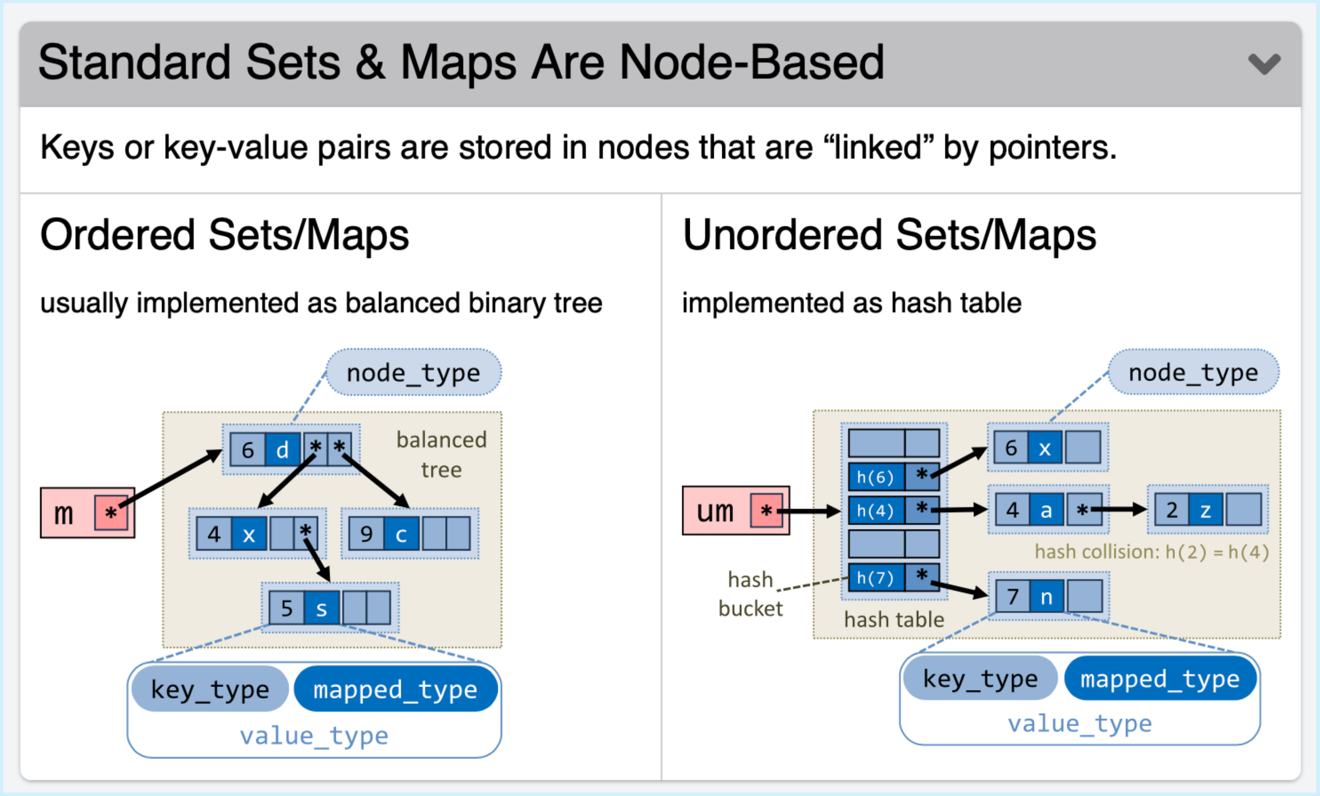
Imperial College London

`vector` vs. `list`

- By default, use a **`vector`**
  - You need a reason not to
  - You can "grow" a vector (e.g., using **`push_back()`**)
  - You can **`insert()`** and **`erase()`** in a vector
  - Vector elements are compactly stored and contiguous
  - For small vectors of small elements all operations are fast
    - compared to lists
- If you don't want elements to move, use a **`list`**
  - You can "grow" a list (e.g., using **`push_back()`** and **`push_front()`**)
  - You can **`insert()`** and **`erase()`** in a list
  - List elements are separately allocated
- Note that there are more containers, e.g.,
  - `map`
  - `unordered_map`

Imperial College
London

Associative Containers



Standard Sets & Maps Are Node-Based

Keys or key-value pairs are stored in nodes that are “linked” by pointers.

Ordered Sets/Maps

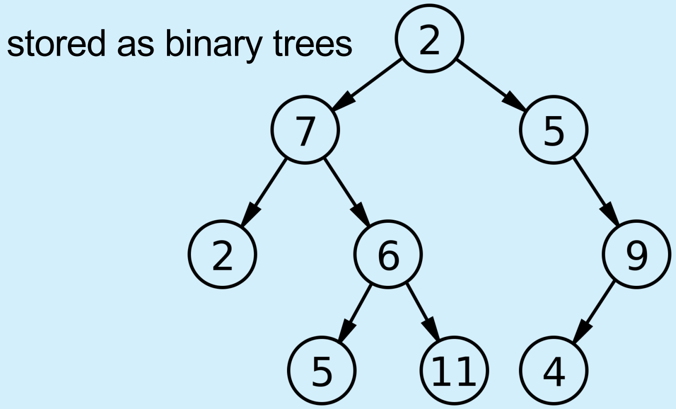
usually implemented as balanced binary tree

Unordered Sets/Maps

implemented as hash table

https://hackingcpp.com/cpp/std/containers.html

Associative Containers



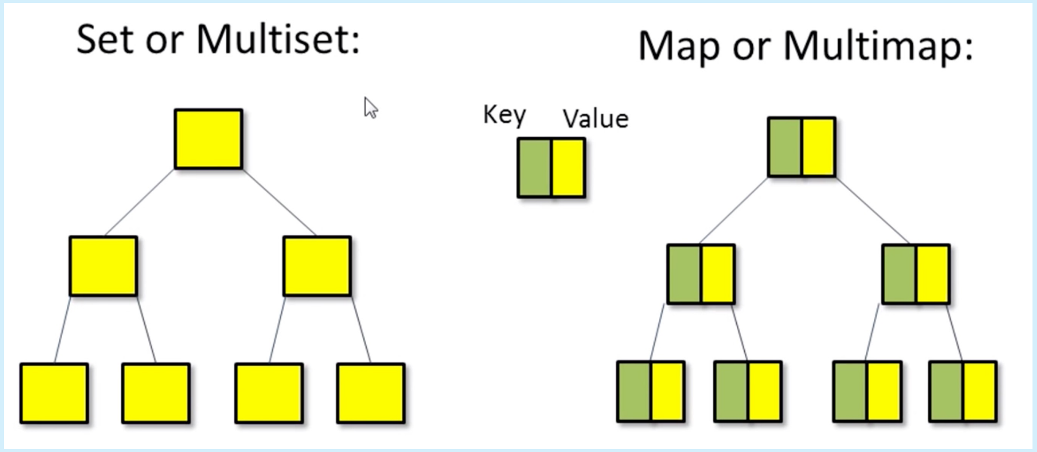
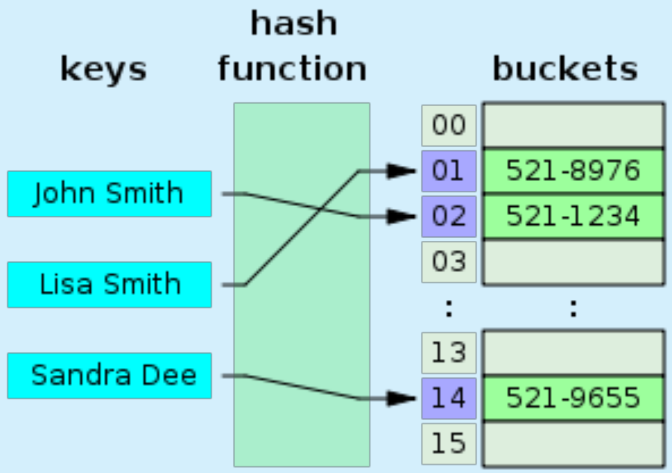


Image taken from Bo Qian's YouTube "Introduction of STL 3: Associative Containers"

The underlying data structure of an ordered associate container is a balanced search tree:
• logarithmic access time
• requires order comparisons of keys
• iteration in key order
• Iterators, pointers and references stay valid until the pointed to element is removed.

Unordered Containers

Stored using hash tables:



search, insert and delete in constant time

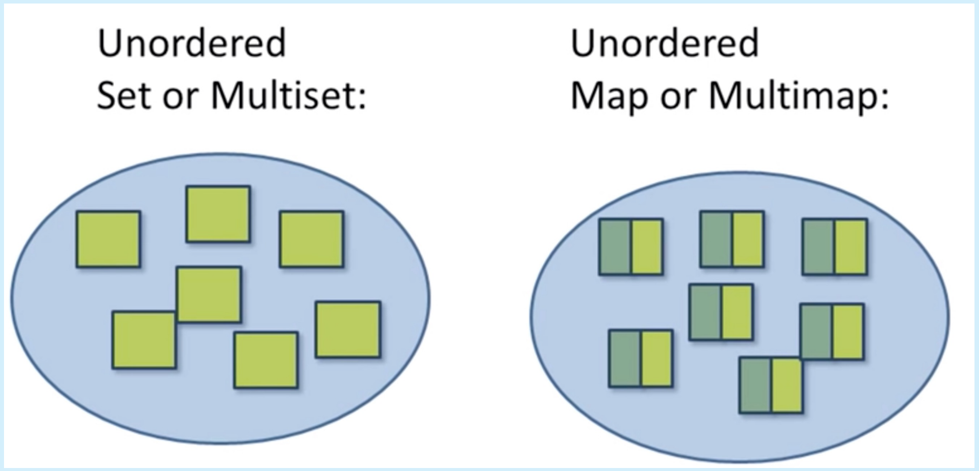
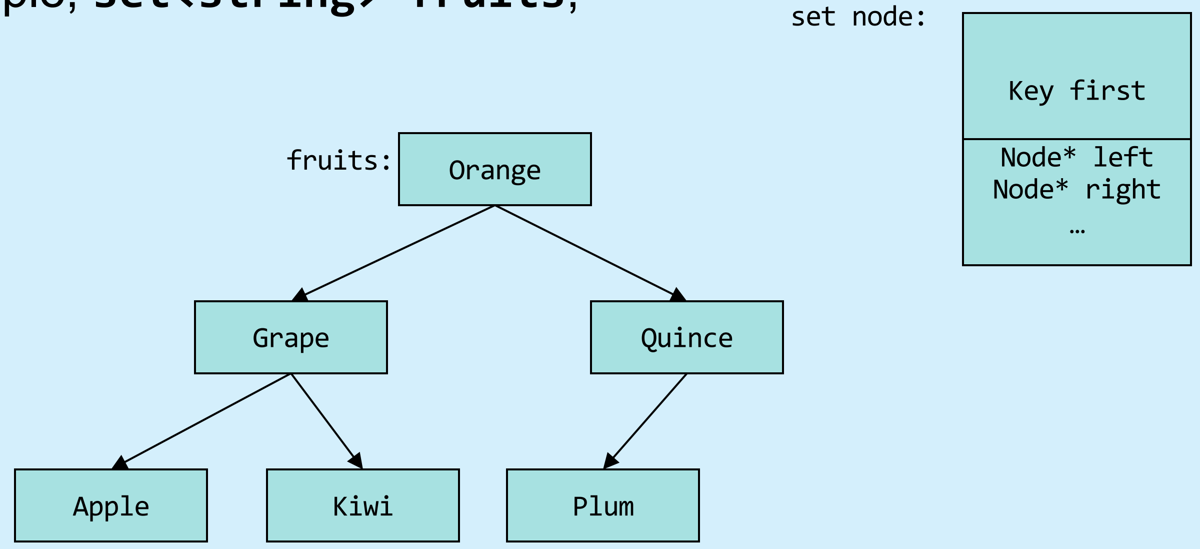


Image taken from Bo Qian's YouTube "Introduction of STL 4: Unordered Containers"

Imperial College London

`std::set`

- A **`set`** is really an ordered balanced binary tree
  - By default ordered by <
  - For example, **`set<string> fruits`**;



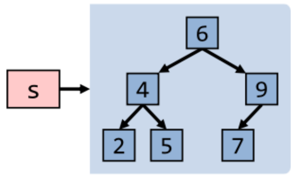
Imperial College
London

Associative Containers: Sets

Sets

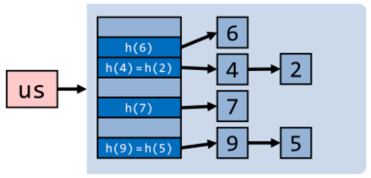
Ordered Sets

`#include <set>`



Hash Sets

`#include <unordered_set>`



set<*Key*> / unordered_set<*Key*>

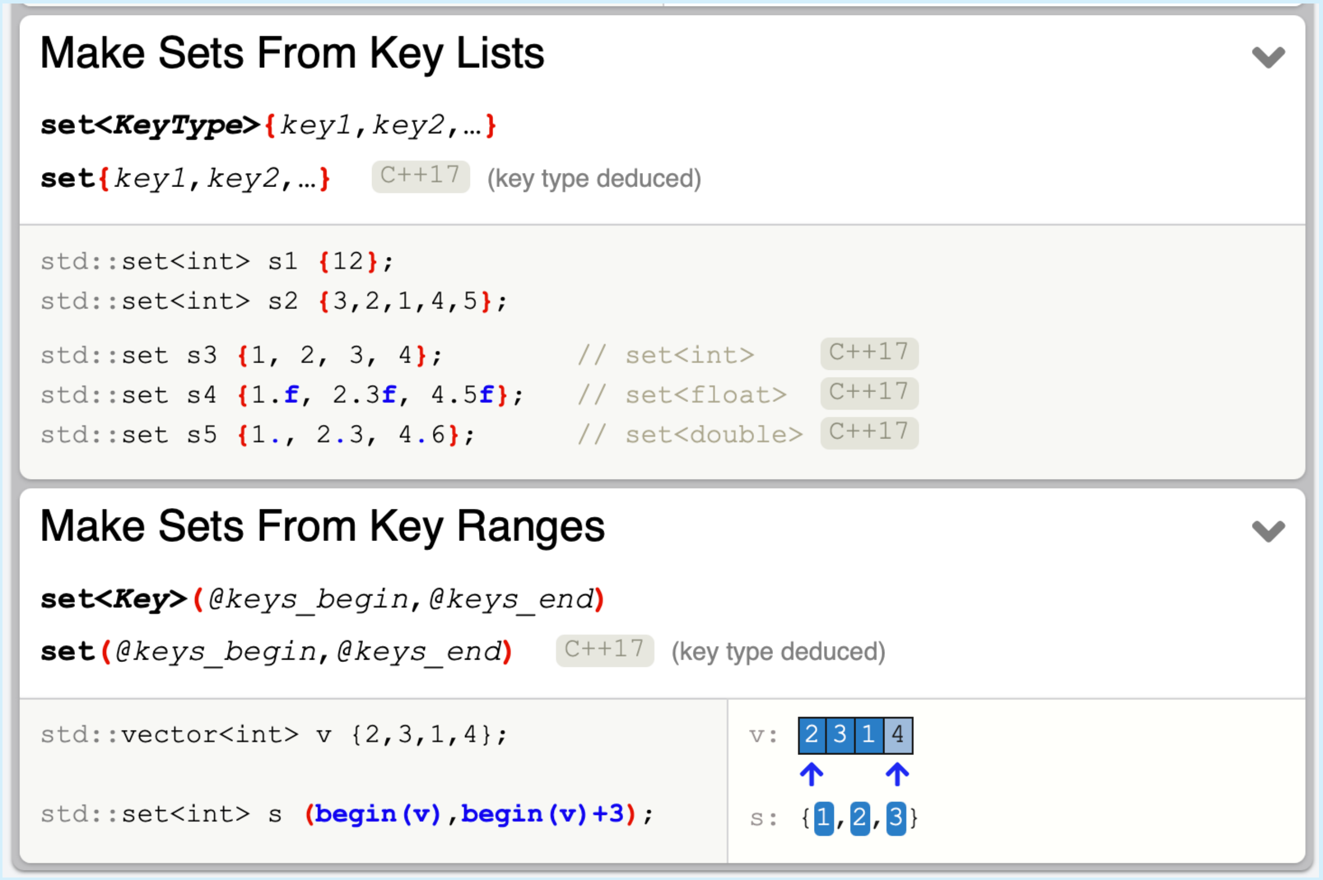
unique orderable / hashable keys

multiset<*Key*> / unordered_*multiset*<*Key*>

multiple equivalent keys possible

Imperial College
London

Example: Sets



Make Sets From Key Lists

set<*KeyType*>{*key1*, *key2*, …**}**

set{*key1*, *key2*, …**}** C++17 (key type deduced)

```
std::set<int> s1 {12};
std::set<int> s2 {3,2,1,4,5};

std::set s3 {1, 2, 3, 4};          // set<int>     C++17
std::set s4 {1.f, 2.3f, 4.5f};     // set<float>   C++17
std::set s5 {1., 2.3, 4.6};        // set<double>  C++17
```

Make Sets From Key Ranges

set<*Key*>(*@keys_begin*, *@keys_end***)**

set(*@keys_begin*, *@keys_end***)** C++17 (key type deduced)

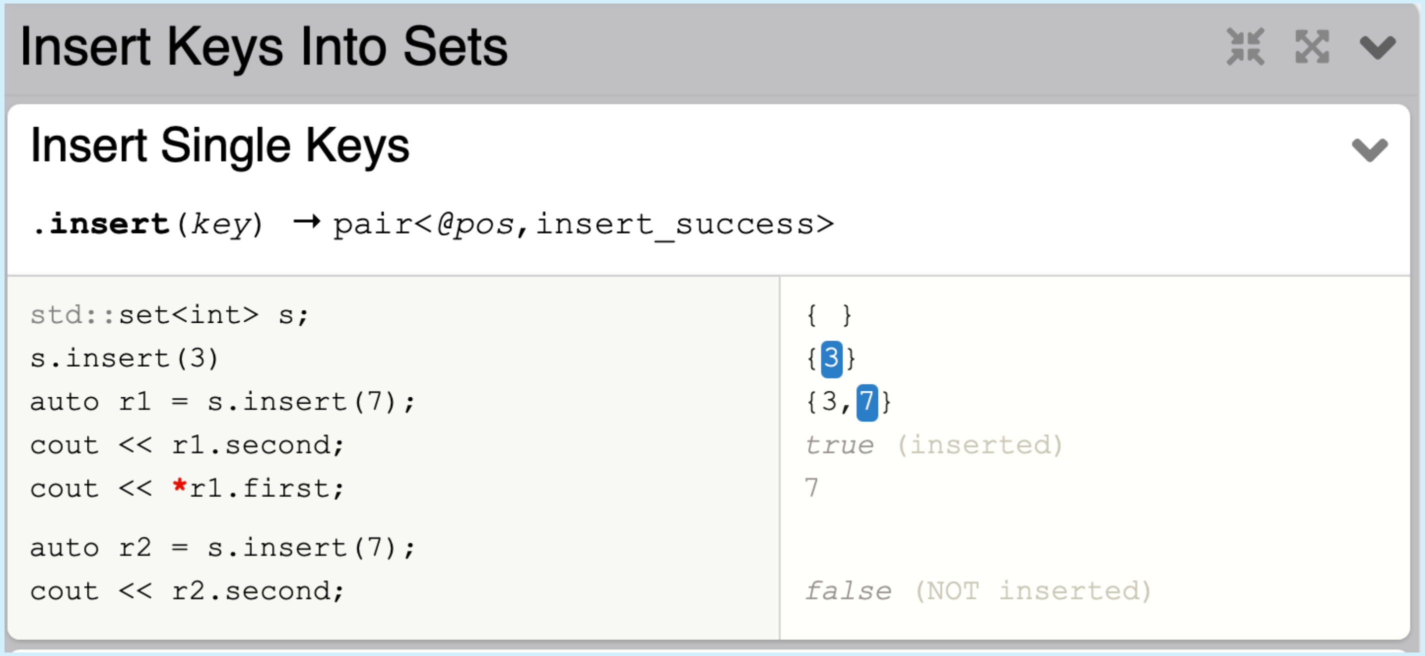
```
std::vector<int> v {2,3,1,4};

std::set<int> s (begin(v),begin(v)+3);
```

```
v: 2 3 1 4
   ↑     ↑
s: {1,2,3}
```

https://hackingcpp.com/cpp/std/containers.html

Imperial College
London

Insert elements: Set


Insert Keys Into Sets

Insert Single Keys

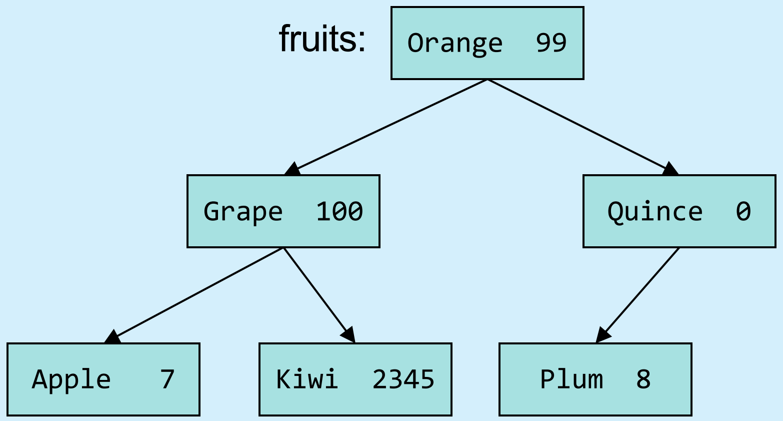
`.insert(key) → pair<@pos,insert_success>`

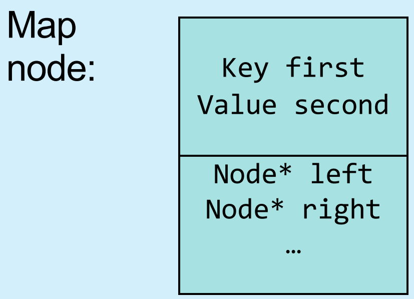
```
std::set<int> s;                    { }
s.insert(3)                         {3}
auto r1 = s.insert(7);              {3,7}
cout << r1.second;                  true (inserted)
cout << *r1.first;                  7

auto r2 = s.insert(7);
cout << r2.second;                  false (NOT inserted)
```

Imperial College London

Map

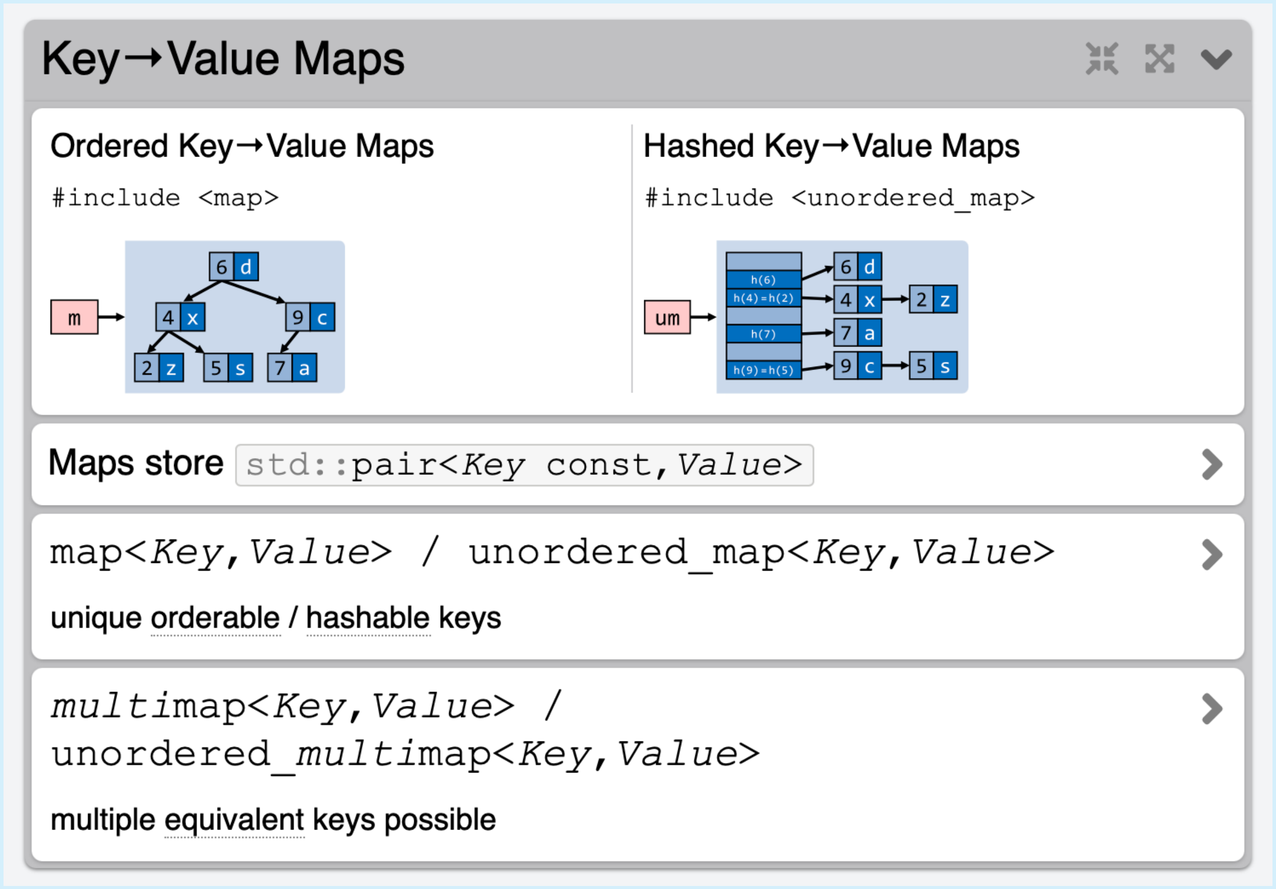
- After **`vector`**, **`map`** is the most useful standard library container
  - Maps (and/or hash tables) are the backbone of scripting languages
- A **`map`** is really an ordered balanced binary tree
  - By default ordered by **`<`** (less than)
  - For example, **`map<string,int> fruits`**;





Imperial College
London

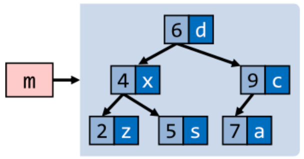
Associative Containers: Maps



Key→Value Maps

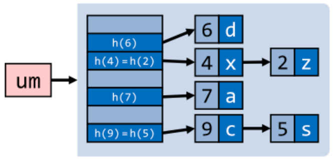
Ordered Key→Value Maps

```
#include <map>
```



Hashed Key→Value Maps

```
#include <unordered_map>
```



Maps store `std::pair<Key const,Value>`

`map<Key,Value> / unordered_map<Key,Value>`

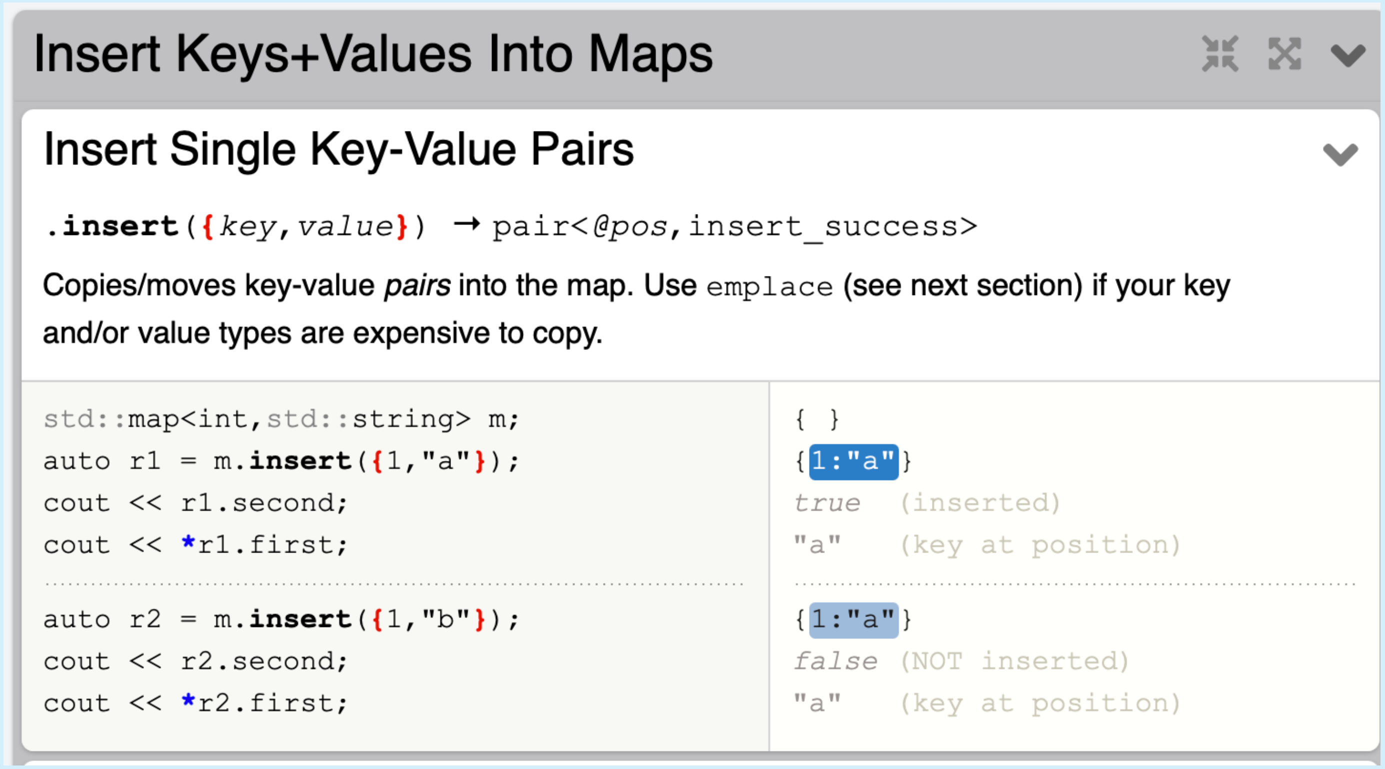
unique orderable / hashable keys

`multimap<Key,Value> / unordered_multimap<Key,Value>`

multiple equivalent keys possible

Imperial College London

Insert elements: Map



Insert Keys+Values Into Maps

Insert Single Key-Value Pairs

`.insert({key,value}) → pair<@pos,insert_success>`

Copies/moves key-value *pairs* into the map. Use `emplace` (see next section) if your key and/or value types are expensive to copy.

```
std::map<int,std::string> m;          { }
auto r1 = m.insert({1,"a"});          {1:"a"}
cout << r1.second;                    true   (inserted)
cout << *r1.first;                    "a"    (key at position)

auto r2 = m.insert({1,"b"});          {1:"a"}
cout << r2.second;                    false  (NOT inserted)
cout << *r2.first;                    "a"    (key at position)
```

Imperial College
London

Map (an associative array)

- For a **vector**, you subscript using an integer
- For a **map**, you can define the subscript to be (just about) any type

Key type

Value type

```
int main()
{
   map<string,int> words;      // keep (word,frequency) pairs
   for (string s; cin>>s; )
         ++words[s];      // note: words is subscripted by a string
                     // words[s] returns an int&
                     // the int values are initialized to 0
   for (const auto&  p : words)
         cout << p.first << ": " << p.second << "\n";
}
```

Imperial College London

Map

Some implementation defined type

```
// note the similarity to vector and  list

template<class Key, class Value> class map {
    // …
    using value_type = pair<Key,Value>;      // a map deals in (Key,Value) pairs

    using iterator = ???;                    // probably a pointer to a tree node
    using const_iterator = ???;

    iterator begin();                        // points to first element
    iterator end();                          // points to one beyond the last element

    Value& operator[ ](const Key&);          // get Value for Key; creates pair if
                                             // necessary, using Value( )
    iterator find(const Key& k);             // is there an entry for k?

    void erase(iterator p);                  // remove element pointed to by p
    pair<iterator, bool> insert(const value_type&);       // insert new (Key,Value) pair
    // …                                     // the bool is false if insert failed
};
```

Imperial College
London

Map example (build some maps)

```
map<string,double> dow;  // Dow-Jones industrial index (symbol, price)

dow["MMM"] = 81.86;
dow["AA"] = 34.69;
dow["MO"] = 54.45;
// …
map<string,double> dow_weight;                      // dow (symbol, weight)
dow_weight.insert(make_pair("MMM", 5.8549));        // just to show that a Map
                                                    // really does hold pairs
dow_weight.insert(make_pair("AA",2.4808));
dow_weight.insert(make_pair("MO",3.8940));            // and to show that notation matters
// …
map<string,string> dow_name; // dow (symbol, name)
dow_name["MMM"] = "3M Co.";
dow_name["AA"] = "Alcoa Inc.";
dow_name["MO"] = "Altria Group Inc.";
// …
```

Imperial College
London

Map example (some uses)

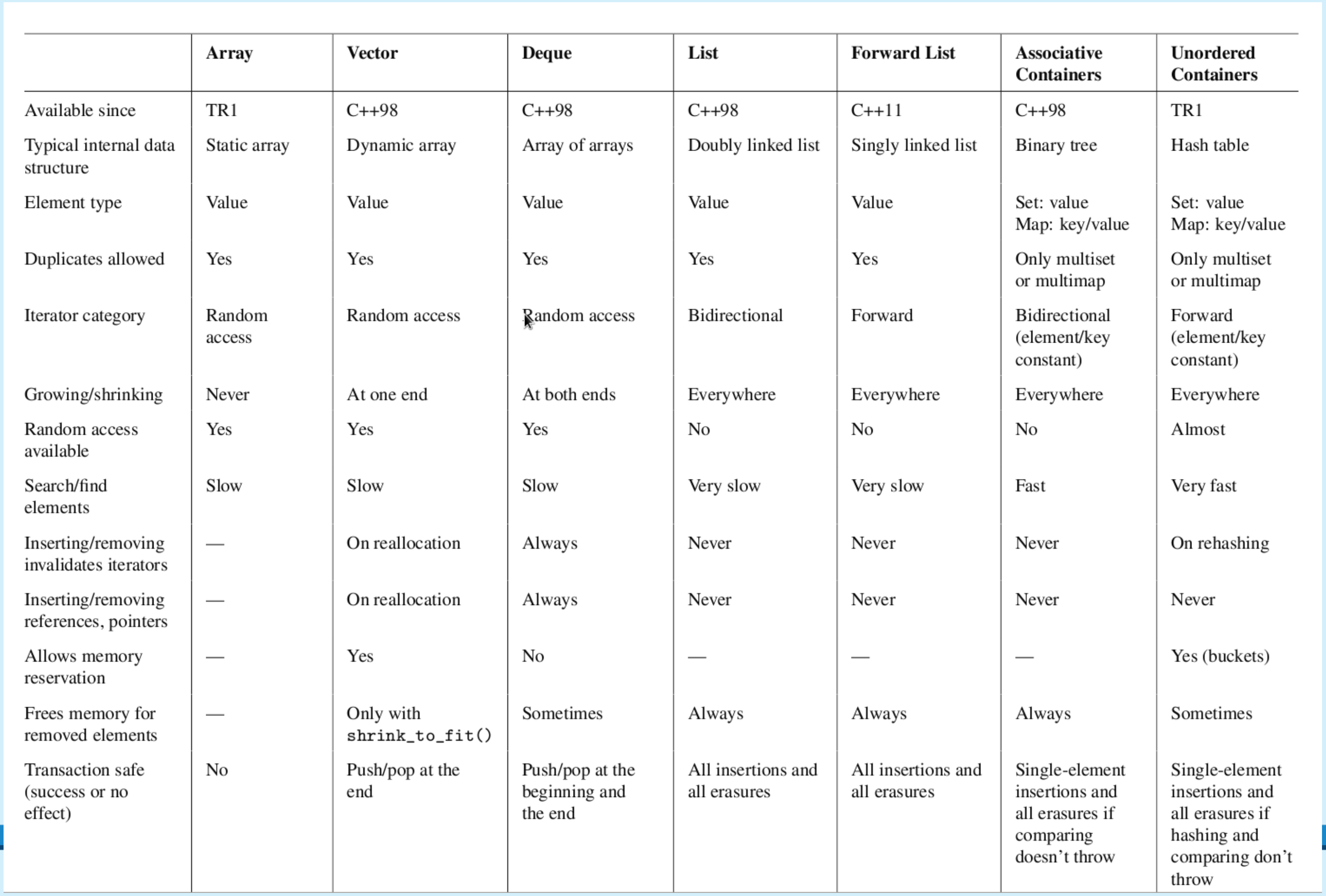
```
double alcoa_price = dow["AA"];                  // read values from a map
double boeing_price = dow["BO"];


if (dow.find("INTC") != dow.end())               // look in a map for an entry
    cout << "Intel is in the Dow\n";


// iterate through a map:

for (const auto& p : dow) {
    const string& symbol = p.first;              // the "ticker" symbol
    cout << symbol  << '\t' << p.second << '\t' << dow_name[symbol] << '\n';
}
```

Imperial College
London

Map example (calculate the DJ index)

```
double value_product(
    const pair<string,double>& a,
    const pair<string,double>& b)       // extract values and multiply
{
    return a.second * b.second;
}

double dj_index =
    inner_product(dow.begin(), dow.end(),         // all companies in index
                  dow_weight.begin(),             // their weights
                  0.0,                            // initial value
                  plus<double>(),                 // add (as usual)
                  value_product                   // extract values and weights
             );                                   // and multiply; then sum
```

Containers have different benefits and costs



	Array	Vector	Deque	List	Forward List	Associative Containers	Unordered Containers
Available since	TR1	C++98	C++98	C++98	C++11	C++98	TR1
Typical internal data structure	Static array	Dynamic array	Array of arrays	Doubly linked list	Singly linked list	Binary tree	Hash table
Element type	Value	Value	Value	Value	Value	Set: value Map: key/value	Set: value Map: key/value
Duplicates allowed	Yes	Yes	Yes	Yes	Yes	Only multiset or multimap	Only multiset or multimap
Iterator category	Random access	Random access	Random access	Bidirectional	Forward	Bidirectional (element/key constant)	Forward (element/key constant)
Growing/shrinking	Never	At one end	At both ends	Everywhere	Everywhere	Everywhere	Everywhere
Random access available	Yes	Yes	Yes	No	No	No	Almost
Search/find elements	Slow	Slow	Slow	Very slow	Very slow	Fast	Very fast
Inserting/removing invalidates iterators	—	On reallocation	Always	Never	Never	Never	On rehashing
Inserting/removing references, pointers	—	On reallocation	Always	Never	Never	Never	Never
Allows memory reservation	—	Yes	No	—	—	—	Yes (buckets)
Frees memory for removed elements	—	Only with `shrink_to_fit()`	Sometimes	Always	Always	Always	Sometimes
Transaction safe (success or no effect)	No	Push/pop at the end	Push/pop at the beginning and the end	All insertions and all erasures	All insertions and all erasures	Single-element insertions and all erasures if comparing doesn't throw	Single-element insertions and all erasures if hashing and comparing don't throw

EXERCISE

1. Part I. Create five containers: a vector, a set, an unordered_set, a map and a multimap. Add the same 100 random entries to the containers (for the sequences containers add the random value r, for the map and multimap add the pair (r, fact(r))).

2. Part II. (After the next segment)

Imperial College London

Advanced Programming

Measuring Time

Adriana Paluszny

Measuring time

- Code needs to be profiled sometimes
- Understand what really takes time
- How much improvement does one strategy have over another

`std::chrono`

- Provides a variety of clocks in the `<chrono>` header
- Useful for when you want to program something that depends on time or for timing your code
- Provides three clocks, all in the `std::`chrono namespace, with each providing a different guarantee

`std::chrono::system_clock`

- `std::chrono::system_clock` is the system wide real-time clock
- It is like your watch, or the clock on the wall
- Gives elapsed time since an implementation specific start date
  - Most use January 1, 1970 at midnight (Unix Time)

`std::chrono::steady_clock`

- `std::chrono::steady_clock` guarantees that its value will never decrease
- Might seem absurd, but measuring time is complicated -- might have to deal with leap seconds and/or inaccurate clocks
- So yes, this kind of stuff happens

Imperial College London

`std::chrono::high_resolution_clock`

- `std::chrono::high_resolution_clock` has the shortest tick period available tick is the smallest atomic change that the clock can measure (the granularity of the clock)
- Beware of situations where tick is, say, millisecond, but clock is only updated every half second!
- Mostly a historical issue now

Imperial College London

now()

- Each clock supports the static member function `now()`, which returns a time point corresponding to the current value of the clock
- time point represents a moment in time
- `chrono` encodes time points using `std::chrono::time_point` type

Imperial College
London

An example

```cpp
#include <iostream>
#include <chrono>
#include <unistd.h>

using namespace std;

// Main function to measure elapsed time of a C++ program
// using Chrono library
int main()
{
    auto start = chrono::steady_clock::now();

    // do some stuff here
    sleep(3);

    auto end = chrono::steady_clock::now();

    cout << "Elapsed time in nanoseconds: "
        << chrono::duration_cast<chrono::nanoseconds>(end - start).count()
        << " ns" << endl;

    cout << "Elapsed time in microseconds: "
        << chrono::duration_cast<chrono::microseconds>(end - start).count()
        << " µs" << endl;

    cout << "Elapsed time in milliseconds: "
        << chrono::duration_cast<chrono::milliseconds>(end - start).count()
        << " ms" << endl;

    cout << "Elapsed time in seconds: "
        << chrono::duration_cast<chrono::seconds>(end - start).count()
        << " sec";

    return 0;
}
```

Imperial College London

Also consider…

- Take the time to watch these amazing **videos**

https://learn.microsoft.com/en-us/shows/c9-lectures-stephan-t-lavavej-core-c-/

https://learn.microsoft.com/en-us/shows/c9-lectures-stephan-t-lavavej-standard-template-library-stl-/

Imperial College London

EXERCISE

1. Part I. Create five containers: a vector, a set, an unordered_set, a map and a multimap. Add the same 100 random entries to the containers (for the sequences containers add the random value r, for the map and multimap add the tuple (r, fact(r))).

2. Part II. Extend your containers (without recreating them) so that they have a total of 10,000 entries each, with tuples (r,r) for the map and multimap. Do this by adding entries to your containers. Measure how long this takes. Then write another loop that erases all of the newly inserted items for each container type. Measure how long this takes.