OntarioTech Science

Kourosh Davoudi
kourosh@uoit.ca

Week 10: Containers
Iterators

CSCI 1061: Programming Workshop II

Learning Outcomes

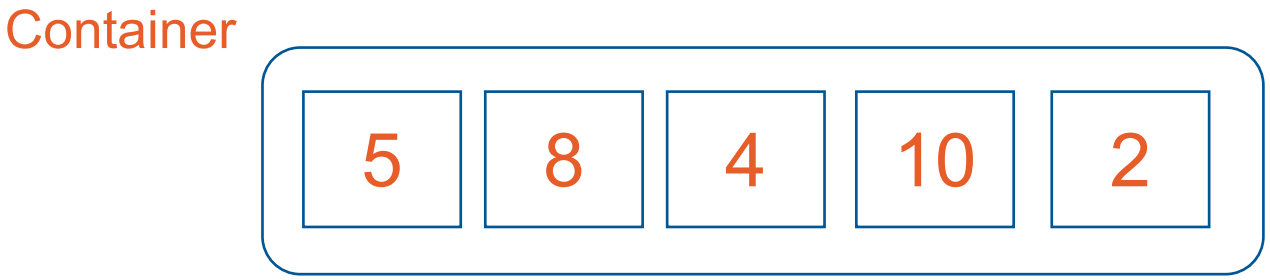
In this week, we learn:

- Different types of Containers
  - Sequential
  - Associative (set and map)
  - Container Adaptors (stack and queue)
- Iterators
  - Different types of access
  - Reverse Iterators
  - Const and Non-const (mutable) Iterators

What is a Container?

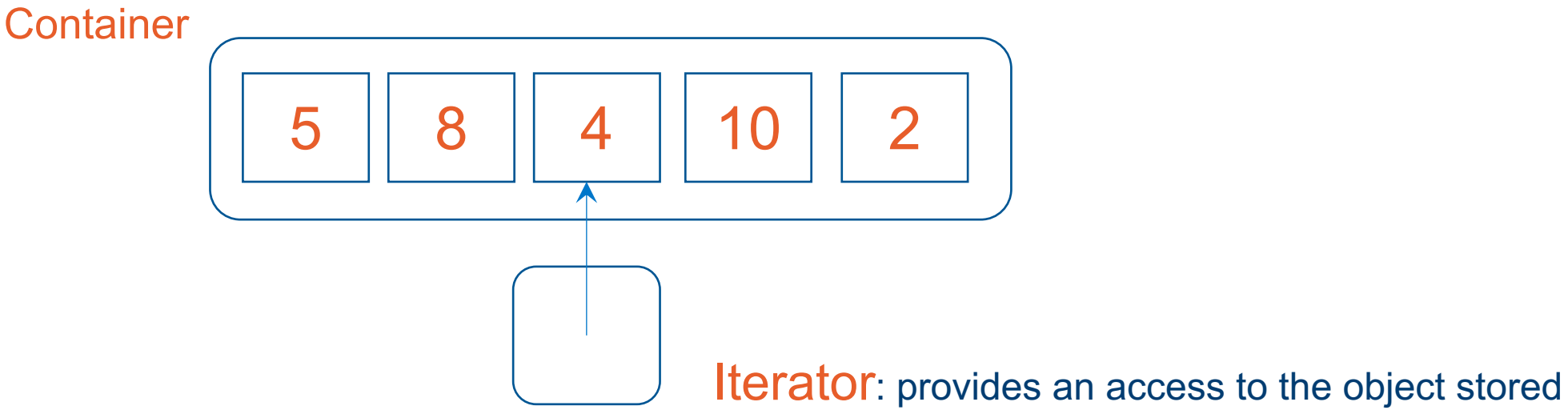
- A **container** is an object that holds other objects.
  - Example: vector class defined in STL

The Standard Template Library



What is iterator?

- An **iterator** is an object used to specify a “position” in a container.
  - All iterators can be incremented to move the position towards the end by one or dereferenced to fetch the value in the container to which the iterator refers.



How to Manipulate Iterators?

++	:advance the iterator to the next data item in a container
--	: moving the iterator to the previous data item in a container
== !=	: test if two iterators point to the same data item in a container
*	: if p is an iterator variable *p give the access to data point p showing the location

Using Iterators and Containers

- C++ STL containers (both sequential and associative) define "helper classes," called iterators, to help iterate over each item in the container.

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

int main()
{
    std::vector<int> v;

    v.push_back(1);
    v.push_back(2);
    v.push_back(3);

    std::vector<int>::iterator i;
    for (i = v.begin(); i != v.end(); ++i)
    {
        std::cout << *i << std::endl;
    }
    return 0;
}
```

Container (vector)

Iterator definition

Dereference operator is used to have access to the objects in the container

vector
Container Example

Special Members of iterators

If c is a container, then

- c.begin()
  - returns an iterator that refers to the first data item in the container.
- c.end()
  - returns an iterator that refers to a position beyond the end of the container.

Kinds of Iterators

- Do all iterators have all of these operators (++,--, etc.)?

NO 

- The vector class has so, let's start with that:

(see iterators.cpp after the next slide)

Kinds of Iterators

- Random Access:

  ++ , -- , == , != , * , random access

- Bidirectional

  ++, --, ==, != , *

- Forward

  ++, ==, != , *

Random Access Iterators	stronger than	Bidirectional Iterators	stronger than	Forward Iterators

Note that different containers have different kinds of iterators:

Which iterators I can use with a container?

- Each container class has "own" iterator type
  - Similar to how each data type has own pointer type

std::vector<int>::iterator

Random Access

std::list<int>::iterator

Bidirectional Access

Constant vs Mutable Iterators

- Mutable iterator:
  - `*p` can be assigned value
  - can change corresponding element in container

```
*p = <anything>; // OK
```

Constant and Mutable Iterators

- Constant iterator:
  - * produces read-only version of element
  - Can use *p to assign to variable or output, but cannot change element in container

```
*p = <anything>; // is illegal
```

Constant Iterator

```cpp
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<string> names;
    names.push_back("John");
    names.push_back("Amanda");

    list<string>::const_iterator i;
    for (i = names.cbegin(); i != names.cend(); ++i)
    {
        cout << *i << endl;
    }
    return 0;
}
```

Container (`list`)

const iterator

Reverse Iterators

- To cycle elements in reverse order
  - Requires container with bidirectional iterators
- You may consider the following:

```
list<string>::iterator p;
for (p=container.end();p!=container.begin(); p--)
      cout << *p << " " ;
```

  - But recall: end() is just "sentinel", begin() not!
  - Might work on some systems, but not most

Reverse Iterator

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<string> names;
    names.push_back("john");
    names.push_back("amanda");

    list<string>::reverse_iterator i;
    for (i = names.rbegin(); i != names.rend(); ++i)
    {
        cout << *i << endl;
    }
    return 0;
}
```

Container (`list`)

reverse iterator

Types of Containers

- Sequential Containers:
  - vector
  - list
  - deque
- The Container Adapters
  - stack
  - queue
- Associative Containers
  - set
  - map

Types of Containers

- Sequential Containers:
  - vector
    - The C++ Standard suggests that vector is the sequence type that should be used by default.
  - List
    - If your program requires frequent insertions and deletions in the middle of a sequence, the list should be used
  - deque
    - The deque should be used if frequent insertions and deletions are needed at the beginning and the end of the sequence.

Sequential Containers

- In a sequential container, the position of each item in the container depends on the time and place of insertion but not on the item's value.

  - Dynamic array: `std::vector<T>`
  - Linked list: `std::list<T>`
  - Deque: `std::deque<T>`

Examples

Sequential Containers

TEMPLATE CLASS NAME	ITERATOR TYPE NAMES	KIND OF ITERATORS	LIBRARY HEADER FILE

`list`	`list<T>::iterator`
	`list<T>::const_iterator`
	`list<T>::reverse_iterator`
	`list<T>::const_reverse_iterator`
		Mutable bidirectional
		Constant bidirectional
		Mutable bidirectional
		Constant bidirectional
			`<list>`

`vector`	`vector<T>::iterator`
	`vector<T>::const_iterator`
	`vector<T>::reverse_iterator`
	`vector<T>::const_reverse_iterator`
		Mutable random access
		Constant random access
		Mutable random access
		Constant random access
			`<vector>`

`deque`	`deque<T>::iterator`
	`deque<T>::const_iterator`
	`deque<T>::reverse_iterator`
	`deque<T>::const_reverse_iterator`
		Mutable random access
		Constant random access
		Mutable random access
		Constant random access
			`<deque>`

From © Absolute C++

Sequential Containers

MEMBER FUNCTION (c IS A CONTAINER OBJECT)	MEANING
`c.size( )`	Returns the number of elements in the container.
`c.begin( )`	Returns an iterator located at the first element in the container.
`c.end( )`	Returns an iterator located one beyond the last element in the container.
`c.rbegin( )`	Returns an iterator located at the last element in the container. Used with `reverse_iterator`. Not a member of `slist`.
`c.rend( )`	Returns an iterator located one beyond the first element in the container. Used with `reverse_iterator`. Not a member of `slist`.
`c.push_back(`*Element*`)`	Inserts the *Element* at the end of the sequence. Not a member of `slist`.
`c.push_front(`*Element*`)`	Inserts the *Element* at the front of the sequence. Not a member of `vector`.
`c.insert(`*Iterator,Element*`)`	Inserts a copy of *Element* before the location of *Iterator*.
`c.erase(`*Iterator*`)`	Removes the element at location *Iterator*. Returns an iterator at the location immediately following. Returns `c.end( )` if the last element is removed.
`c.clear( )`	A `void` function that removes all the elements in the container.
`c.front( )`	Returns a reference to the element in the front of the sequence. Equivalent to `*(c.begin( ))`.
`c1 == c2`	True if `c1.size( ) == c2.size( )` and each element of `c1` is equal to the corresponding element of `c2`.
`c1 != c2`	`!(c1 == c2)`

From © Absolute C++

Associative Containers

- Associative container: allows us to use non integer as the index !
- Example (`map`):

```
map<string, int> super_heros;
super_heros["batman"] = 32;
```

Key

Value

- Capabilities:
  - Add elements
  - Delete elements
  - Ask if element is in set

Map Member Functions

MEMBER FUNCTION (m IS A MAP OBJECT)	MEANING
`m.insert(`*Element*`)`	Inserts *Element* in the map. *Element* is of type `pair<`*KeyType*`, T>`. Returns a value of type `pair<iterator, bool>`. If the insertion is successful, the second part of the returned pair is `true` and the iterator is located at the inserted element.
`m.erase(`*Target_Key*`)`	Removes the element with the key *Target_Key*.
`m.find(`*Target_Key*`)`	Returns an iterator located at the element with key value *Target_Key*. Returns `m.end( )` if there is no such element.
`m[`*Target_Key*`]`	Returns a reference to the object associated with the *Target_Key*. If the map does not already contain such an object, then a default object of type `T` is inserted.
`m.size( )`	Returns the number of pairs in the map.
`m.empty( )`	Returns `true` if the map is empty; otherwise, returns `false`.
`m1 == m2`	Returns `true` if the maps contain the same pairs; otherwise, returns `false`.

Map Example

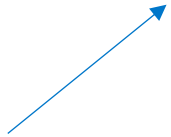
```
#include <map>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    map<string, int> super_heros;
    super_heros["batman"] = 32;
    super_heros["wolverine"] = 137;
    super_heros["jean gray"] = 25;
    super_heros["superman"] = 35;

    map<string, int>::iterator i;
    for (i = super_heros.begin(); i != super_heros.end(); ++i)
    {
        cout << "Age of " << i->first << " is " << i->second << endl;
    }

    return 0;
}
```



Can use [ ] notation to access the map

Associative Containers

- Associative container: allows us to use non integer as the index !
- Example (`set`):

```
set<string> super_heros;
super_heros.insert("batman");
super_heros.insert("wolverine");
```


Key

- Capabilities:
  - Add elements
  - Delete elements
  - Ask if element is in set

Set Member Functions

MEMBER FUNCTION (s IS A SET OBJECT)	MEANING
`s.insert(`*Element*`)`	Inserts a copy of *Element* in the set. If *Element* is already in the set, this has no effect.
`s.erase(`*Element*`)`	Removes *Element* from the set. If *Element* is not in the set, this has no effect.
`s.find(`*Element*`)`	Returns an iterator located at the copy of *Element* in the set. If *Element* is not in the set, `s.end( )` is returned. Whether the iterator is mutable or not is implementation dependent.
`s.erase(`*Iterator*`)`	Erases the element at the location of the *Iterator*.
`s.size( )`	Returns the number of elements in the set.
`s.empty( )`	Returns `true` if the set is empty; otherwise, returns `false`.
`s1 == s2`	Returns `true` if the sets contain the same elements; otherwise, returns `false`.

Set Example

```
#include <set>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    set<string> super_heros;

    super_heros.insert("batman");
    super_heros.insert("wolverine");
    super_heros.insert("jean gray");
    super_heros.insert("superman");

    set<string>::iterator p;

    cout << "Set contains:" << endl;
    for (p = super_heros.begin(); p != super_heros.end(); ++p)
    {
        cout << *p << endl;
    }

    cout << "Is batman is in the set?" << endl;
    if (super_heros.find("batman")==super_heros.end( ))
        cout << " no " << endl;
    else
        cout << " yes " << endl;

    return 0;
}
```

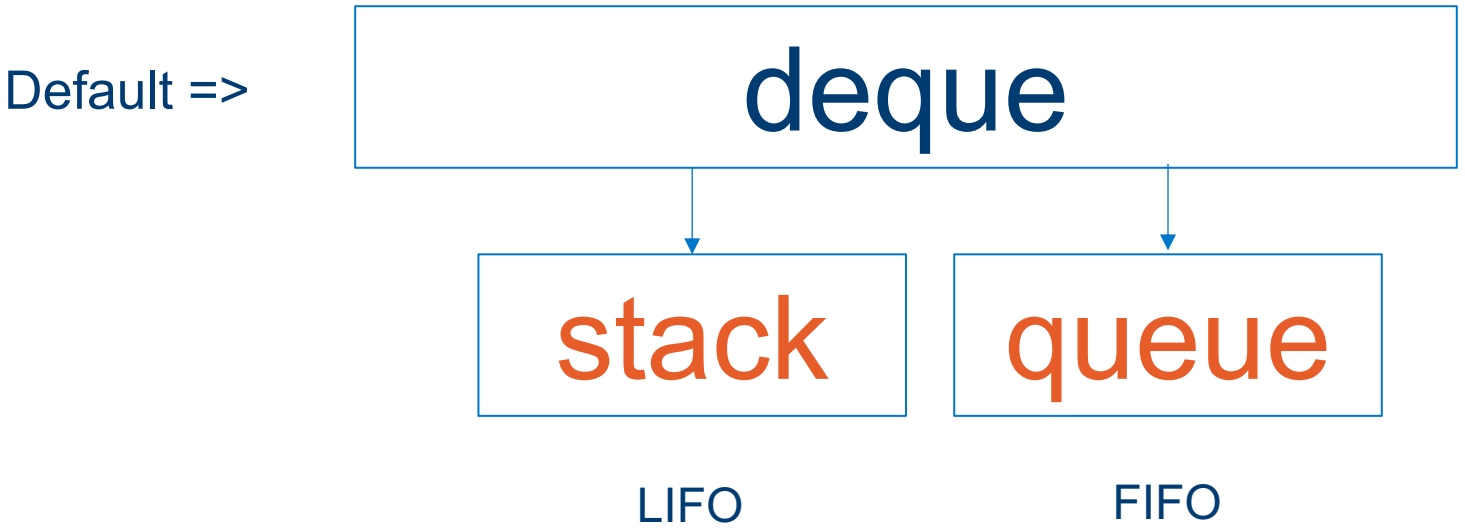
find returns an iterator to the (key, value) pair if the key is found; otherwise, it returns an iterator equal to end().

Container Adapters

- A container adapter does not directly implement the structures that hold the data items.
- Rather, it provides a new interface between the user and an existing container.
- These are called adapters because they use one of the three sequence containers (vector, list, or deque)

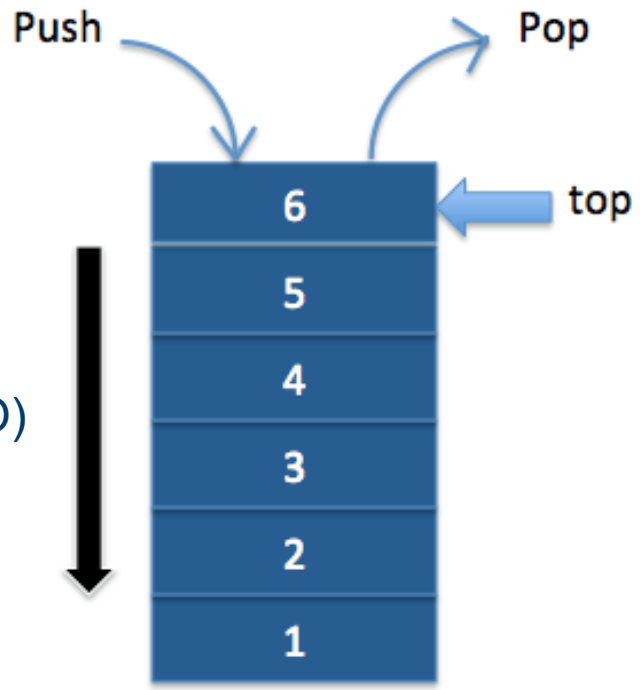
Container Adapters stack and queue

- Example: stack template class by default implemented on top of deque template class:

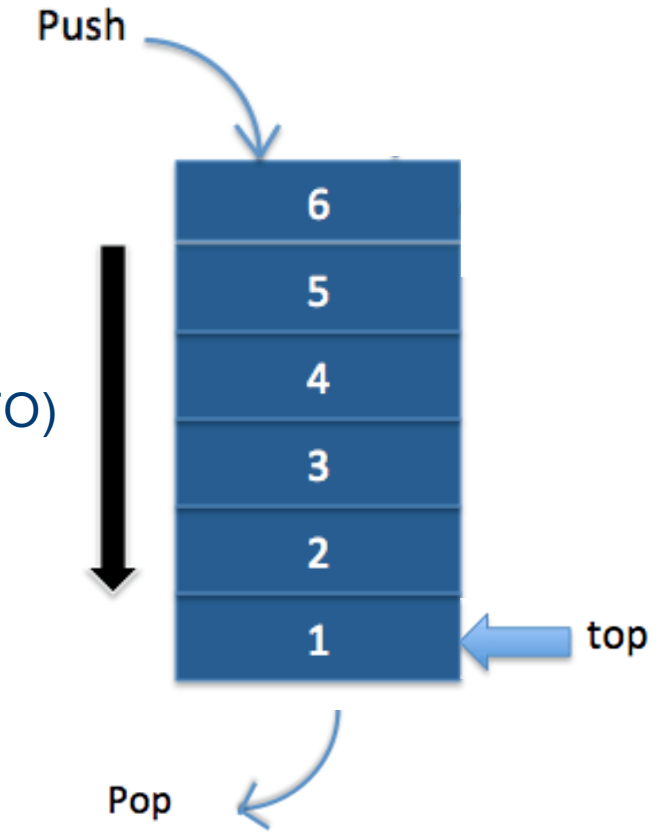


Stack and Queue

Last In First Out (LIFO)



First In First Out (FIFO)



Stack Member Functions

MEMBER FUNCTION (s IS A STACK OBJECT)	MEANING
`s.size( )`	Returns the number of elements in the stack.
`s.empty( )`	Returns `true` if the stack is empty; otherwise, returns `false`.
`s.top( )`	Returns a mutable reference to the top member of the stack.
`s.push(`*Element*`)`	Inserts a copy of *Element* at the top of the stack.
`s.pop( )`	Removes the top element of the stack. Note that `pop` is a `void` function. It does not return the element removed.
`s1 == s2`	True if `s1.size( ) == s2.size( )` and each element of `s1` is equal to the corresponding element of `s2`; otherwise, returns `false`.
The `stack` template class also has a default constructor, a copy constructor, and a constructor that takes an object of any sequence class and initializes the stack to the elements in the sequence. It also has a destructor that returns all storage for recycling, and a well-behaved assignment operator.	

Queue Member Functions

MEMBER FUNCTION (q IS A QUEUE OBJECT)	MEANING
`q.size( )`	Returns the number of elements in the queue.
`q.empty( )`	Returns `true` if the queue is empty; otherwise, returns `false`.
`q.front( )`	Returns a mutable reference to the front member of the queue.
`q.back( )`	Returns a mutable reference to the last member of the queue.
`q.push(`*Element*`)`	Adds *Element* to the back of the queue.
`q.pop( )`	Removes the front element of the queue. Note that `pop` is a `void` function. It does not return the element removed.
`q1 == q2`	`True` if `q1.size( ) == q2.size( )` and each element of `q1` is equal to the corresponding element of `q2`; otherwise, returns `false`.