



四川大學  
SICHUAN UNIVERSITY

# Object Oriented Programming—C++

## Lecture5 Iterators and Pointers

Qijun Zhao

College of Computer Science

Sichuan University

Spring 2023

## Containers

- Containers are ways to collect related data together and work with it logically
- Two types of containers: **sequence** and **associative**
- Container adaptors wrap existing containers to permit new/restrict access to the interface for the clients.

# There are two types of containers:

## Sequence:

- Containers that can be accessed sequentially
- Anything with an inherent order goes here!

## Associative

- Containers that don't necessarily have a sequential order
- More easily searched
- Maps and sets go here!

# Sequence Containers: Summary

- Sequence containers are for when you need to enforce some order on your information!
- Can usually use an **std::vector** for most anything
- If you need particularly fast inserts in the front, consider an **std::deque**
- For joining/working with multiple lists, consider an **std::list** (very rarely)

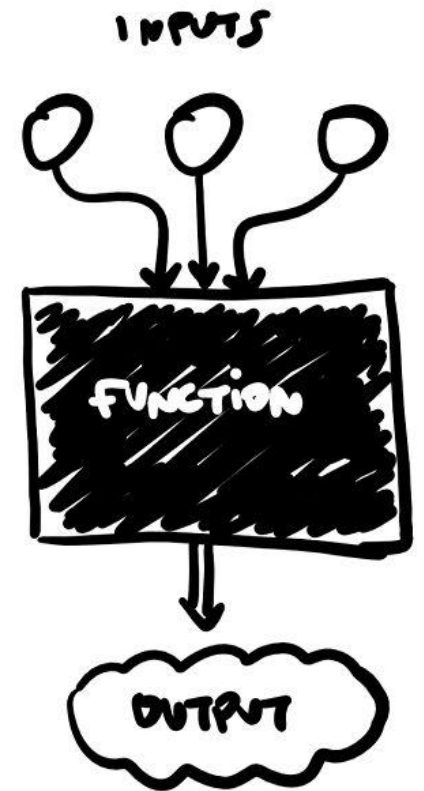
Lots of similarities between maps/sets! Broad tips:

- Unordered containers are **faster**, but can be difficult to get to work with nested containers/collections
- If using **complicated data types**/unfamiliar with hash functions, use an ordered container

# Container Adaptors

Container adaptors are “wrappers” to existing containers!

- Wrappers **modify the interface** to sequence containers and change what the client is allowed to do/how they can interact with the container.



# The STL

```
template <class T, class Container = deque<T> > class queue;
```

**queues** are implemented as **containers adaptors**, which are classes that use an encapsulated object of a specific container class as its **underlying container**, providing a specific set of member functions to access its elements. Elements are **pushed** into the "**back**" of the specific container and **popped** from its "**front**".

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

- empty
- size
- front
- back
- push\_back
- pop\_front

- **Iterators**
- **Pointers**



## All containers are collections of objects...

So how do we access those objects?

- What if we want to print out everything in a vector?
- Or loop until we find a certain object in a set?

How is this done in the STL?

# All containers are collections of objects...

We'd like to have a for-loop, probably!

What would that look like?

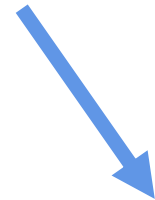
```
for (initialization; termination condition; increment) {
```



**Where do we  
start?**



**When do we  
end?**



**something...++???**

# Introducing Iterators!

Containers all implement something called an **iterator** to do this!

- Iterators let you access **all** data in containers programmatically!
- An iterator has a certain **order**; it “knows” what element will come next
  - Not necessarily the same each time you iterate!

# Think of your container as a file cabinet!

An iterator lets you go through the files one at a time!

- You can see where the front and back of your drawer are.
- You can move your finger from one to the next, because you kept your place.
- You can take out any file you've your hand on, and read/write whatever you'd like in it.
- You can compare the relative location of any two files just by looking at where they are in the cabinet.



All containers implement iterators, but they're not all the same!

- Each container has its own iterator, which can have different behavior.

- All iterators implement a few shared operations:

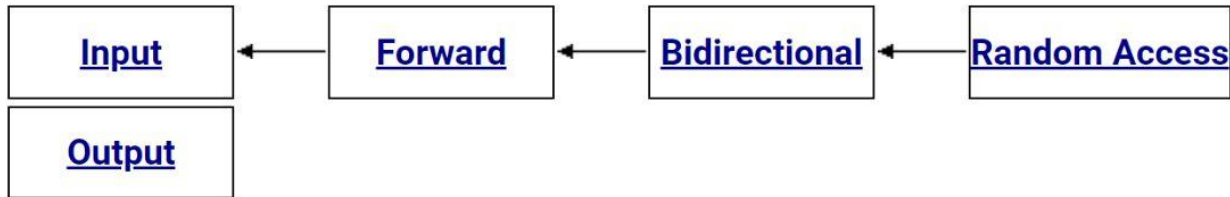
- Initializing → `iter = s.begin();`
- Incrementing → `++iter;`
- Dereferencing → `*iter;`
- Comparing → `iter != s.end();`
- Copying → `new_iter = iter;`

What other behaviors can iterators have?

# That depends!

## 🌀 Iterator categories

Iterators are classified into five categories depending on the functionality they implement:



Input and output iterators are the most limited types of iterators: they can perform sequential single-pass input or output operations.

Forward iterators have all the functionality of input iterators and if they are not **constant iterators** also the functionality of output iterators, although they are limited to one direction in which to iterate through a range (forward). All standard containers support at least forward iterator types.

Bidirectional iterators are like forward iterators but can also be iterated through backwards.

Random-access iterators implement all the functionality of bidirectional iterators, and also have the ability to access ranges non-sequentially: distant elements can be accessed directly by applying an offset value to an iterator without iterating through all the elements in between. These iterators have a similar functionality to standard pointers (pointers are iterators of this category).

## What does that mean?

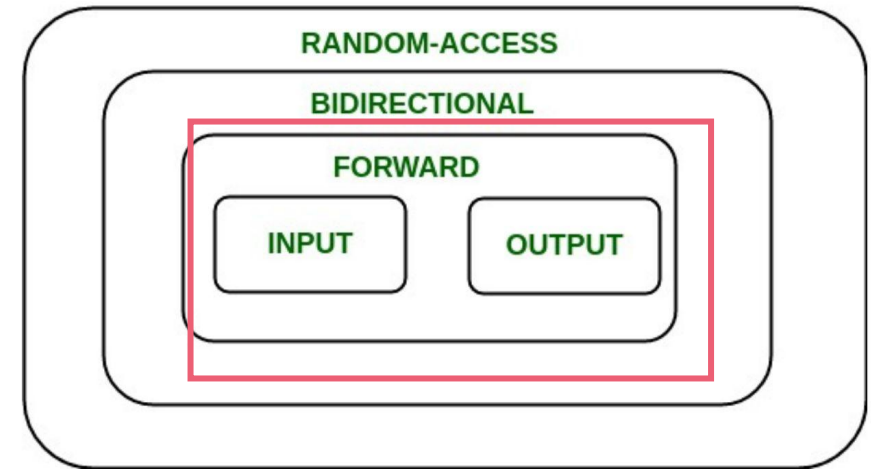
**Forward** iterators are the minimum level of functionality for standard containers.

- **Input** iterators can appear on the RHS (right hand side) of an = operator

```
auto elem = *it;
```

- **Output** iterators can appear on the LHS (left hand side) of an = operator

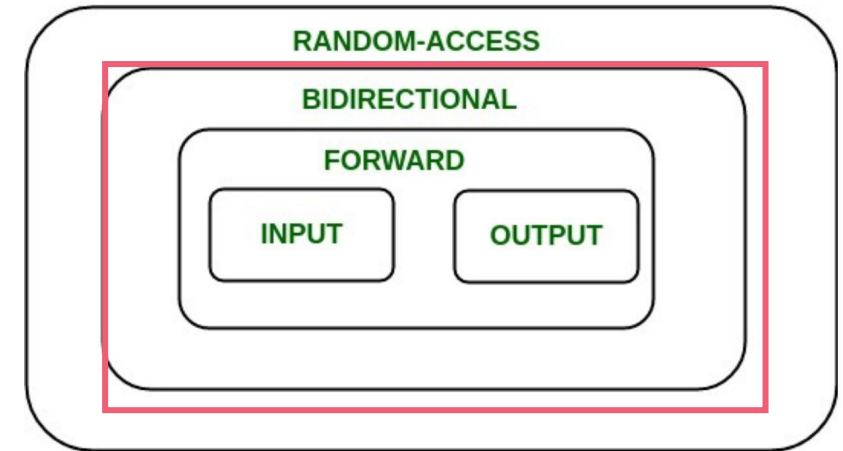
```
*elem = value;
```



# What does that mean?

**Bidirectional** iterators can go forward as well as backward!

- `--iter;`
- Still has the same functionality of forward iterators!

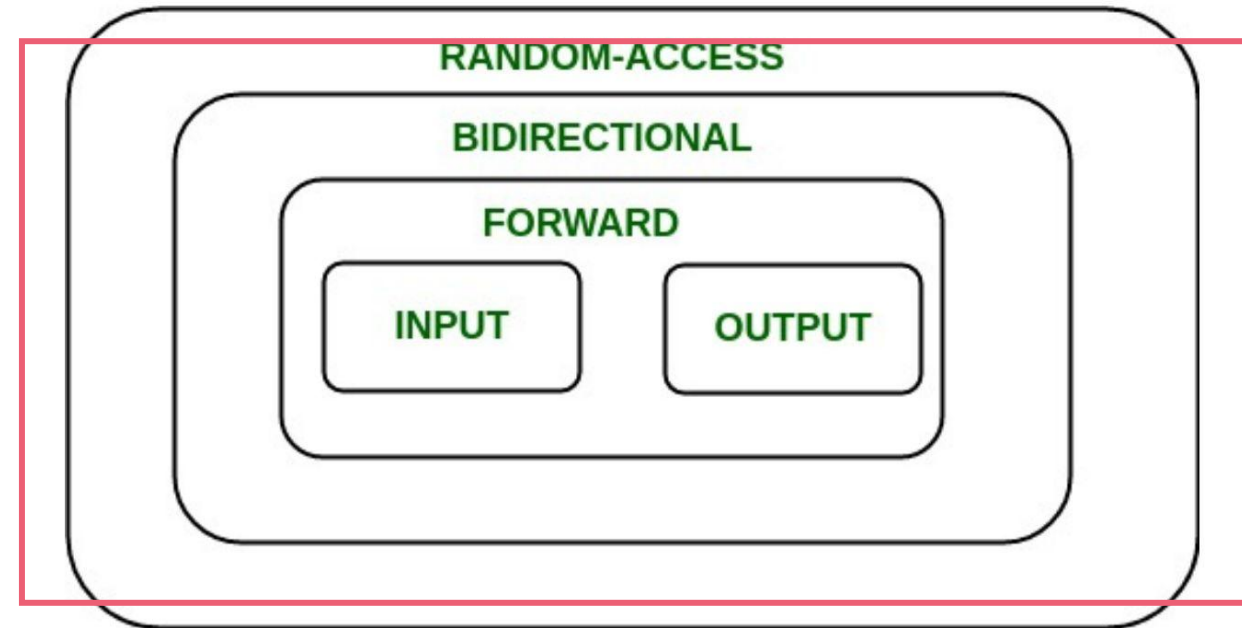




## What does that mean?

**Random-access** iterators allow you to directly access values without visiting all elements sequentially.

- `iter += 5;`
- Think of vectors; `vec[1]` or `vec[17]` or...
- Be careful not to go out of bounds!



`iter += 3; ?`

# Categorizing STL iterators

Vectors and deques have the most powerful iterators!

- Creating your own containers means creating their iterators as well.
- You can access elements in stacks and queues one-by-one, but you have to change the container to do so!
- Iteration with iterators is const

Container	Type of Iterator
Vector	Random-Access
Deque	Random-Access
List	Bidirectional
Map	Bidirectional
Set	Bidirectional
Stack	No Iterator
Queue	No Iterator
Priority Queue	No Iterator

## Why `++iter`; ?

Why not `iter++`; ?

There's a difference in when the value is returned!

- `iter++` returns the value **before** being incremented.
- `++iter` returns the value **after** being incremented.

Using an iterator, we already have the previous value! It's slightly more inefficient to use `iter++`;

This is now outdated!  
`iter++` to your heart's content!

## Let's check out that for loop again!

```
for (initialization; termination condition; increment) {  
    auto iter = set.begin()    iter != set.end();    ++iter
```

Now we can access each element individually!

If we want the element and not just a reference to it, we dereference (\*iter).

```
const auto& elem = *iter;
```

## Let's check out that for loop again!

If we have a map, we can use **structured binding** to be more efficient while dereferencing!

```
std::map<int> map{{1, 6}, {1, 8}, {0, 3}, {3, 9}};  
for (auto iter = map.begin(); iter != map.end(); ++iter) {  
    const auto& [key, value] = *iter;           // structured binding!
```

This is a C++ **for-each** loop!

- Iterators
- **Pointers**

Iterators are a particular type of pointer!

- Iterators “point” at particular elements in a **container**.
- Pointers can “point” at any objects in your code!

# Memory and You

Variables created in your code take up **space** on your computer.

They live in **memory** at **specific addresses**.

Pointers reference those memory addresses and not the object themselves!

```
int val = 18;
```

```
int* ptr = &val;
```

**int\* ptr**

int val



#0106





# Dereferencing

Pointers are marked by the asterisk (\*) next to the type of the object they're pointing at when they're declared.

The **address of a variable** can be accessed by using **&** before its name, same as when passing by reference!

If you want to access the data stored at a pointer's address, dereference it using an asterisk again.

```
std::cout >> *ptr >> std::endl;
```

```
int val = 18;
```

```
int* ptr = &val;
```

```
int* ptr
```

int val



#0106



## What if the object has member variables?

If we need to access a pointer's object's member variables, instead of dereferencing (`*ptr`) and then accessing (`.var`), there's a shorthand!

**`*ptr.var == ptr->var`**

## What's the difference?

- Iterators are a type of pointer!
- Iterators have to point to elements in a container, but pointers can point to any object!
  - Why is this? All objects stored inside the big container known as **memory**!
- Can access memory addresses with **&** and the data at an address/pointer using **\***



四川大學  
SICHUAN UNIVERSITY

# Coding for love, Coding for the world

Qijun Zhao

qjzhao@scu.edu.cn

