

Programming and Laboratory - 2

A–L: <https://www.unive.it/data/course/379954>

M–Z: <https://www.unive.it/data/course/379955>

12. List container — Part 3

Giulio Ermanno Pibiri — giulioermanno.pibiri@unive.it

Nicola Prezza — nicola.prezza@unive.it

Department of Environmental Sciences, Informatics and Statistics

Academic year 2022/2023

Overview

- Iterators: basic concepts and types
- Iterator interface: properties and methods
- Implementation of a (forward) iterator for the `list<Val>` container
- Invalidation: common mistakes

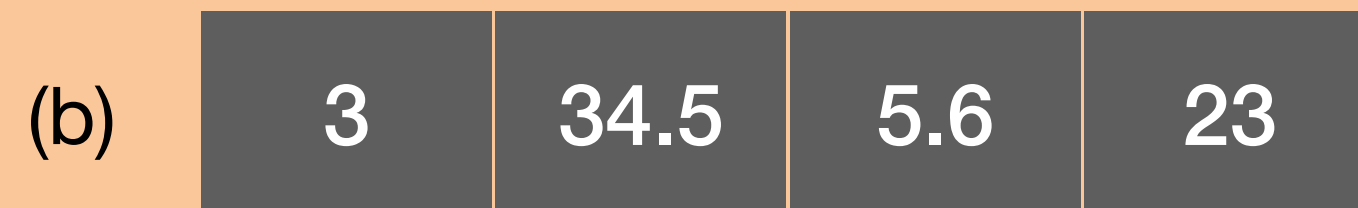
Iterators in C++

- An iterator is an object that points to an element inside a container.
- Similarly to a pointer, an iterator can be used to read the element and/or modify it.
- All containers in the standard template library of C++ (STL, that we will see soon) expose methods to create iterators pointing to their elements.

Iterators in C++

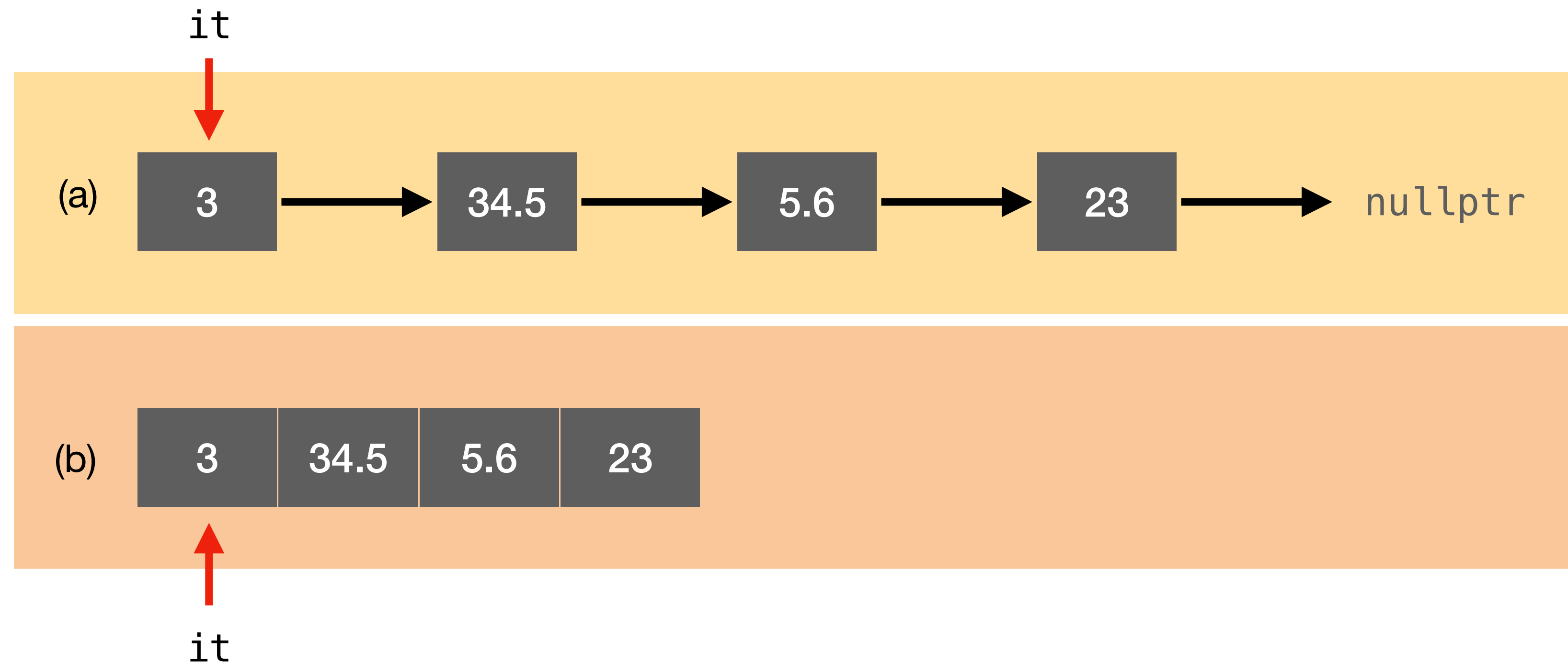
- An iterator is an object that points to an element inside a container.
- Similarly to a pointer, an iterator can be used to read the element and/or modify it.
- All containers in the standard template library of C++ (STL, that we will see soon) expose methods to create iterators pointing to their elements.
- **Q.** Why are iterators useful?
- **A.** Because they provide an **easy interface** to access the data inside a container, abstracting away the details of the actual container implementation.
- Let's consider an example.

Iterators in C++ — Example



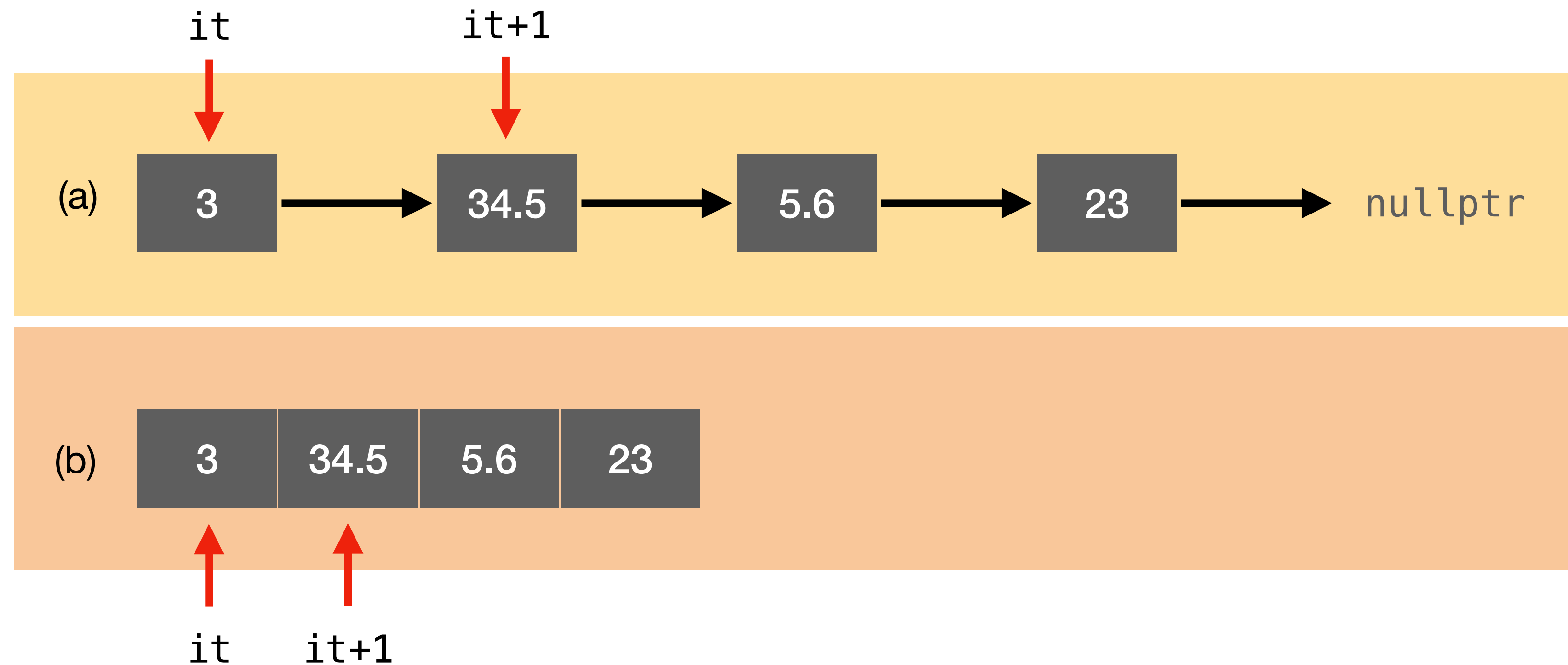
- Two different implementations of a `list<Val>` container: implementation (a) uses a **singly-linked list**; implementation (b) uses an **array**.
- The idea is to make this difference **invisible** to clients of our container via iterators.

Iterators in C++ — Example



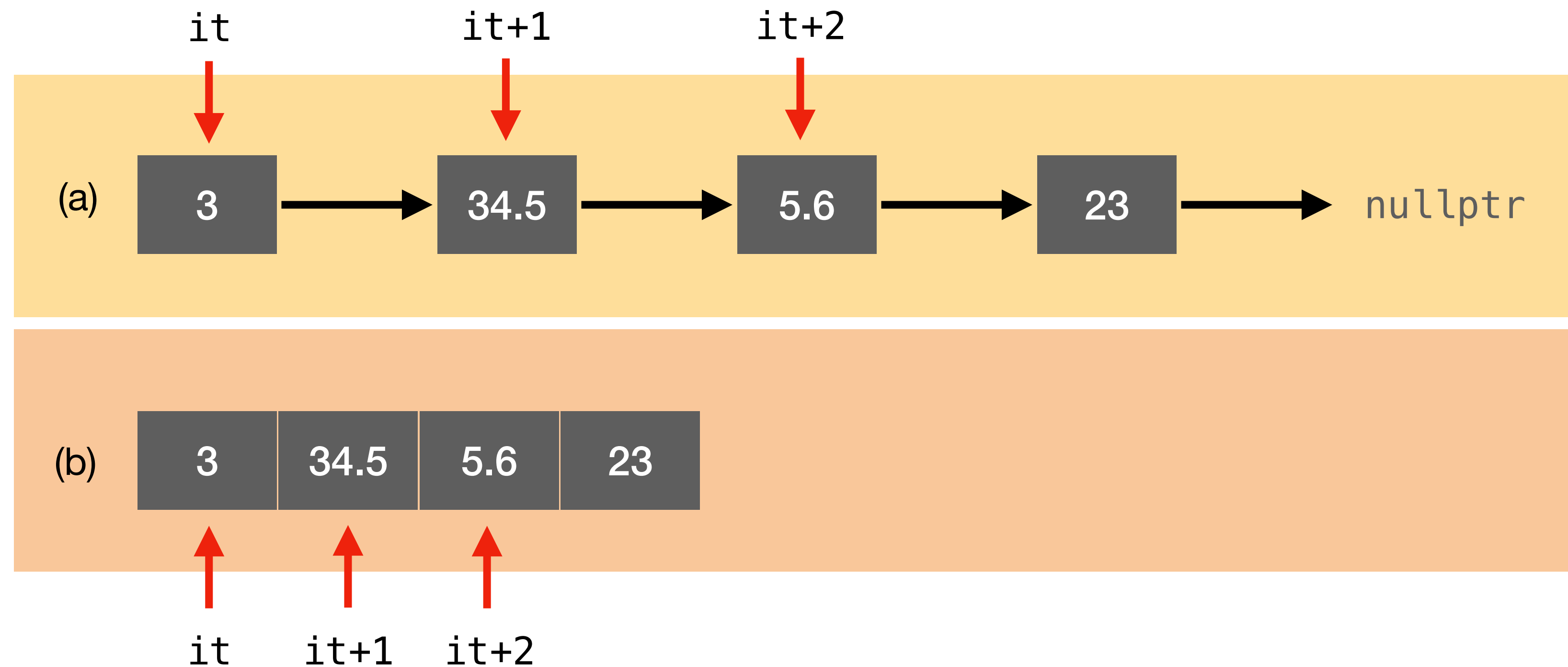
- Two different implementations of a `list<Val>` container: implementation (a) uses a **singly-linked list**; implementation (b) uses an **array**.
- The idea is to make this difference **invisible** to clients of our container via iterators.
- All that we need is that both implementations offer the same **iterator interface**.

Iterators in C++ — Example



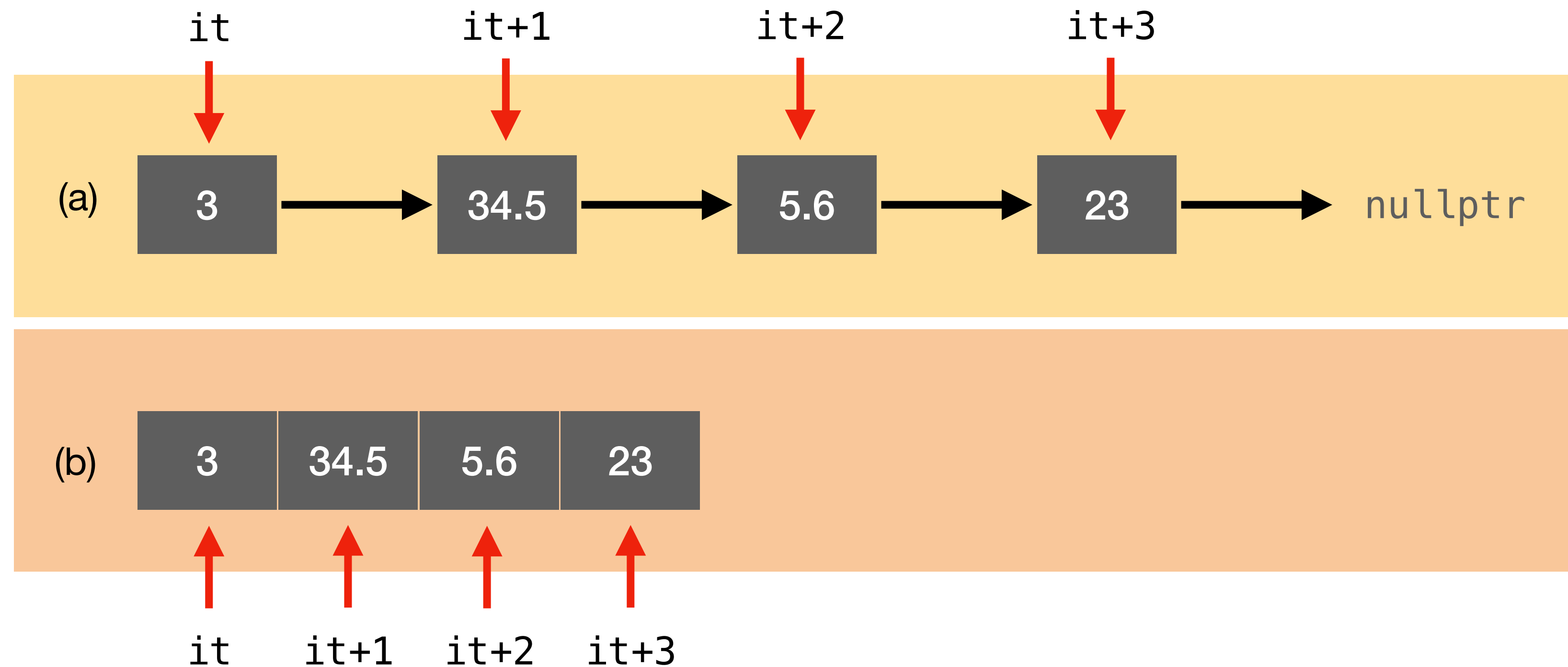
- Two different implementations of a `list<Val>` container: implementation (a) uses a **singly-linked list**; implementation (b) uses an **array**.
- The idea is to make this difference **invisible** to clients of our container via iterators.
- All that we need is that both implementations offer the same **iterator interface**.

Iterators in C++ — Example



- Two different implementations of a `list<Val>` container: implementation (a) uses a **singly-linked list**; implementation (b) uses an **array**.
- The idea is to make this difference **invisible** to clients of our container via iterators.
- All that we need is that both implementations offer the same **iterator interface**.

Iterators in C++ — Example



- Two different implementations of a `list<Val>` container: implementation (a) uses a **singly-linked list**; implementation (b) uses an **array**.
- The idea is to make this difference **invisible** to clients of our container via iterators.
- All that we need is that both implementations offer the same **iterator interface**.

Iterator interface

- Using the iterator interface (that we are about to implement), we can write the following snippet of code:

```
list<double> x;  
(...) // code that fills x  
for (iterator it = x.begin(); it != x.end(); ++it) {  
    ... std::cout << *it << '\n';  
}
```

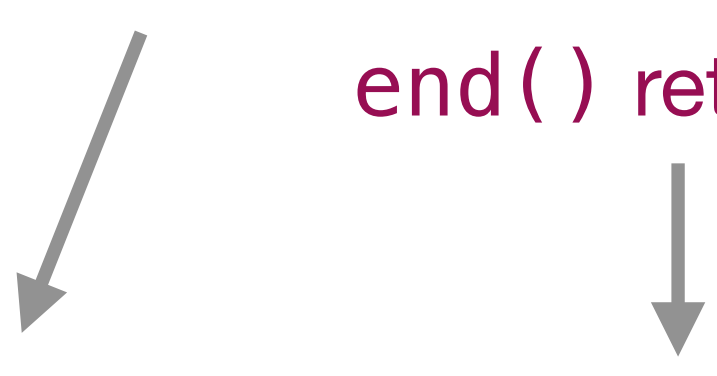
Iterator interface

- Using the iterator interface (that we are about to implement), we can write the following snippet of code:

```
list<double> x;  
(...) // code that fills x  
for (iterator it = x.begin(); it != x.end(); ++it) {  
    ... std::cout << *it << '\n';  
}
```

`begin()` returns an iterator pointing to the first element

`end()` returns an iterator pointing to **one-past** the last element



Iterator interface

- Using the iterator interface (that we are about to implement), we can write the following snippet of code:

```
list<double> x;  
(...) // code that fills x  
for (iterator it = x.begin(); it != x.end(); ++it) {  
    ... std::cout << *it << '\n';  
}
```

`begin()` returns an iterator pointing to the first element

`end()` returns an iterator pointing to **one-past** the last element

we can increment the iterator to point to the next element

Iterator interface

- Using the iterator interface (that we are about to implement), we can write the following snippet of code:

```
list<double> x;  
(...) // code that fills x  
for (iterator it = x.begin(); it != x.end(); ++it) {  
    ... std::cout << *it << '\n';  
}
```

`begin()` returns an iterator pointing to the first element

`end()` returns an iterator pointing to **one-past** the last element

we can increment the iterator to point to the next element

we can dereference the iterator to read/write the element

Iterator interface

- Using the iterator interface (that we are about to implement), we can write the following snippet of code:

```
list<double> x;  
(...) // code that fills x  
for (iterator it = x.begin(); it != x.end(); ++it) {  
    ... std::cout << *it << '\n';  
}
```

begin() returns an iterator pointing to the first element

end() returns an iterator pointing to **one-past** the last element

we can dereference the iterator to read/write the element

we can increment the iterator to point to the next element

- As if it were a pointer! But *without* knowing *how* the list container is actually implemented.

Iterator interface

- Using the iterator interface (that we are about to implement), we can write the following snippet of code, which is equivalent to the previous one:

```
list<double> x;  
(...) // code that fills x  
for (double val : x) { ← this is called a “range-based” for loop  
    ... std::cout << val << '\n';  
}
```

Iterator interface

- Using the iterator interface (that we are about to implement), we can write the following snippet of code:

```
list<double> x;  
(...) // code that fills x  
iterator it = x.begin();  
iterator end = x.end();  
*it = 3; ++it;  
*it = 5; ++it;  
if (it == end) std::cout << "no more elements!" << std::endl;
```


Iterator interface

- Using the iterator interface (that we are about to implement), we can write the following snippet of code:

```
list<double> x;  
(...) // code that fills x  
iterator it = x.begin();  
iterator end = x.end();  
*it = 3; ++it; ← modify the first element and advance to next element  
*it = 5; ++it;  
if (it == end) std::cout << "no more elements!" << std::endl;
```



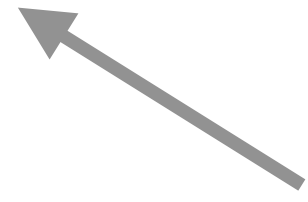
Iterator interface

- Using the iterator interface (that we are about to implement), we can write the following snippet of code:

```
list<double> x;  
(...) // code that fills x  
iterator it = x.begin();  
iterator end = x.end();  
*it = 3; ++it; ← modify the first element and advance to next element  
*it = 5; ++it; ← modify the second element and advance to next element  
if (it == end) std::cout << "no more elements!" << std::endl;
```

Iterator interface

- Using the iterator interface (that we are about to implement), we can write the following snippet of code:

```
list<double> x;  
(...) // code that fills x  
iterator it = x.begin();  
iterator end = x.end();  
*it = 3; ++it;   
*it = 5; ++it;   
if (it == end) std::cout << "no more elements!" << std::endl;  

```

modify the first element and advance to next element

modify the second element and advance to next element

test if we have reached the end of the container

We can use STL algorithms with iterators

- The standard template library (STL) of C++ implements a wide range of algorithms working on containers. We will better see this when we will talk about the STL.
- Again: **the algorithm does not have to know the implementation details of your container**; it is sufficient that it has the capability to read/write the elements in a container.
- This is exactly the purpose of iterators.

We can use STL algorithms with iterators

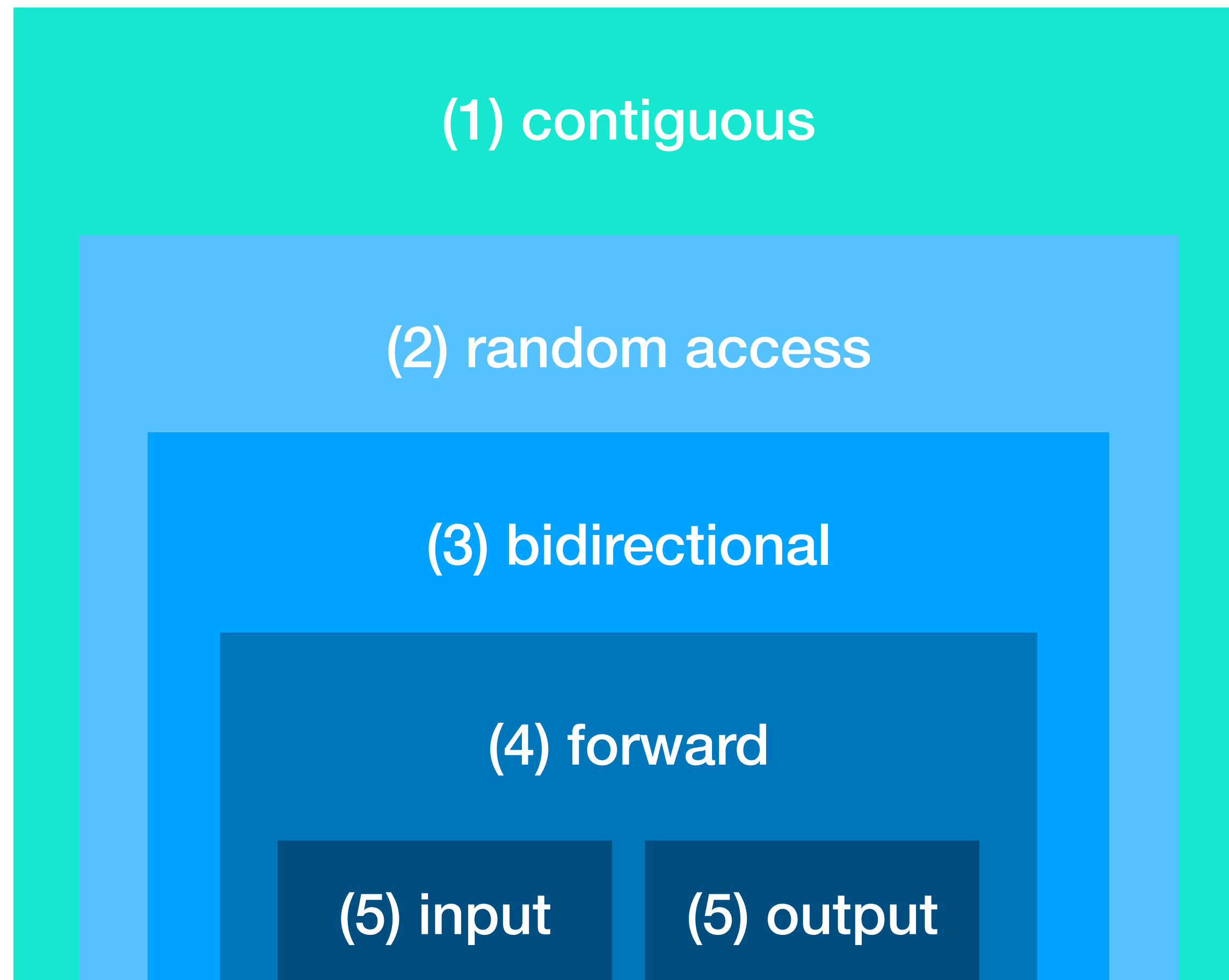
- The standard template library (STL) of C++ implements a wide range of algorithms working on containers. We will better see this when we will talk about the STL.
- Again: **the algorithm does not have to know the implementation details of your container**; it is sufficient that it has the capability to read/write the elements in a container.
- This is exactly the purpose of iterators.
- Examples.

```
std::vector<int> v{3, 6, 1, -34, 4, 1};  
std::sort(v.begin(), v.end());  
  
list<float> l;  
l += 0; l += 1; l += 3;  
std::fill(l.begin(), l.end(), 3.14);
```

Iterator types

- C++ defines **six** types of iterators:
 - **Input** iterator — it can scan the container (in forward direction) only once and cannot modify the element it points to;
 - **Output** iterator — it can scan the container (in forward direction) only once and cannot read the element it points to;
 - **Forward** iterator — it can scan the container forward as many times as we want and can read/write the element it points to;
 - **Bidirectional** iterator — same as previous one but it can scan in both forward and backward direction;
 - **Random access** iterator — same as previous one but can also jump to a middle position (access is not necessarily sequential);
 - **Contiguous** iterator — same as previous one but with the property that logically adjacent elements are also physically adjacent in memory.

Iterator types — Hierarchy



- Each iterator type in category i is also an iterator of type in category $i + 1$.
- For example, a bidirectional iterator is also a forward and input/output iterator.
- Or, a random access iterator is also bidirectional, forward, input/output.

The iterator class

iterator class — Definition

```
template<typename Val>
struct list {
    ... struct iterator {
    ... (... )
    ... };
    (... )
};
```



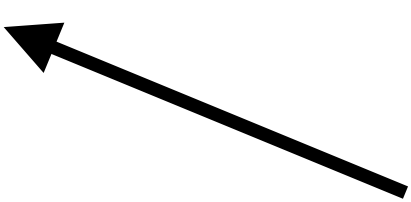
- The class `iterator` is a public class implemented **inside** the container because the idea is that we create an iterator over the element of a specific container.
- We do not have a single `iterator` class working for any container; each container has to implement its own.

iterator class — Properties

```
template <typename Val>
struct list {
    struct iterator {
        using iterator_category = std::forward_iterator_tag;
        using value_type = Val;
        using pointer = Val*;
        using reference = Val&;

        (...)
    };

    (...)
};
```

- 
- The C++ language requires us to define the following properties: the type of the iterator (or “category”), the value of the element pointed to, etc.
 - This is required by the language to make it possible to use STL’s algorithms on our container.
 - **Note.** The keyword `using` and `typedef` do the same task.

iterator class — Constructor

```
template <typename Val>
struct list {

    struct iterator {

        using iterator_category = std::forward_iterator_tag;
        using value_type = Val;
        using pointer = Val*;
        using reference = Val&;

        iterator(node* ptr);

        (...)

    private:
        node* m_ptr;

    };

    (...)

};
```

- Behind the scenes, our iterator's implementation will manipulate a pointer to a list's node.

iterator class — Methods

```
template <typename Val>
struct list {

    struct iterator {

        using iterator_category = std::forward_iterator_tag;
        using value_type = Val;
        using pointer = Val*;
        using reference = Val&;

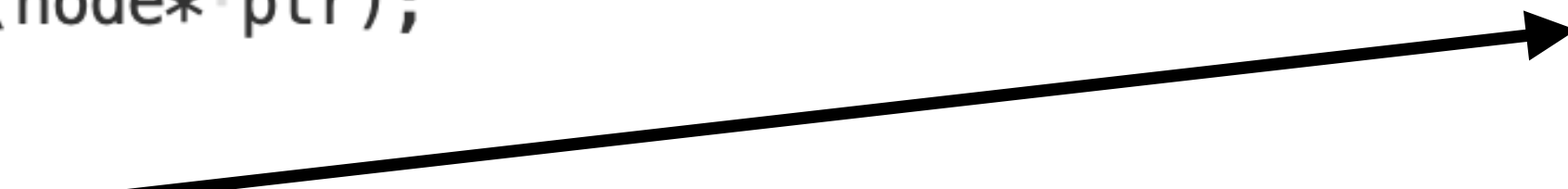
        iterator(node* ptr);

        (...)

    private:
        node* m_ptr;
    };

    (...)

};
```



```
        reference operator*() const;
        pointer operator->() const;
        iterator& operator++();
        iterator operator++(int /* dummy */);
        bool operator==(iterator const& rhs) const;
        bool operator!=(iterator const& rhs) const;
```

We are almost done!

```
template <typename Val>
struct list {
    struct iterator { ... };

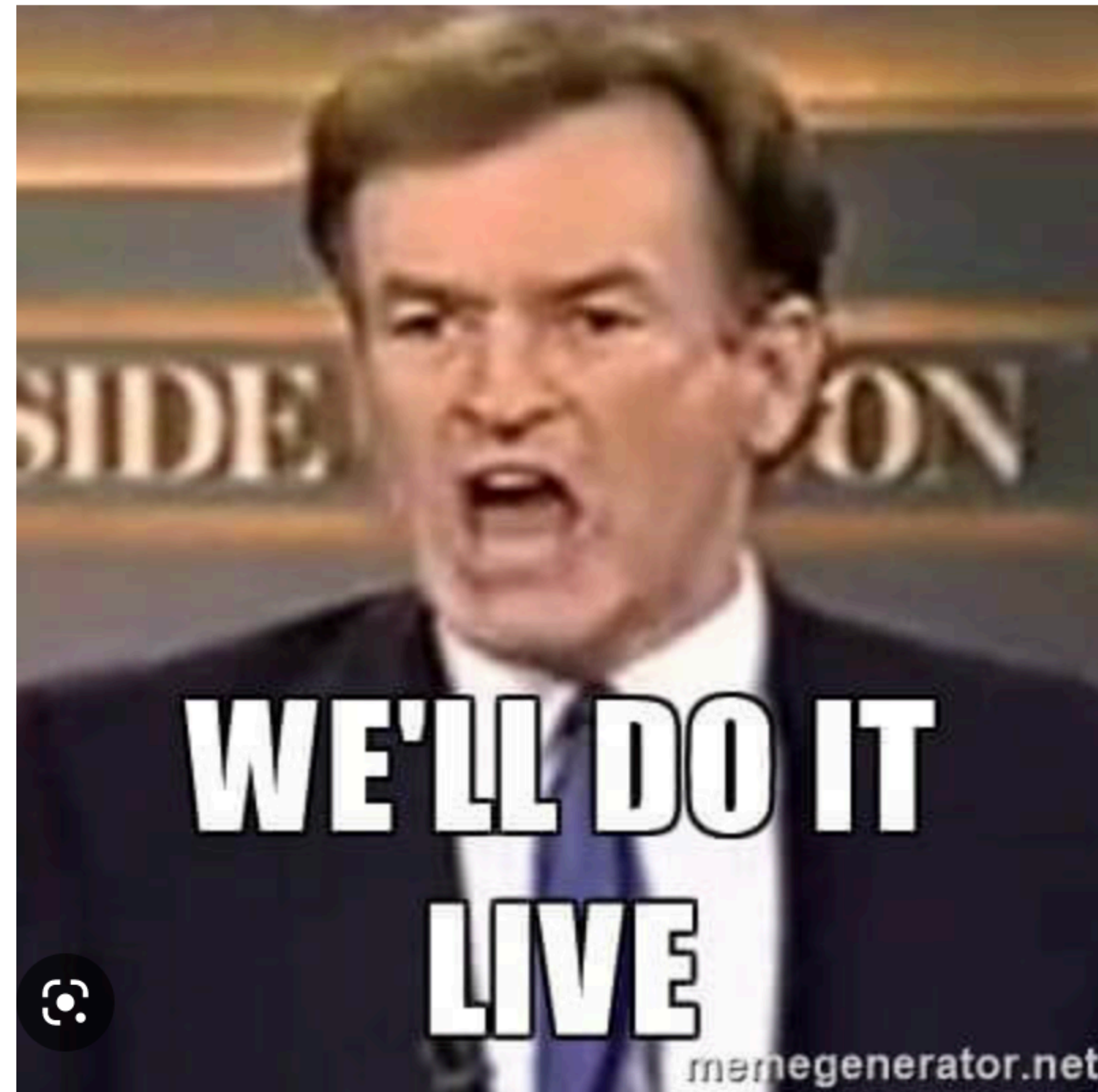
    (...)

    iterator begin();
    iterator end();

private:
    (...)
};
```

- The last missing step is to define the methods `begin()` and `end()` in our `list<Val>` container that gives the ability to create an iterator to the first element and to (*one-past*) the last element.

Let's now implement the iterator class for `list<Val>`



The `const_iterator` class

The `const_iterator` class

- The `iterator` class we have implemented can be used for both reading and writing the elements of our container.
- But what happens if our container cannot be modified? For example, consider these functions.

```
int process(list<double> const& l) {  
    ...  
}
```

```
template<typename Val>  
bool list<Val>::operator==(list<Val> const& rhs) const {  
    ...  
}
```


The `const_iterator` class

- The `iterator` class we have implemented can be used for both reading and writing the elements of our container.
- But what happens if our container cannot be modified? For example, consider these functions.

```
int process(list<double> const& l) {  
    ...  
}
```

```
template<typename Val>  
bool list<Val>::operator==(list<Val> const& rhs) const {  
    ...  
}
```

- We need a `const_iterator` class, an iterator that can only read the elements without modifying them.
- **Note.** `const_iterator` and `const` iterator are not the same!

The const_iterator class

```
... struct const_iterator {  
...     using iterator_category = std::forward_iterator_tag;  
...     using value_type = const Val;  
...     using pointer = Val const*;  
...     using reference = Val const&;  
  
...     const_iterator(node* ptr);  
  
...     reference operator*() const;  
...     pointer operator->() const;  
...     const_iterator& operator++();  
...     const_iterator operator++(int /* dummy */);  
  
...     bool operator==(const_iterator const& rhs) const;  
...     bool operator!=(const_iterator const& rhs) const;  
  
... private:  
...     node const* m_ptr;  
... };
```

- Essentially the same as iterator but note the const requirement!

And lastly...

```
template <typename Val>
struct list {

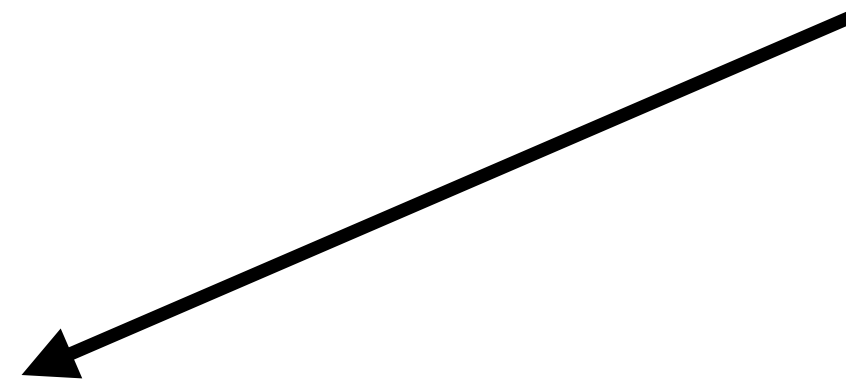
    ... struct iterator { ... };
    ... struct const_iterator { ... };

    ... ( ... )

    ... iterator begin();
    ... iterator end();
    ... const_iterator begin() const;
    ... const_iterator end() const;

private:
    ... ( ... )
};
```

- The use of the `const` keyword here instructs the compiler to call the proper version of `begin()` or `end()` based on the “**constness**” of the context.



Example of a const context

```
/* is this container equal to rhs? */
template <typename Val>
bool list<Val>::operator==(list<Val> const& rhs) const {
    ... auto ptr1 = front();
    ... auto ptr2 = rhs.front();
    ... while (ptr1 or ptr2) {
        ... if ((!ptr1 and ptr2) or (!ptr2 and ptr1)
        ...     or (ptr1->val != ptr2->val))
        ... {
        ...     return false;
        ... }
        ... ptr1 = ptr1->next;
        ... ptr2 = ptr2->next;
    ... }
    ... return true;
}
```

- Let's re-write it with iterators!

Example of a const context

```
/* is this container equal to rhs? */
template <typename Val>
bool list<Val>::operator==(list<Val> const& rhs) const {
    auto it1 = begin();
    auto it2 = rhs.begin();
    while (it1 != end() or it2 != rhs.end()) {
        if ((it1 == end() and it2 != rhs.end()) or
            (it1 != end() and it2 == rhs.end()) or (*it1 != *it2)) {
            return false;
        }
        it1++;
        it2++;
    }
    return true;
}
```

← here, both it1 and it2 are
of type const_iterator

Invalidation

- Iterators are objects created by a specific container that point to the elements in the container.
- A reasonable question is, therefore: what happens to an iterator created by a container if the memory for the container changes?
- The memory held by a container can change, for example, when:
 - inserting/removing elements (after a memory re-allocation);
 - the container is move assigned;
 - the container is destroyed;
 - etc.

Invalidation

- Iterators are objects created by a specific container that point to the elements in the container.
- A reasonable question is, therefore: what happens to an iterator created by a container if the memory for the container changes?
- The memory held by a container can change, for example, when:
 - inserting/removing elements (after a memory re-allocation);
 - the container is move assigned;
 - the container is destroyed;
 - etc.
- Intuitively, the iterator will then point to “garbage”. It is, therefore, considered **invalid**.

Invalidation

- There is no universal rule for iterator invalidation.
- In general, **an iterator is invalid if the memory for the container it was created by has changed.**
- But this depends on the specific data structure used to implement the container!
- Knowing when an iterator is in a valid state or not, requires knowledge of the container implementation.

Invalidation

- There is no universal rule for iterator invalidation.
- In general, **an iterator is invalid if the memory for the container it was created by has changed.**
- But this depends on the specific data structure used to implement the container!
- Knowing when an iterator is in a valid state or not, requires knowledge of the container implementation.
- Therefore, it is good practice to **avoid mixing iteration with operations that can cause memory re-/de-allocation.**
- Let's consider some examples.

Invalidation — Example

```
std::vector<int> v{3, 6, 1, -34, 4, 1};  
auto it_v = v.begin();  
v.insert(it_v, 5);  
std::cout << *it_v << std::endl; // ouch!
```

- A memory re-allocation happens when inserting 5 at the beginning of the vector v. Hence, the iterator `it_v` is invalid after the `insert`.

Invalidation — Example

```
... list<float> l;  
... ((l += 0) += 1) += 3;  
... auto it_l = l.begin();  
... l.push_front(-1);  
... l.push_back(4);  
... for (; it_l != l.end(); ++it_l) std::cout << *it_l << ' ';  
... std::cout << std::endl;
```

- **Q.** Does this work? Is `it_l` invalid or not after `push_front/push_back`?

Invalidation — Example

```
... list<float> l, g;  
... l += 0;  
... l += 1;  
... l += 2;  
... g += 3;  
... g += 4;  
... auto it_l = l.begin();  
... l = std::move(g);  
... for (; it_l != l.end(); ++it_l) std::cout << *it_l << ' '; // ouch!  
... std::cout << std::endl;
```

- After g is moved into l, the memory of l has changed!