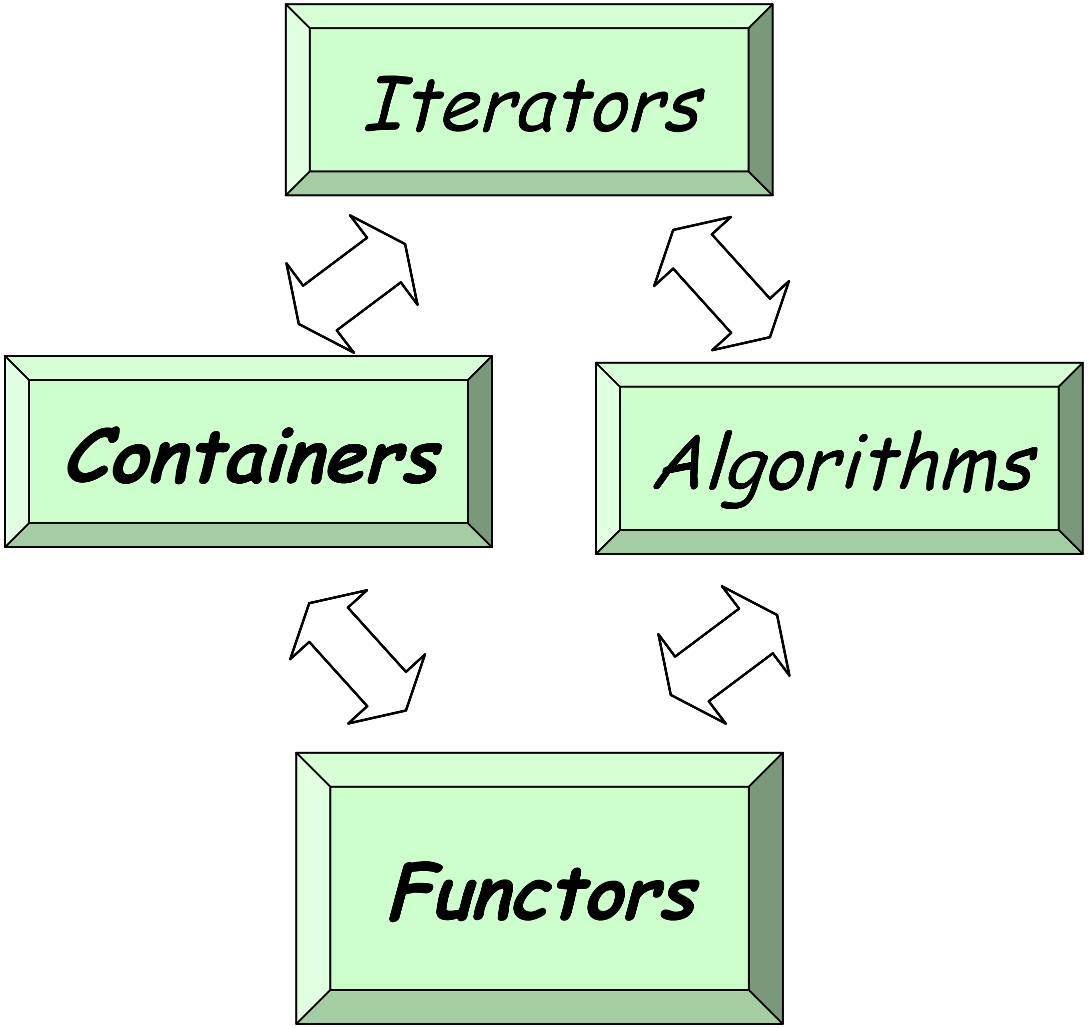
The Standard C++ Library

Version 1: Dr. Ofir Pele
Version 2: Dr. Erel Segal-Halevi

Main Components



Containers

Fixed size:

- pair, tuple, array

Sequence containers:

- forward_list, list, vector, deque, [basic_string]

Associative containers:

- set, multiset, map, multimap
- *unordered*_set, multiset, map, multimap

Container adaptors:

- stack, queue, priority_queue

Pairs, Tuples

(folder 0)

- Can hold a fixed number of elements of various types.
- Particularly useful in a **return** statement, to let your function return several values.
- Shortest (most automated) version:

```
auto f () {
 return tuple(5,'a',"hello");
}
// in main:
auto [ii,cc,ss] = f();
```

- Longer versions in folder 0.

Sequential Containers

Objects are ordered by the user

	insert first	insert middle	insert last	random access	iterate forward	iterate back	*find*	storage
forward_ list	fast	fast	fast	slow	fast	slow	*slow*	heap
list	fast	fast	fast	slow	fast	fast	*slow*	heap
deque	fast	~slow	fast	~fast	fast	fast	*slow*	heap
vector	slow	slow	~fast	fast	fast	fast	*slow*	heap
basic_ string	slow	slow	~fast	fast	fast	fast	*slow*	heap
array	-	-	-	fast	fast	fast	*slow*	stack

Associative Containers

Ordered: Objects are ordered by "<".
Insertion, deletion, find: O(log n).

set: Unique keys.

map: Associate unique keys to values.

multiset: Allows multiple keys.

multimap: Associate keys to multiple values.

Unordered: Insertion, deletion, find: O(1).

unordered_set: Unique keys.

unordered_map: Associate keys to values.

unordered_multiset: Allows multiple keys.

unordered_multimap: Associate keys to multiple values.

Container Adaptors

Take any sequential container;

return a container with a given interface:

- **stack**: last in – first out;
- **queue**: first in – first out;
- **priority_queue**: best in – first out.

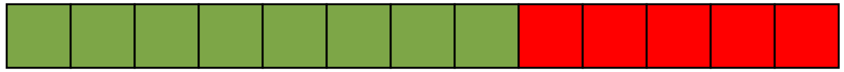
Containers – General Rules

- Holds **copies** of elements.
- **Assumes** elements have: Copy Ctor & operator =

Assignable -
types with operator=
and copy Ctor

STL: Sequential Containers

vector<T>



- Contiguous array of elements of type T
- Random access
- Can grow on as needed basis
- Most useful in practice

```
std::vector<int> v(200);
v[0]= 45;
v[100]= 32;
v.emplace_back(60); //C++11
```

Vectors of ints

1)Creating an empty vector and filling it:

```
std::vector<int>   vec;
vec.push_back(42);
vec.emplace_back(42); // equivalent
```

2)Creating a vector with 10 ints with value 42:

```
std::vector<int>   vec(10,42);
std::vector<int>   vec(10); // default is 0
```

3)Initializing a vector like an array:

```
std::vector<int>   vec { 42, 52, 62 };
```

4)Initializing a vector from iterators:

```
std::vector<int>   v2(vec.begin(),vec.end());
```

Vectors of objects (folder 1)

1)Creating an empty vector and filling it:

```
std::vector<MyClass>   vec;
vec.push_back(MyClass{42,43});
vec.emplace_back(42,43); // more efficient
```

2)Creating a vector with 10 objs:

```
std::vector<MyClass>  vec(10,MyClass{42,43});
std::vector<MyClass>  vec(10); // default ctor
```

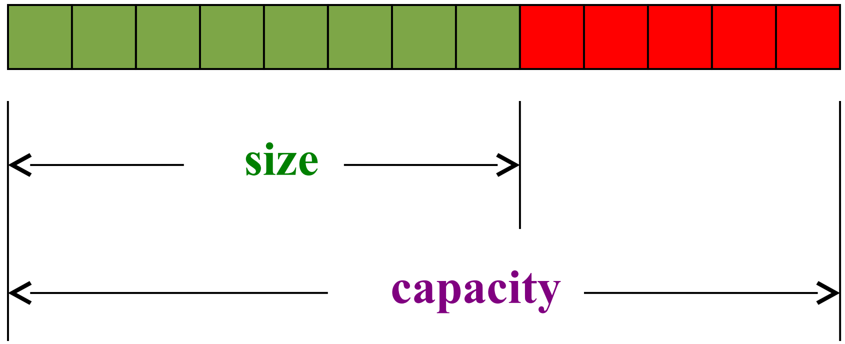
3)Initializing a vector like an array (calls ctor):

```
std::vector<MyClass>  vec { {42,43}, {52}, {62,72} };
```

4)Initializing a vector from iterators:

```
std::vector<MyClass>  v2(vec.begin(),vec.end());
```

size and capacity



- **The first "size" elements are constructed (initialized)**
- **The last "capacity - size" elements are uninitialized**
- push_back / emplace_back use the uninitialized elements until they are full; then, they **multiply the vector capacity by 2**.

emplace_back / push_back Average Time Complexity

If we inserted **n** elements we paid:

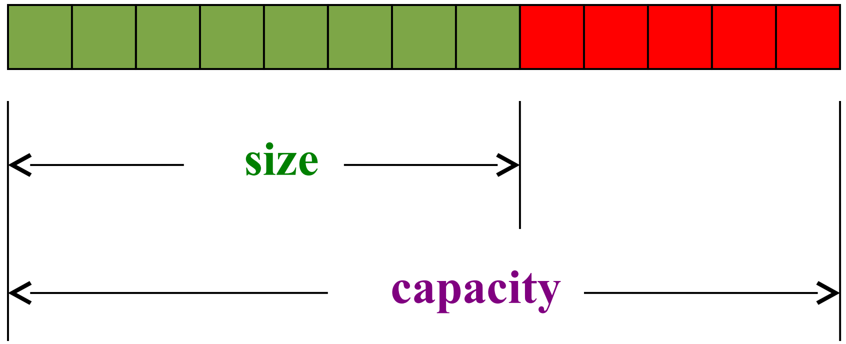
1+**2**+1+**4**+1+1+1+**8**+…+**n** =

O(n) + **1+2+4+…+n** =

O(n)

On average an each insertion cost O(1)

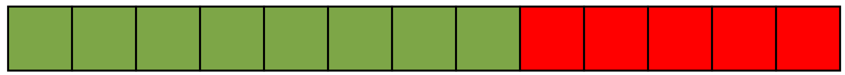
size and capacity methods



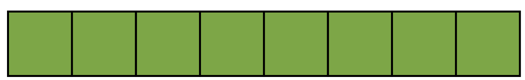
- `uint **size**() const;`
- `uint **capacity**() const;`
- `void **reserve**(uint new_capacity);`

```
  // ensure that the capacity is
  // at least "new_capacity".
```

vector<T> v



v.shrink_to_fit() // c++11



Accessing elements

Without boundary checking:

- **reference operator[](size_type n)**
- **const_reference operator[](size_type n) const**

With boundary checking:

- **reference at(size_type n)**
- **const_reference at(size_type n) const**

Associated types in vector (folder 8)

vector<typename T>::

- value_type - The type of object, T, stored
- reference - Reference to T
- const_reference - const Reference to T
- iterator - Iterator used to iterate through a vector *(how would you write it?)*

vectors: C++ vs. Java

- Look at cplusplus documentation of vector.
- Look at Java documentation of Vector.
- Differences:
  - Simple class vs. interface and vtable.
  - Simple elements vs. class elements.
  - Two accessors (with and without range check) vs. a single accessor

deque

- More efficient insertion at start and middle;
- Less efficient deallocation.
- How do we know? - performance tests:
- https://www.codeproject.com/Articles/5425/An-In-Depth-Study-of-the-STL-Deque-Container
- Implementation – non contiguous blocks:
  https://stackoverflow.com/a/6292437/827927

basic_strings

- The well-known **string** is just a typedef for **basic_string<char>.**
- **basic_string** can be used with any char-like type (folder 1).

	String operations (e.g. substr, replace, stol)	Non-trivial classes (e.g. with vptr, heap usage)
vector	No	Yes
basic_ string	Yes	No

STL: Associative Containers

Associative Containers

Supports efficient retrieval of elements (values) based on keys.

(Typical) Implementation:

- red-black binary trees
- hash-table

Sorted Associative Containers

set

- A set of unique keys ordered by <

map

- Associate a value to key (associative array)
- Unique value of each key, ordered by <

multiset, multimap

- Same, but allow multiple values

unordered_set, unordered_map

- Same, but without order (faster).

Sorted Associative Containers & Order

- Sorted associative containers assume that their elements are *LessThanComparable*.
- They use operator< as default order.
- We can control order using our own

comparison function.

- We need to use a **functor**.

Functor

A functor in C++ is an object with an **operator()**. Examples:

- Pointer to function (like in C);
- A class that implements **`operator();`**
- Lambda [] expressions.

Example (see also folder 2)

```
class c_str_less {
public:
   bool operator()(const char* s1,
                   const char* s2) {
     return (strcmp(s1,s2) < 0);
   }
};

c_str_less cmp;  // declare an object
if(cmp("aa","ab"))
    …
if( c_str_less()("a","b") )
```

Creates temporary objects, and then call operator()

Template comparator example

```
template<typename T>
class less {
public:
  bool operator()(const T& lhs, const T& rhs)
  { return lhs < rhs; }
};

  less<int> cmp;  // declare an object
  if( cmp(1,2) )
    …
  if( less<int>()(1,2) )
    …
```

Creates temporary objects, and then call operator()

Using Comparators

```
// ascending order
// uses operator < for comparison
set<int>  s1;
set<int,less<int>> s1; // same

// descending order
// uses operator > for comparison
set<int,greater<int>> s2;
```

Using Comparators

```
set<int,MyComp> s3;
```

Creates a default constructed MyComp object.

```
MyComp cmp(42);
set<int,MyComp> s4(cmp);
```

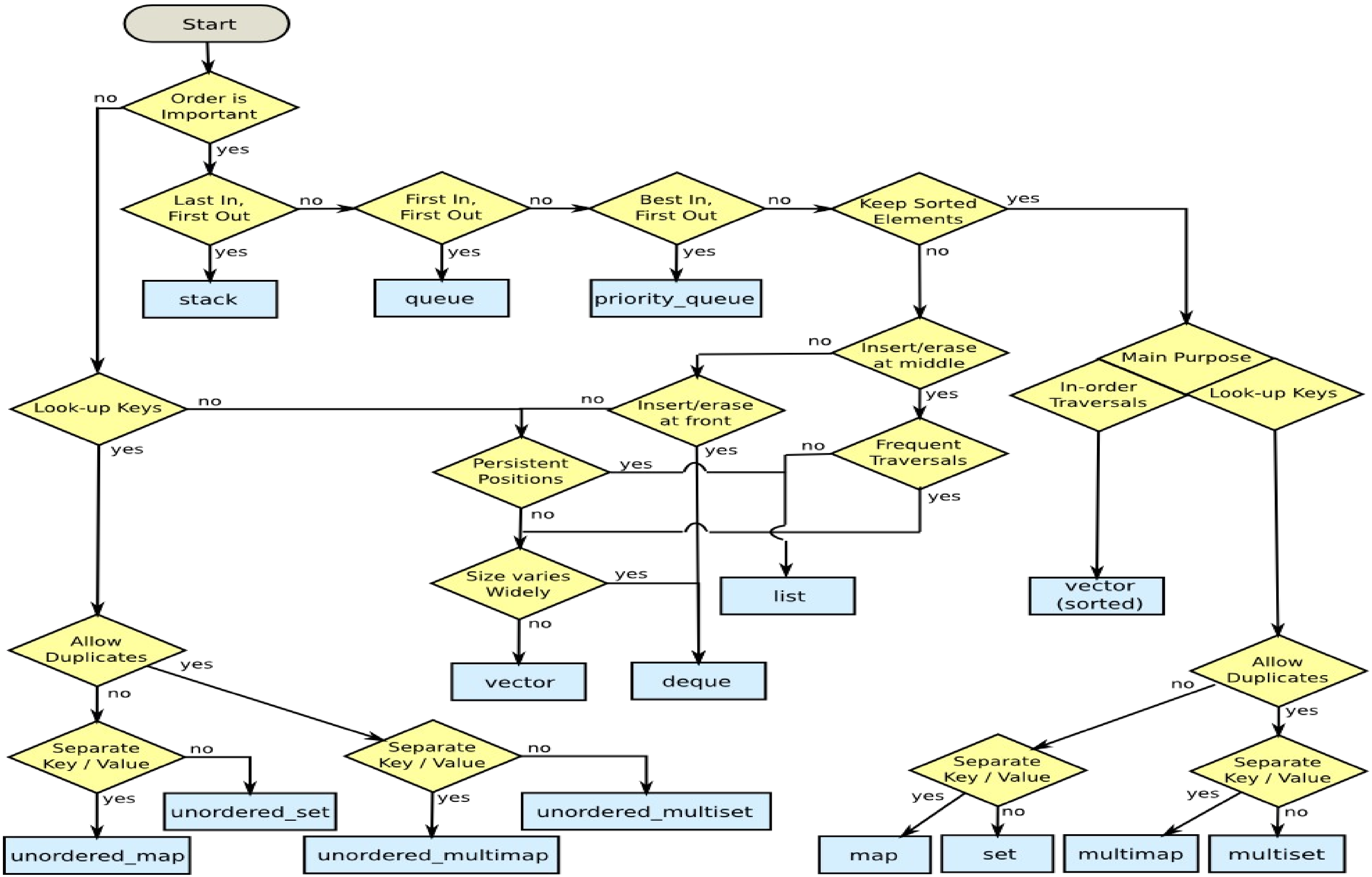
Use given MyComp object.

Why should we use classes as functors?

So that we get the “power” of classes:

- Inheritance.
- Parameterize our functions in run time. (folder 2).
- Accumulate information.

How to choose a container?



Mikael Persson https://stackoverflow.com/a/22671607/827927