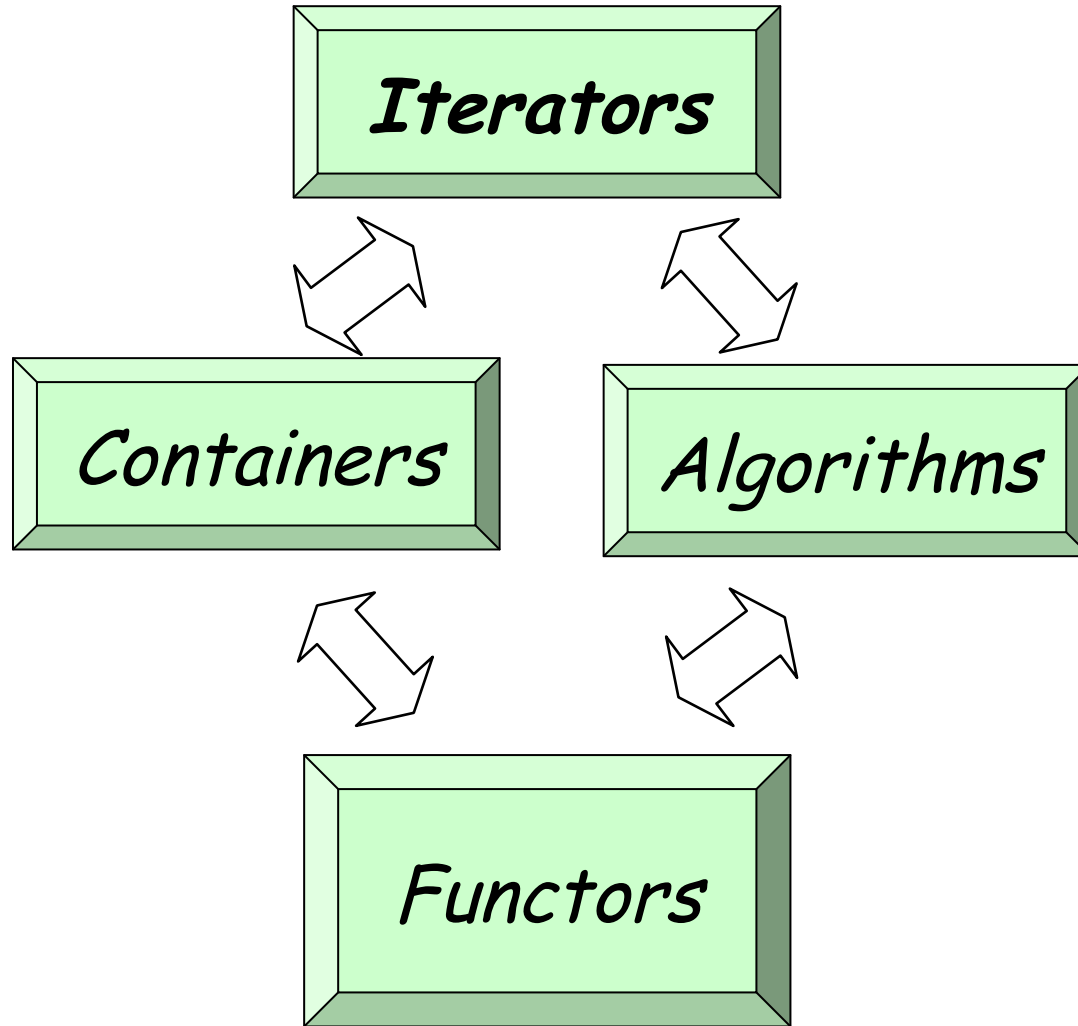


The Standard C++ Library – Iterators

Version 1: Dr. Ofir Pele

Version 2: Dr. Erel Segal-Halevi

Main Components



Why Iterators?

- Instead of writing e.g. "find" for vector, "find" for unordered_set, "find" for array, etc. -
- - we write only **one** find that accepts two iterators (begin and end):
<http://www.cplusplus.com/reference/algorithm/find/>
- The same "find" would work for *any* container that defines the iterators correctly, and even for non-containers such as "range", "chain".

Iterator types

**++,
input**

Input Iterator

**++,
output**

Output Iterator

++, I/O

Forward Iterator

**++, --,
I/O**

Bi-directional Iterator

**Pointer
arithmetic**

Random-Access Iterator

Iterator Types

	Output	Input	Forward	Bi-directional	Random
Read		<code>x = *i</code>	<code>x = *i</code>	<code>x = *i</code>	<code>x = *i</code>
Write	<code>*i = x</code>		<code>*i = x</code>	<code>*i = x</code>	<code>*i = x</code>
Iteration	<code>++</code>	<code>++</code>	<code>++</code>	<code>++</code> , <code>--</code>	<code>++</code> , <code>--</code> , <code>+</code> , <code>-</code> , <code>+=</code> , <code>-=</code>
Comparison		<code>==</code> , <code>!=</code>	<code>==</code> , <code>!=</code>	<code>==</code> , <code>!=</code>	<code>==</code> , <code>!=</code> , <code><</code> , <code>></code> , <code><=</code> , <code>>=</code>

- Output: write only and can write only once
- Input: read many times each item
- Forward supports both read and write
- Bi-directional support also decrement
- Random supports random access
(just like C pointer)

Iterators & Containers

Input/output/forward iterators:

- iostreams (folder 1)

Bidirectional iterators:

- list, map, set

Random access iterators:

- vector

Iterators & Containers

```
class NameOfContainer {  
    ...  
    typedef ... iterator; // iterator type  
    iterator begin();      // first element  
    iterator end();       // element after last
```

```
NameOfContainer<...> c  
  
    ...  
  
    NameOfContainer<...>::iterator it;  
    for( it= c.begin(); it!=c.end(); ++it)  
        // do something that changes *it
```

Iterators & Containers: **c++11**

```
class NameOfContainer {  
    ...  
    typedef ... iterator; // iterator type  
    iterator begin();      // first element  
    iterator end();        // element after last
```

```
NameOfContainer<...> c
```

```
...
```

```
for(auto it= c.begin(); it!=c.end(); ++it)  
    // do something that changes *it
```


Iterators & Containers: **c++11**

```
class NameOfContainer {  
    ...  
    typedef ... iterator; // iterator type  
    iterator begin();      // first element  
    iterator end();        // element after last
```

```
NameOfContainer<...> c
```

```
...
```

```
for(auto& val : c)
```

```
    // do something that changes val
```

const_iterators & Containers

```
class NameOfContainer {  
    ...  
    typedef ... const_iterator; // iterator type  
    const_iterator begin() const;    // first element  
    const_iterator end() const;      // element after last
```

```
    NameOfContainer<...> c
```

```
    ...
```

```
    NameOfContainer<...>::const_iterator it;
```

```
    for( it= c.begin(); it!=c.end(); ++it)
```

```
        // do something that does not change *it
```

const_iterators & Containers: c++11

```
class NameOfContainer {  
    ...  
    typedef ... const_iterator; // iterator type  
    const_iterator cbegin() const;    // first element  
    const_iterator cend() const;      // element after last
```

```
    NameOfContainer<...> c
```

```
    ...
```

```
    for(auto it= c.cbegin(); it!=c.cend(); ++it)  
        // do something that does not change *it
```

const_iterators & Containers: c++11

```
class NameOfContainer {  
    ...  
    typedef ... const_iterator; // iterator type  
    const_iterator cbegin() const;    // first element  
    const_iterator cend() const;      // element after last
```

```
NameOfContainer<...> c
```

```
...
```

```
for(const auto& val : c)
```

```
    // do something that does not change val
```

const_iterators & Containers

...

```
const_iterator cbegin() const;  
const_iterator cend() const;  
const_iterator begin() const;  
const_iterator end() const;
```

...

```
iterator begin();  
iterator end();
```

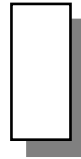
Note that the `begin()` and `end()` methods that return regular iterator are not **const** methods. i.e: if we get a container by `const` (`const ref`, ...) we can't use these methods. We have to use the methods that return **const_iterator**

[end of first week]

IntBufferSwap example revisited

- See folder 2.
- Focus on iterator and `const_iterator`.

Iterators & Sequence Containers

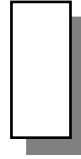


```
SeqContainerName<...> c;
```

```
SeqContainerName<...>::iterator i, j;
```

- `c.insert(i, x)` – inserts `x` before `i`
- `c.insert(i, first, last)`
 - inserts elements in `[first, last)` before `i`
- `c.erase(i)` – erases the element that `i` points to
- `c.erase(i, j)`
 - erase elements in range `[i, j)`

Iterators & Sequence Containers **c++11**



```
SeqContainerName<...> c;
```

```
SeqContainerName<...>::iterator i, j;
```

- **c.emplace(i, p1, ..., pn):**

Constructs and inserts before **i** an object with a constructor that gets **p1, ..., pn** parameters

Iterator validity

- When working with iterators, we have to remember that their validity can change

What is wrong with this code?

```
Container<...> c;
```

```
...
```

```
for(auto i= c.begin(); i!=c.end(); ++i )
```

```
    if( f( *i ) ) { // some test
```

```
        c.erase(i) ;
```

```
    }
```

Iterator validity

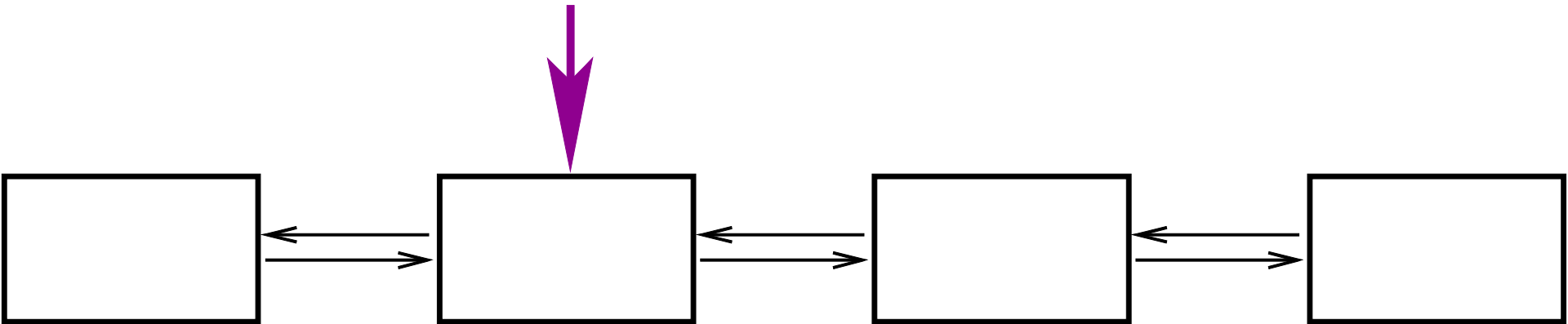
Two cases:

- list, set, map
 - `i` is not a legal iterator

Iterator validity

Two cases:

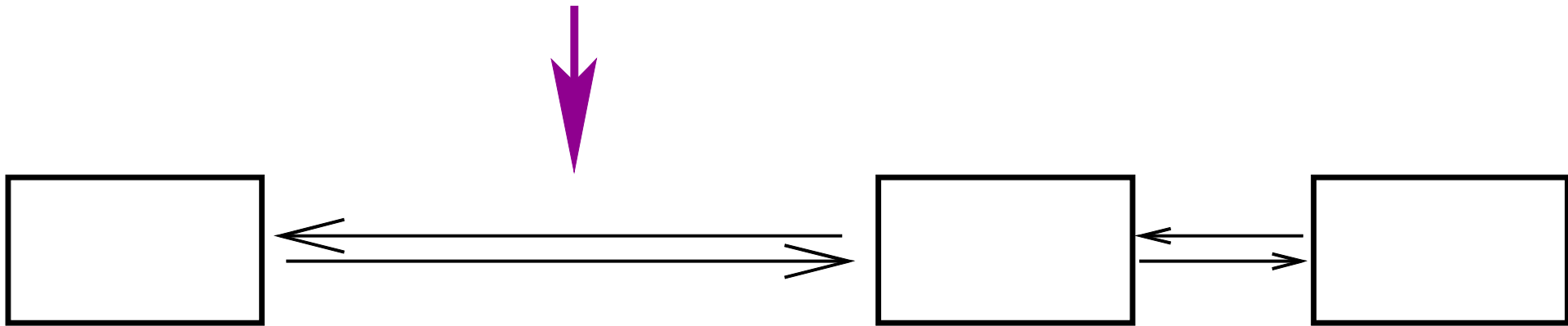
- **list**, set, map
 - `i` is not a legal iterator



Iterator validity

Two cases:

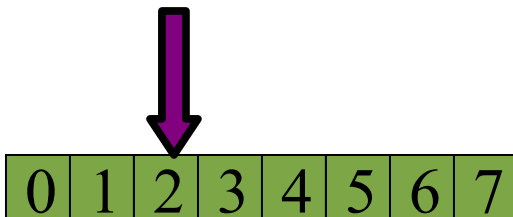
- **list**, set, map
 - `i` is not a legal iterator



Iterator validity

Two cases:

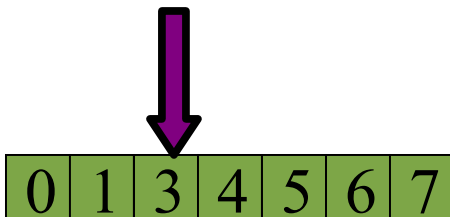
- list, set, map
 - i is not a legal iterator
- **vector**
 - i points to the element after



Iterator validity

Two cases:

- list, set, map
 - `i` is not a legal iterator
- **vector**
 - `i` points to the element after



Iterator validity

Two cases:

- list, set, map
 - `i` is not a legal iterator
- **vector**
 - `i` points to the element after

**In either case,
this is not what we want...**

Erasing during iteration (folder 3)

```
Container<...> c;
```

```
...
```

```
for(auto i= c.begin(); i!=c.end(); /*no ++i*/ )  
    if( f( *i ) ) { // some test  
        i = c.erase(i);  
    } else {  
        ++i;  
    }
```


Iterators & Map

Suppose we work with:

```
map<string,int> dictionary;  
map<string,int>::iterator it;  
...  
it = dictionary.begin();
```

What is the type of `*it` ?

Iterators & Map

Every STL container type Container defines

`Container::value_type`

Type of elements stored in container

- This is the type returned by an iterator

`Container::value_type operator*() ;`

Iterators & Map

- Ok, so what type of elements does a map return?
- `map<KeyType, ValueType>` keeps pairs
 - `KeyType key` – “key” of entry
 - `ValueType value` – “value” of entry

Pairs

```
template< typename T1, typename T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair( const T1& x, const T2& y )
        : first(x), second(y)
    {}
};
```

Map value_type

```
template< typename Key, typename T,  
          typename Cmp = less<Key> >  
class map {  
public:  
    typedef pair<const Key, T> value_type;  
  
    typedef Key key_type;  
    typedef T mapped_type;  
    typedef Cmp key_compare;  
};
```

Using map iterator (folder 4)

```
map<string,int> dict;  
...  
  
for( auto i = dict.cbegin();  
    i != dict.cend();  
    ++i )  
{  
    cout << i->first << " "  
        << i->second << "\n";  
}
```

Using map iterator

```
map<string,int> dict;
```

```
...
```

```
for( const auto& val : dict) {
```

```
    cout << val.first << " "
```

```
        << val.second << "\n";
```

```
}
```

Iterators and Assoc. Containers (folder 4)

Additional set of operations:

- `iterator C::find(key_type const& key)`

Return iterator to first element with **key**.

Return `end()` if not found

- `iterator C::lower_bound(key_type const& key)`

Return iterator to first element greater or equal to **key**

- `iterator C::upper_bound(key_type const& key)`

Return iterator to first element greater than **key**