Software Languages.Lab

Structuur van Computerprogramma's 2

dr. Dirk Deridder
Dirk.Deridder@vub.ac.be
http://soft.vub.ac.be/

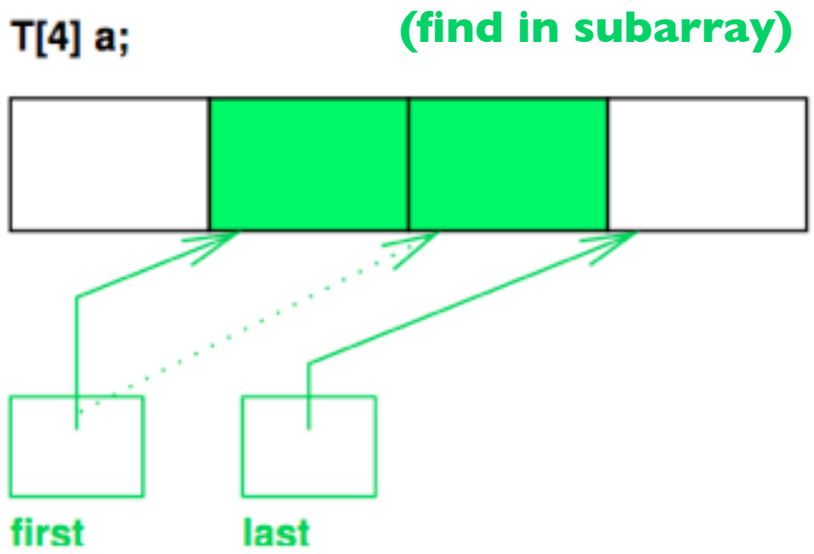
Vrije Universiteit Brussel - Faculty of Science and Bio-Engineering Sciences - Computer Science Department - Pleinlaan 2 - 1050 Brussels - Belgium

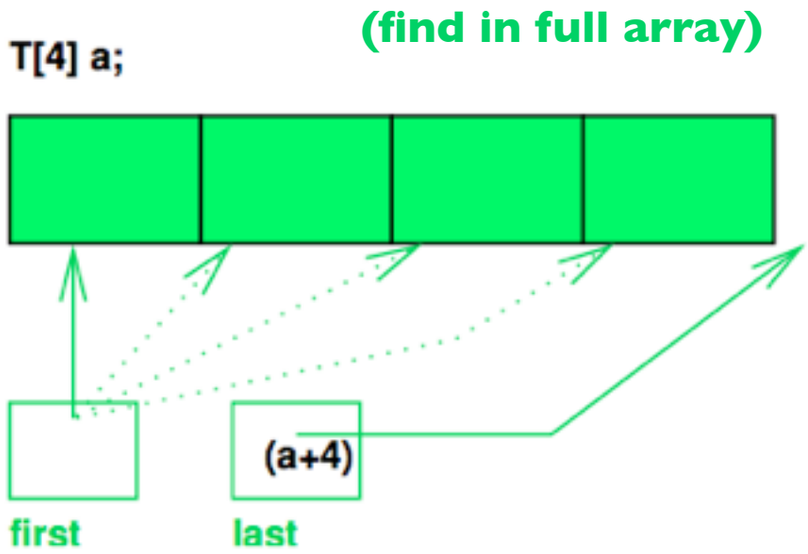
Chapter 6 - Generic Programming using the STL

Structuur van Computerprogramma's 2

Motivation

Sequential Search in an Array

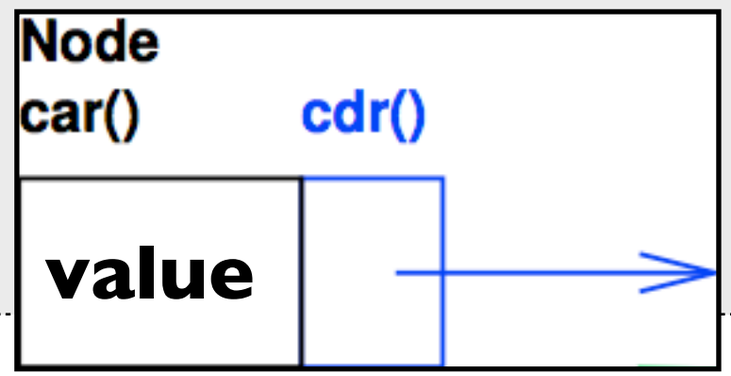




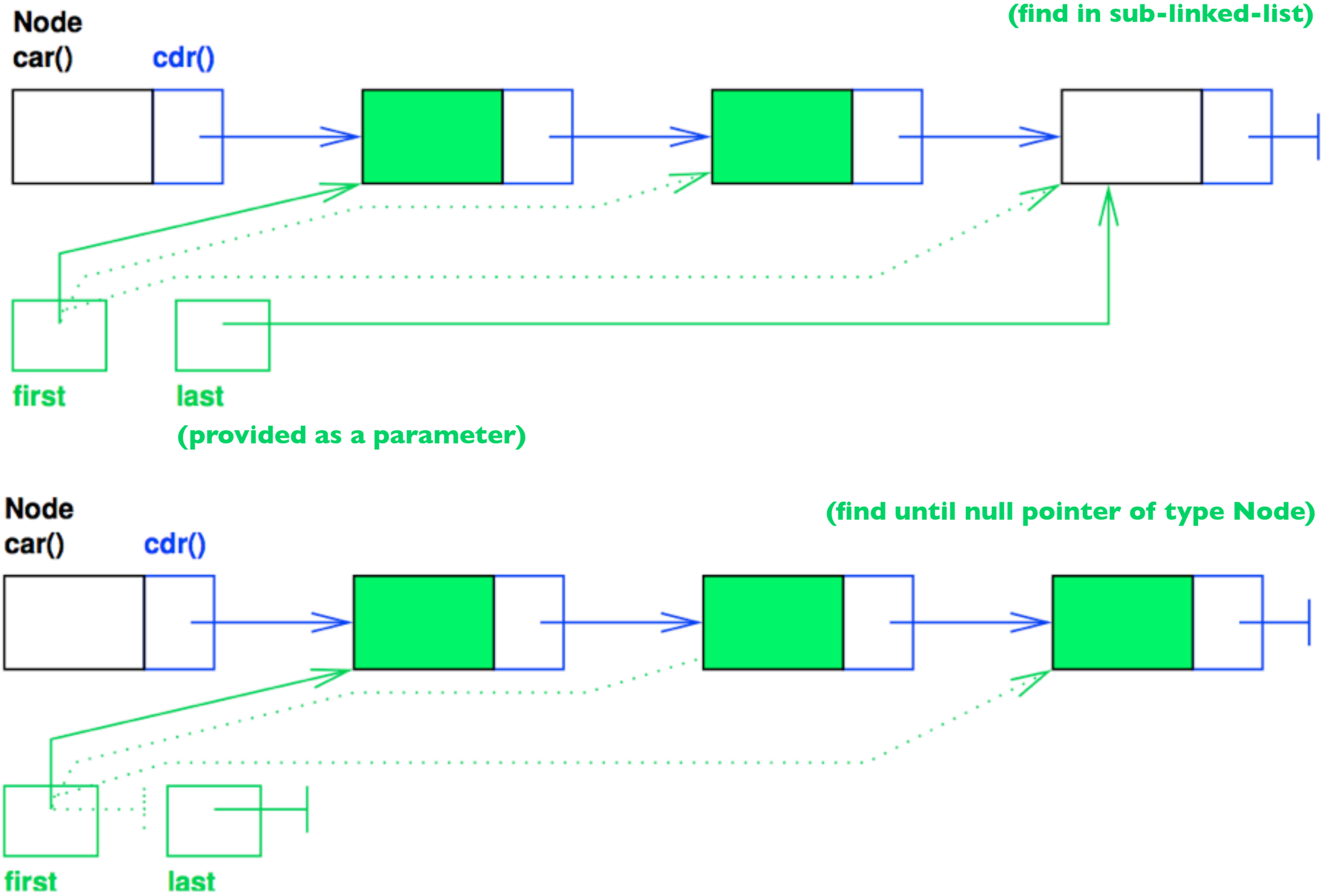

```
// sequential search for value in [*first..*last[;
// return last if not found
template<typename T>
const T*
find(const T* first, const T* last, const T& value) {
   while (first != last && *first != value)
      ++first;
   return first;
}

extern int a[SIZE];

// find ’20’ in array a
find(a, a+SIZE, 20); // return pointer to 20 or a+SIZE
```

Sequential Search in a Linked List

```
// Node<T>* is a (too) simple linked list of T
template<typename T>
class Node {
public:
// constructor
   Node(const T& val, Node* cdr = 0) : value_(val), cdr_(cdr) { }

// “cons” creates a node (value, next) as in Scheme (i.e. prepend)
   Node* cons(const T& val) { return new Node(val, this); }

// “car” returns first slot of a node (value)
   const T& car() const { return value_; }

// “cdr” returns second slot of a node (next)
   Node* cdr() const { return cdr_; }

private:
   T value_;
   Node* cdr_;
};
```



Sequential Search in a Linked List



Sequential Search in a Linked List

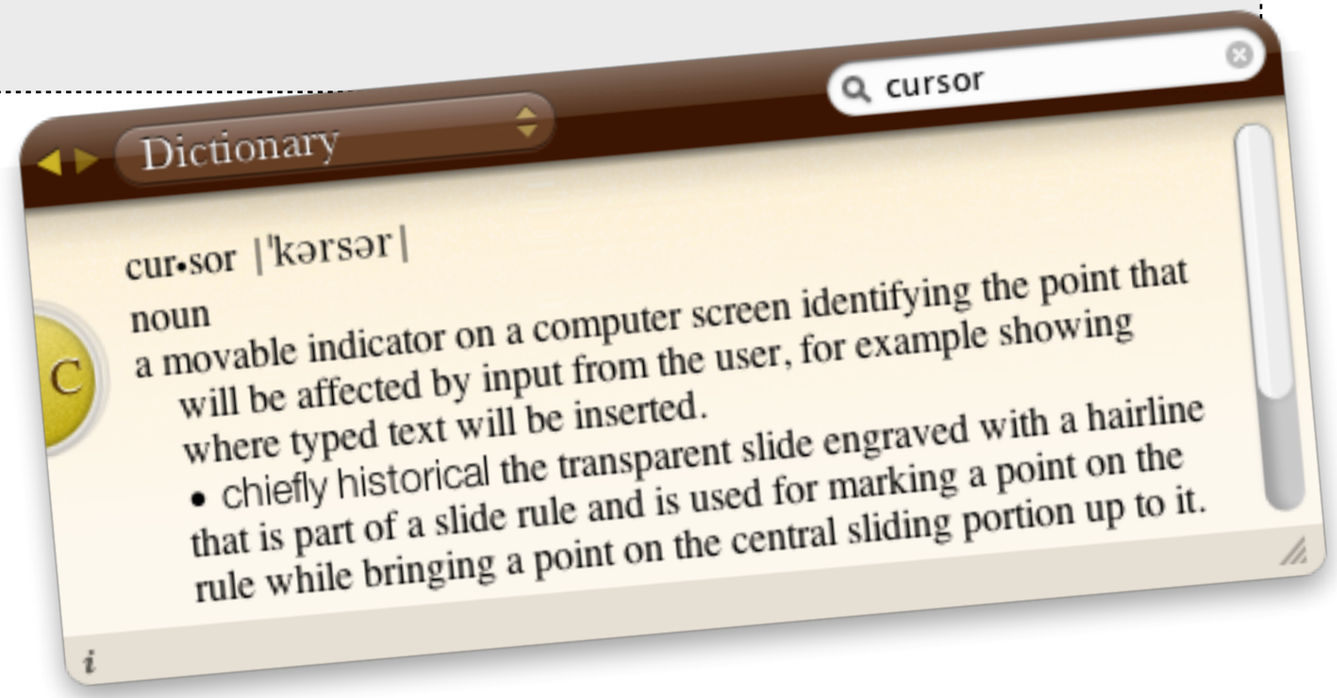
```
template<typename T>
Node<T>* // search for value in linked list [first .. last]
find(Node<T>* first, Node<T>* last, const T& value) {
   while (first != last && first->car() != value)
      first = first->cdr();
   return first;
}

extern Node<int> *list; // a pointer to a Node with int values

// find ’20’ in linked list from start until null pointer
find(list, static_cast<Node<int>*>(0), 20);
```

The Essence of Sequential Search in a Container

Advance a **Cursor** in the **Container** until either

- the value is found, or
- all elements of the container have been checked

```
Cursor
find(Cursor start, Cursor end, T value) {
   while ( start!=end &&
           (object_pointed_to_by_start != value) )
      advance start to next position in container;
   return start;
}
```



Requirements on Cursor

```
class Cursor { // + constructors still to be added
// ...
public:
// DEREFERENCE cursor to obtain an object
// from the container
   T operator*();
// RETURN cursor that refers to the NEXT
// object in the container
   Cursor operator++();
// COMPARE two cursors
   bool operator!=(Cursor);
// ASSIGN a cursor to another one (not really required by find)
   Cursor& operator=(const Cursor&);
};

// sequential search
Cursor find(Cursor start, Cursor end, T value) {
   while (start != end && (*start != value))
      ++start;
   return start;
}
```

A Generic find Function

A **generic algorithm** is "pure", i.e. independent of the data types on which it operates

```
template <typename InputIterator, typename T>
InputIterator
find(InputIterator first, InputIterator last, const T& value)
   while (first != last && *first != value)
      ++first;
   return first;
}
```

This works

- for arrays, **`InputIterator`** is **`T*`**
- for linked lists, we must implement a **`Node<T>::Cursor`** class that implements the requirements

Linked List Iterator

```
template<typename T>
class Node {
public: // ...
   // Cursor that satisfies find requirements
   class Cursor {
   public:
      Cursor(Node* node = 0) : node_(node) { }
   // Dereference
      const T& operator*() const { return node_->car(); }
   // Return next
      Cursor& operator++() {
         node_ = node_->cdr();
         return *this;
      }
   // Compare
      bool operator!=(const Cursor& c) const { return node_ != c.node_; }
   private:
      Node* node_;
   };
};
```

Linked List Iterator Example

```
extern Node<int> *list;
find(Node<int>::Cursor(list), Node<int>::Cursor(), 20); // start, end, value
```

Improve the 'readability' by defining:

```
template<typename T>
class Node {
public:
   // ...
   class Cursor {
      // ...
   };
   // return Cursor pointing to the start of the list
   Cursor begin() {
      return Cursor(this);
   }
   // return Cursor pointing past the end of the list
   Cursor end() {
      return Cursor();
   }
};
```

then `find(list->begin(), list->end(), 20)` also works

Containers and Algorithms

The STL provides many **Container Class Templates**:

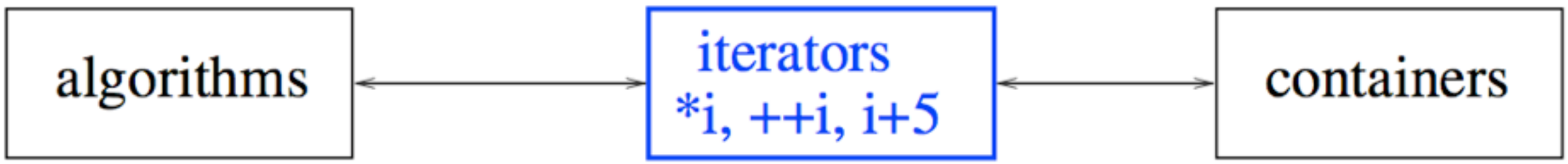
- Sequences: vector, list, dequeue
- Associative: set, multiset, map, multimap, hash table
- Adapters: stack, queue, priority queue

It also supports more than 50 algorithms that use these containers:

- Non-mutating sequence operations: e.g. find, find_if, for_each, ...
- Mutating sequence operations: e.g. copy, replace, erase, ...
- Sorting operations
- ...

Why Iterators?

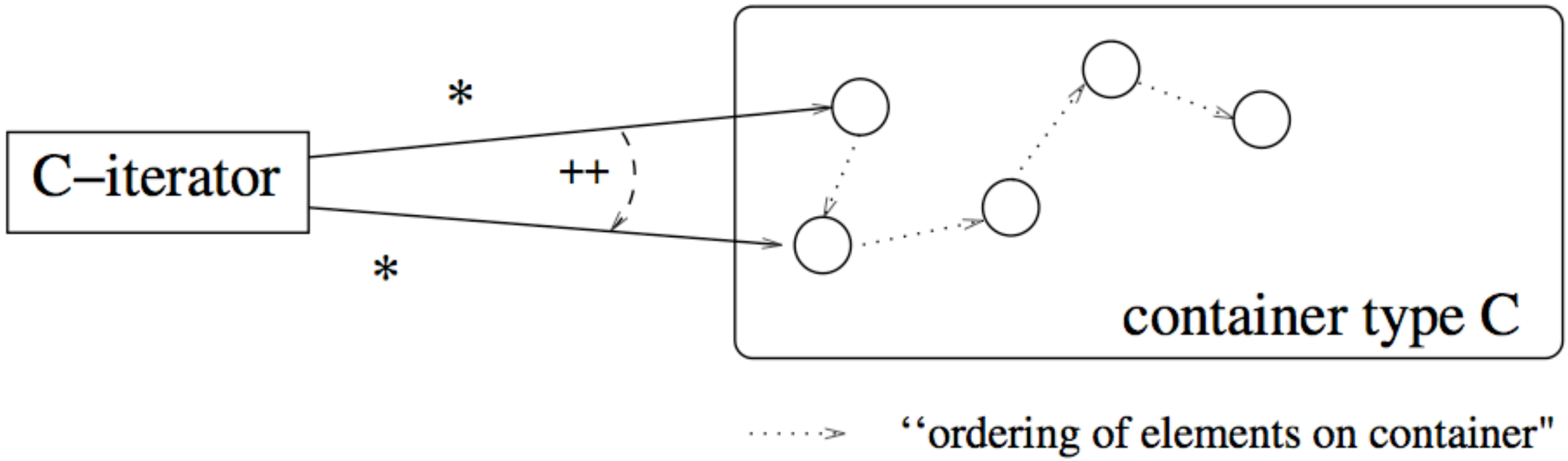
- The goal is to write algorithms that work for many containers
- To implement each algorithm for each container would require hundreds of implementations
- This is possible by putting a new abstraction between algorithms and containers (data structures) : **iterators**



An iterator is like an **abstract pointer**: it may support

- dereference (using `operator*()`): `*it`
- increment, decrement: `++it`, `--it`, `it++`, `it--`
- random access: `it+5`

A Container and its Iterator




```
template<typename Element>
class C {
   CIterator begin(); // "pointer" to first element in C
   CIterator end();   // "pointer" beyond last element in C
};

template<typename Element>
class CIterator {
public:
   Element operator*(); // dereference to get value of an element
   CIterator operator++(); // increment to get next container element
};
```

A Simple Algorithm using Containers

```
// works for any iterator on any container
template<typename Iterator, typename T>
// return iterator pointing to cell containing value or last
Iterator
find(Iterator first, Iterator last, const T& value) {
   while (first != last && *first != value)
      ++first;
   return first;
}

C<std::string> box;
std::string s("abc");
CIterator it;

if ( (it=find(box.begin(), box.end(), s)) == box.end())
   cout << s << " not found" << endl;
else
   cout << s << " found: " << *it << endl;
```