The Standard Template Library

Alessandro SAVINO
Politecnico di Torino (Italy)

alessandro.savino@polito.it

www.testgroup.polito.it

License Information

Disclaimer

- **We disclaim any warranties or representations as to the accuracy or completeness of this material.**
- **Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.**
- **Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.**

Goal

- **This lecture presents a global overview of the basic concepts behind the C++ Template Library.**

Prerequisites

- **Basic knowledge of C++ programming language**

Homework

- **None**

Outline

- **The Standard Template Library**
- **The STL's Components**
- **Containers definition**
- **STL Containers**
- **STL Containers: The Vector**
- **STL Iterators**
- **STL Algorithms**

Outline

- The Standard Template Library
- The STL's Components
- Containers definition
- STL Containers
- STL Containers: The Vector
- STL Iterators
- STL Algorithms

The Standard Template Library

- **The STL is a complex piece of software engineering that uses some of C++'s most sophisticated features**
- **STL provides an incredible amount of generic programming power**
- **It provides general purpose, “templatized” classes and functions**

The STL

- Designed by Alex Stepanov
- General aim: the most general, most efficient, most flexible representation of concepts (algorithms, containers, etc.)
- Works for any data type:
  - integers, floating-point numbers, polynomials, ...

The STL

- Deals with organization of code and data
  - Built-in types, user-defined types, and data structures
- Optimizing disk access was among its original uses
  - Performance was always a key concern

Rel. 24/04/2016

Outline

- The Standard Template Library
- The STL's Components
- Containers definition
- STL Containers
- STL Containers: The Vector
- STL Iterators
- STL Algorithms

The STL's Components

- **Container Classes**
- **Generic Algorithms**
- **Iterators**
- **Function Objects**
- **Allocators**
- **Adaptors**

The STL's Components

- **Container Classes**
  - **generic multiple-data managers**
- **Generic Algorithms**
- **Iterators**
- **Function Objects**
- **Allocators**
- **Adaptors**

The STL's Components

- Container Classes
- Generic Algorithms
  - common use algorithms (~60 standard algorithms) without any data bound
    - searching (e.g. find())
    - sorting (e.g. sort())
    - mutating (e.g. transform())
    - numerical (e.g. accumulate())
- Iterators
- Function Objects
- Allocators
- Adaptors

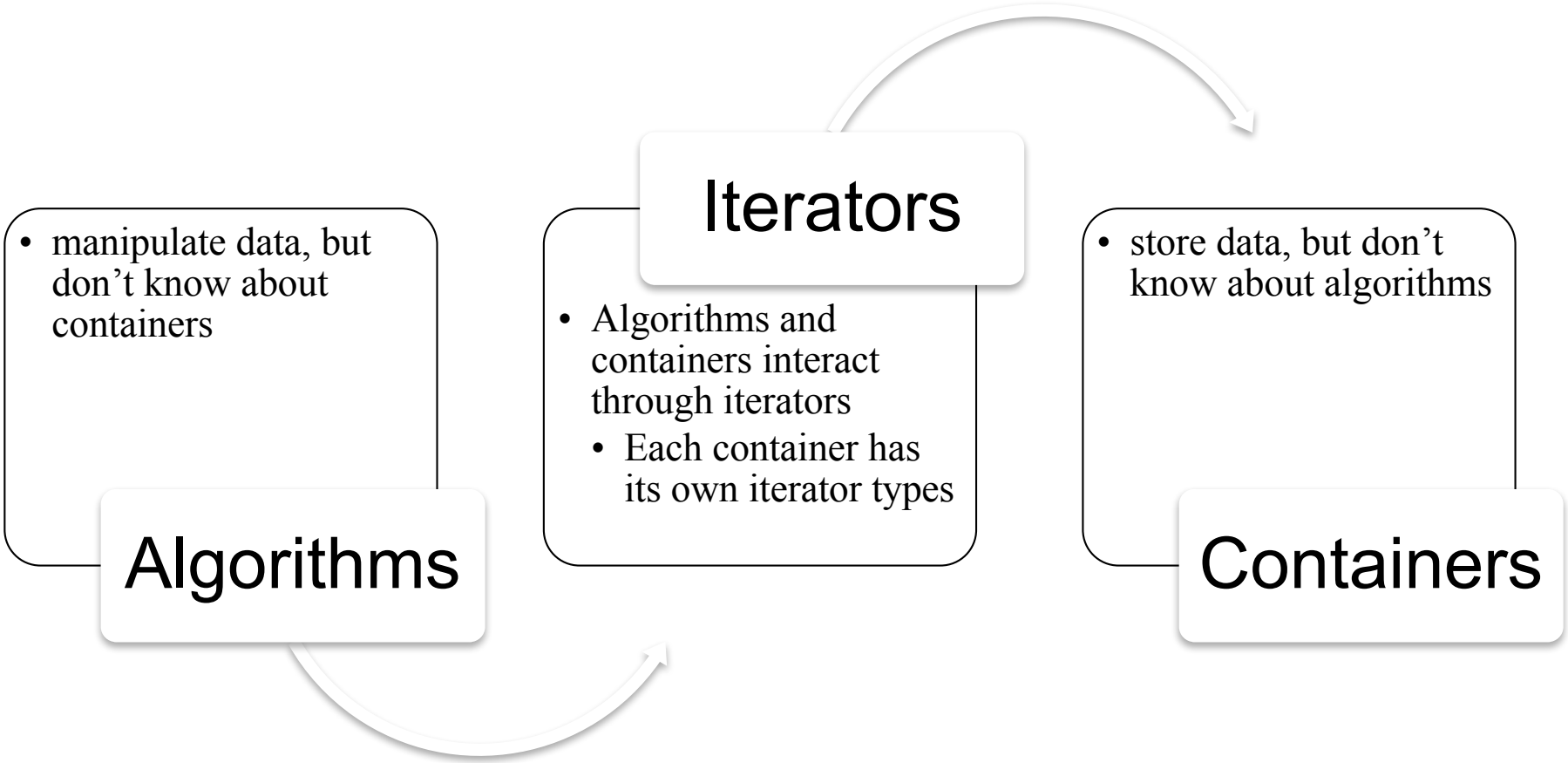
The STL's Components

- Container Classes
- Generic Algorithms
- Iterators
  - "container walkers" for accessing container elements
  - Iterators provide an abstraction of container access, which in turn allows for generic algorithms
- Function Objects
- Allocators
- Adaptors

The STL's Components

- Container Classes
- Generic Algorithms
- Iterators
- Function Objects
  - C++ objects that can be called like a function to implement "callbacks"
- Allocators
- Adaptors

The STL Basic Model



STL Advantages

- **Standardized**
- **Thin & efficient**
- **Little inheritance; no virtual functions**
- **Small**
- **Flexible and extensible**
- **Naturally open source**

STL Disadvantages

- **Template syntax**
- **Difficult to read & decipher**
- **Poor or incomplete compiler support**
- **Code bloat potential**
- **No constraints on template types**
- **Limited container types**

Outline

- The Standard Template Library
- The STL's Components
- Containers definition
- STL Containers
- STL Containers: The Vector
- STL Iterators
- STL Algorithms

Container Definition

- **A Container is an object that holds another object**
- **More powerful and flexible than arrays**
  - **It will grow or shrink dynamically and manage their own memory**
  - **It keeps track of how many objects they hold**
- **They belong to the STL Library.**

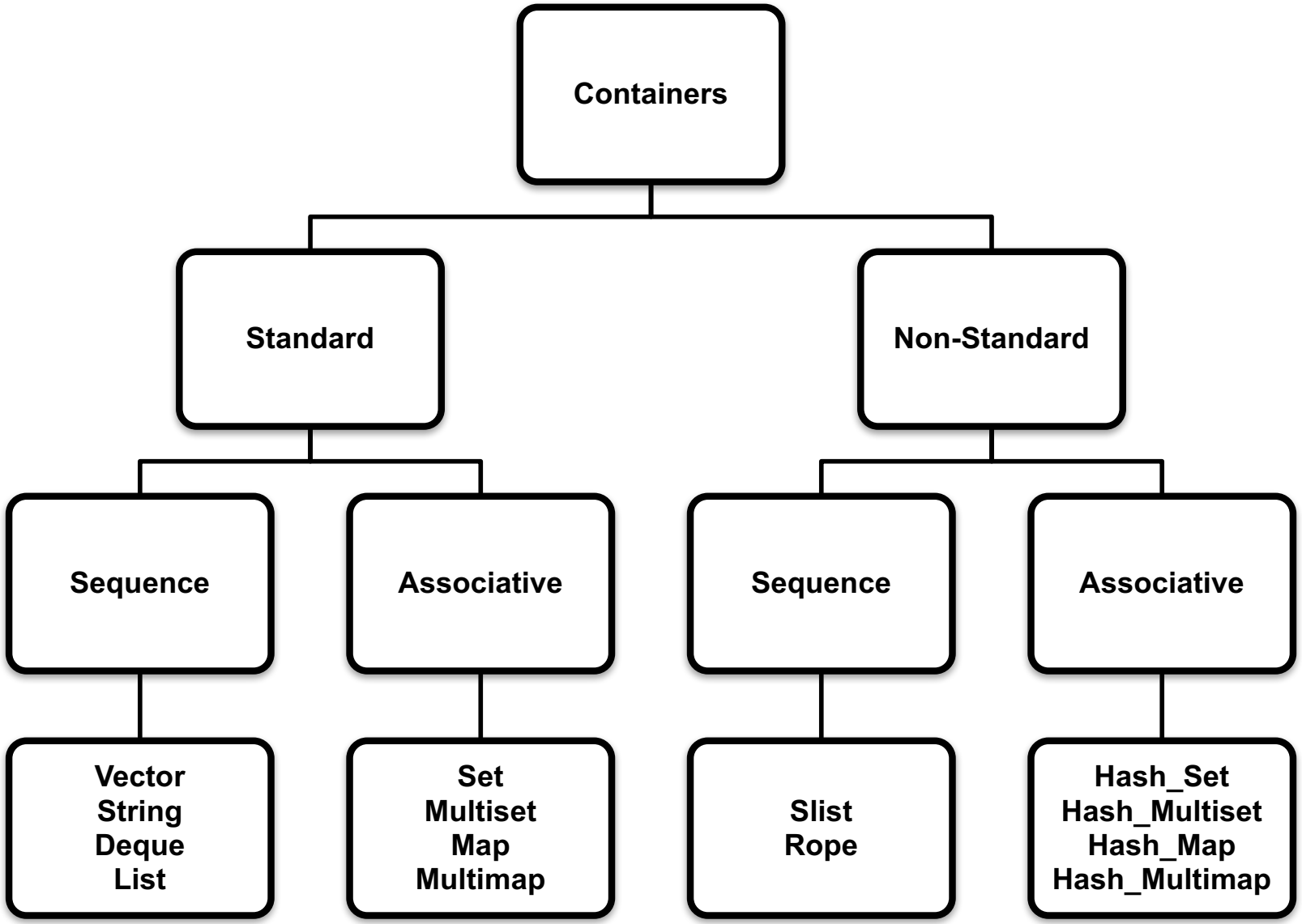
Container Class

- **A container class is capable of holding a collection of items.**
  - **you must care only about the operation you need for the actual application!**
- **In C++, container classes can be implemented as a class, along with member functions to add, remove, and examine items.**
  - **Containers provide iterators that point to its elements.**
  - **Containers provide a minimal set of operations for manipulating elements**

Outline

- The Standard Template Library
- The STL's Components
- Containers definition
- STL Containers
- STL Containers: The Vector
- STL Iterators
- STL Algorithms

Types of Container



STL Containers

- **Sequence Container**
  - **Stores data by position in linear order:**
    - **First element, second element , etc:**
- **Associate Container**
  - **Stores elements by key, such as name, social security number or part number**
  - **Access an element by its key which may bear no relationship to the location of the element in the container**

Outline

- **The Standard Template Library**
- **The STL's Components**
- **Containers definition**
- **STL Containers**
- **STL Containers: The Vector**
- **STL Iterators**
- **STL Algorithms**

Vector Container (I)

- **Generalized array that stores a collection of elements of the same data type**
  - **It is similar to an array**
    - **Vectors allow access to its elements by using an index in the range from 0 to n-1 where n is the size of the vector**
- **Vector has operations that allow the collection to grow and contract dynamically at the rear of the sequence**
  - **Do not need to specify quantity of elements**

Rel. 24/04/2016

Vector Container (II)

- **Index access ([] operator) like an array**
- **Vectors allow to add / remove an element in constant time at the last of vector**
- **Vectors allow to add / remove an element in linear time in the middle of vector**
  - **try to do that with arrays...**
- **It is the type of sequence container that should be used by default.**

Vector Basic Usage

- **Allows direct access to the elements via an index operator**
- **Indices for the vector elements are in the range from 0 to size() -1**

```
#include <vector>

int main() {
      vector <int> v(20);
      v[5]=15;
      cout << ”Vector Size: “ << v.size();
}
```

Vector Basic Usage

- **Allows direct access to the elements via an index operator**
- **Indices for the vector elements are in the range from 0 to size() -1**

```
#include <vector>

int main() {
      vector <int> v(20);
      v[5]=15;
      cout <                or Size: " << v.size();
```

This means ”generate a vector of 20 (int) items”

Rel. 24/04/2016

Vector Basic Usage

- **Allows direct access to the elements via an index operator**
- **Indices for the vector elements are in the range from 0 to size() -1**

```
#include <vector>

int main() {
      vector <int> v(20);
      v[5]=15;
```

If you want to set all items to a specific value, you must resort to another constructor:
vector<T> v(num, val);

Vector Basic Usage

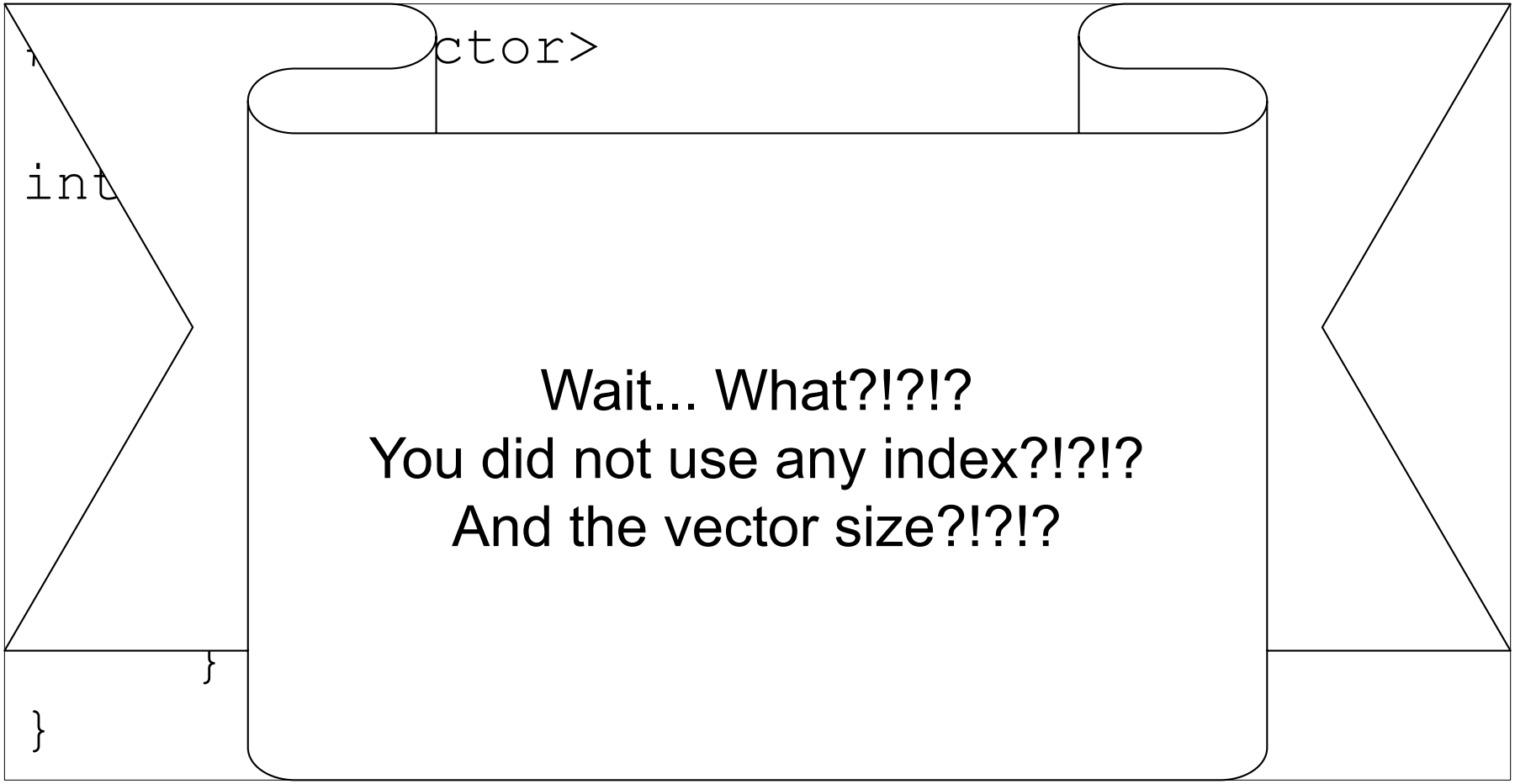
- **Despite array limitation, iterative fill of the vector is even more simple, using the proper method.**

```
#include <vector>

int main() {
      vector <int> v;
      int value;
      cin >> value;
      while (value > 0) {
            v.push_back(value);
            cin >> value;
      }
}
```


Vector Basic Usage

- **Despite array limitation, iterative fill of the vector is even more simple, using the proper method.**



Vector Basic Usage

- **Despite array limitation, iterative fill of the vector is even more simple, using the proper method.**

push_back method add the parameter to the end of v, increasing the size of v by 1.
It also increases the capacity of v if v was full before the new value was added.

```
        cin >>
        while (value      0) {
                v.push_back(value);
                cin >> value;
        }
}
```

Vector Advanced Usages

- **Due to its class nature, you can resort to the copy constructor to duplicate one vector into another one.**

```
#include <vector>

int main() {
     vector <int> v(20);
     ...
     vector <int> v2(v);
}
```

Vector Advanced Usages

- **It is also possible to implement even more operations:**
  - **clear the content (and set the size to 0)**

```
#include <vector>

int main() {
     vector <int> v(20);
     ...
     v.clear();
}
```

Vector Advanced Usages

- **It is also possible to implement even more operations:**
  - **resize it (if new size is lower it keeps current size, but take care of the data...)**

```
#include <vector>

int main() {
     vector <int> v(20);
     ...
     v.resize(23); // new size is 23
}
```

Vector Advanced Usages

- **It is also possible to implement even more operations:**
  - **resize it (if new size is lower it keeps current size, but take care of the data...)**

```
#include <vector>

int main() {
     vector <int> v(20);
     ...
     v.resize(19); // new size is 20
}
```

Vector Advanced Usages

- **It is also possible to implement even more operations:**
  - **resize it (if new size is lower it keeps current size, but take care of the data...)**

but index 19 could be not meaningful anymore!

```
    vector <int> v(20);
    ...
    v.resize(19); // new size is 20
}
```

Vector Advanced Usages

- **It is also possible to implement even more operations:**
  - **checking it's emptiness**

```
#include <vector>

int main() {
     vector <int> v(20);
     ...
     if (v.empty())
          cout << "Vector v is empty!";
}
```

Vector Advanced Usages

- **It is also possible to implement even more operations:**
  - **checking it's emptiness**

Remember: empty return a bool, it does not empty the vector!

```
        vect          v(20);
        ...
        if (v.empty())
              cout << "Vector v is empty!";
}
```

Vector Common Usage

- **Last but not least... since vector is a template class, you can define vectors of any type you need.**

```
#include <vector>
#include "Rectangle.hpp"

int main() {
     vector <int> v(20);
     vector < Rectangle <int> > v2;
}
```

Vector Common Usage

- **Last but not least... since vector is a template class, you can define vectors of any type you need.**

... also of template classes (if they are not containers...)

```
#include <vector>
#include "Rectangle.hpp"

int main() {
     vector <int> v(20);
     vector < Rectangle <int> > v2;
}
```

Vector vs Array

- **When dealing with arrays, several issue may reduce the flexibility in your coding.**
  - **Let's see some example starting from this very basic code**

```
int main() {
   int v[10];
   ...
   foo(v);
}
```

Vector vs Array

- **Passing arrays to functions**

```
int main() {
   int v[10];
   ...
   foo(v);
}
```

Is there a way to let foo now how many actual elements are stored in v without actually modify foo function?

Vector vs Array

- **Passing arrays to functions**

```
int main() {
   int v[10];
   ...
   foo(v, v_size);
}
```

No!
You must add a size_t actual_size variable to the prototype

void foo(int *vect);

⬇

void foo(int *vect, size_t actual_size);

Rel. 24/04/2016

Vector vs Array

- **Passing vector to functions**
  - **The container solution**

```
int main() {
   vector<int> v;
   ...
   foo(v);
}
```

void foo(vector<int> vect);

⬇

the actual size is retrieved "dynamically" by resorting to .size() container's public method.

Vector vs Array

- **Working on an array like a variable passed by value.**

```
int main() {
   int v[10];
   ...
   foo(v);
}
```

Is there a way to let foo work on v without actually modify it?

Vector vs Array

- **Working on an array like a variable passed by value.**

```
int main() {
   int v[10];
   ...
   foo(v);
}
```

Actually no, unless you work on a copy of v made:

Vector vs Array

- **Working on an array like a variable passed by value.**
  - **Some solutions are available but...**

```
int main() {
   int v[10];
   ...
   for (i=0;i<10;i+
      v2[i]=v[i]
   foo(v2);
}
```

Actually no, unless you work on a copy of v made:

1. before calling the function

1. This must be done ever (lots of extra vectors required).
2. You still have to cope with the actual size vs max size of the vector...

Vector vs Array

- **Working on an array like a variable passed by value.**
  - **Some solutions are available but...**

```
int main() {
   int v[10];
   ...
   foo(v);
}
```

Actually no, unless you work on a copy of v made:

1. before calling the function
2. same code but inside foo

1. You still have to cope with the actual size vs max size of the vector by modifying the function

Rel. 24/04/2016

Vector vs Array

- **Working on a vector like a variable passed by value.**
  - **Previous container solution already provide you of a good way of working things out**

```
int main() {
  vector<int> v;
  ...
  foo(v);
}
```

void foo(vector<int> vect);

⬇

the container is passed by value because it is a single object not a reference to memory!

Vector vs Array

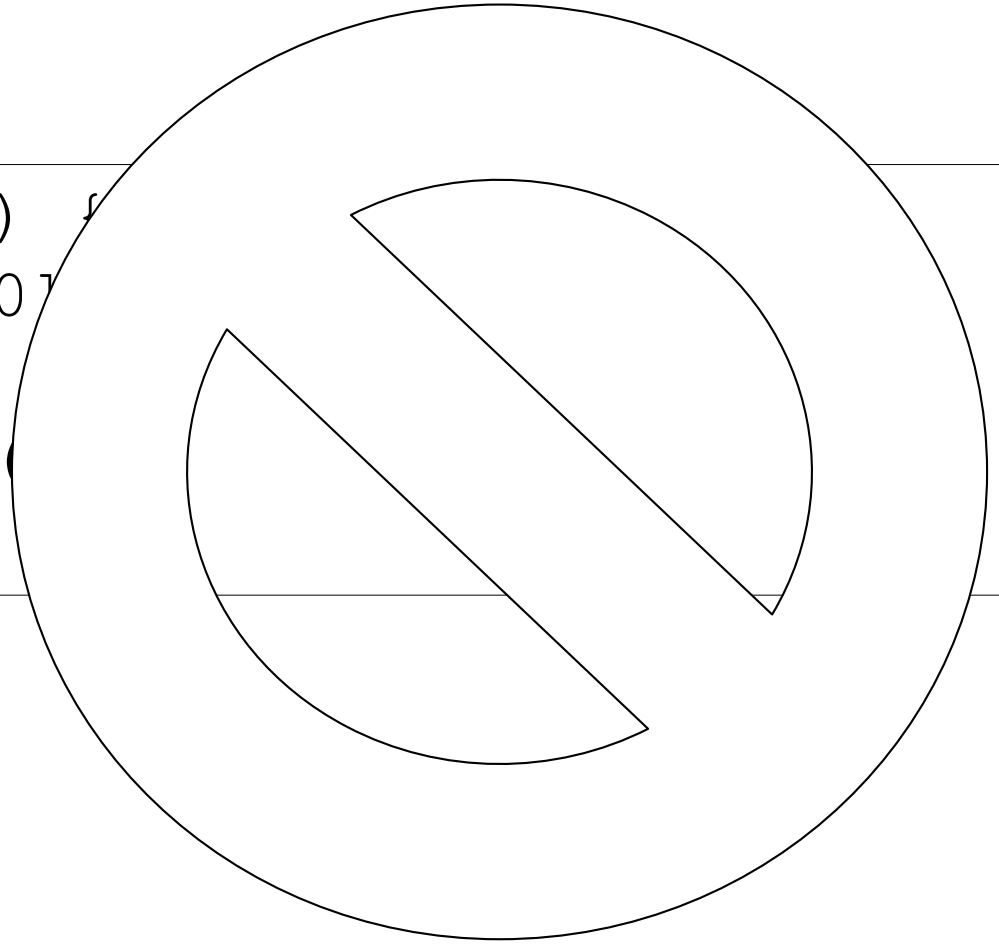
- **Most complex of all: returning an array.**
  - **Red now means error!**

```
int main() {
   int v[10];
   ...
   v = foo();
}
```

Vector vs Array

- **Most complex of all: returning an array.**
  - **Red now means error!**

```
int main() {
   int v[10]
   ...
   v = foo(
}
```



Vector vs Array

- **Most complex of all: returning an array.**
  - **Red now means error!**

```
int main() {
   int v[10];
   ...
   v = foo();
}
```

int [] foo(); cannot be defined in C. In fact, v is a static reference to a vector, thus an address, and you cannot modify it.

Vector vs Array

- **Most complex of all: returning an array.**
  - **You can resort to dynamic memory but...**

```
int main() {
   int *v;
   ...
   v = foo();
}
```

int * foo();

⬇

this is a working solution, with several drawbacks:

1. the actual size of v?

Rel. 24/04/2016

Vector vs Array

- **Most complex of all: returning an array.**
  - **You can resort to dynamic memory but...**

```
int main() {
   int *v;
   size_t s;
   ...
   v = foo(&s);
}
```

int * foo();

⬇

this is a working solution, with several drawbacks:

1. the actual size of v?

⬇

int * foo(size_t * actual_size);

⬇

you must change the prototype to pass one extra variable by reference

Rel. 24/04/2016

Vector vs Array

- **Most complex of all: returning an array.**
  - **You can resort to dynamic memory but...**

```
int main() {
   int *v;
   ...
   v = foo();
}
```

int * foo();

⬇

this is a working solution, with several drawbacks:

1. the actual size of v?
2. when & where v is destroyed?

Vector vs Array

- **Most complex of all: returning an array.**
  - **You can resort to dynamic memory but...**

```
int main() {
   int *v;
   ...
   v = foo();
   ...
   free(v);
   ...
}
```

int * foo();

⬇

this is a working solution, with several drawbacks:

1. the actual size of v?
2. when & where v is destroyed?

⬇

a free operation is required!

Rel. 24/04/2016

Vector vs Array

- **Most complex of all: returning an array.**
  - **Very easy using containers**

```
int main() {
   vector<int> v;
   ...
   v = foo();
   ...
}
```

vector<int> foo();

⬇

because v is a container, and is provided by a copy constructor and/or a = operator, once you return a vector object you get it copied into v.

End of it.

Vector vs Array

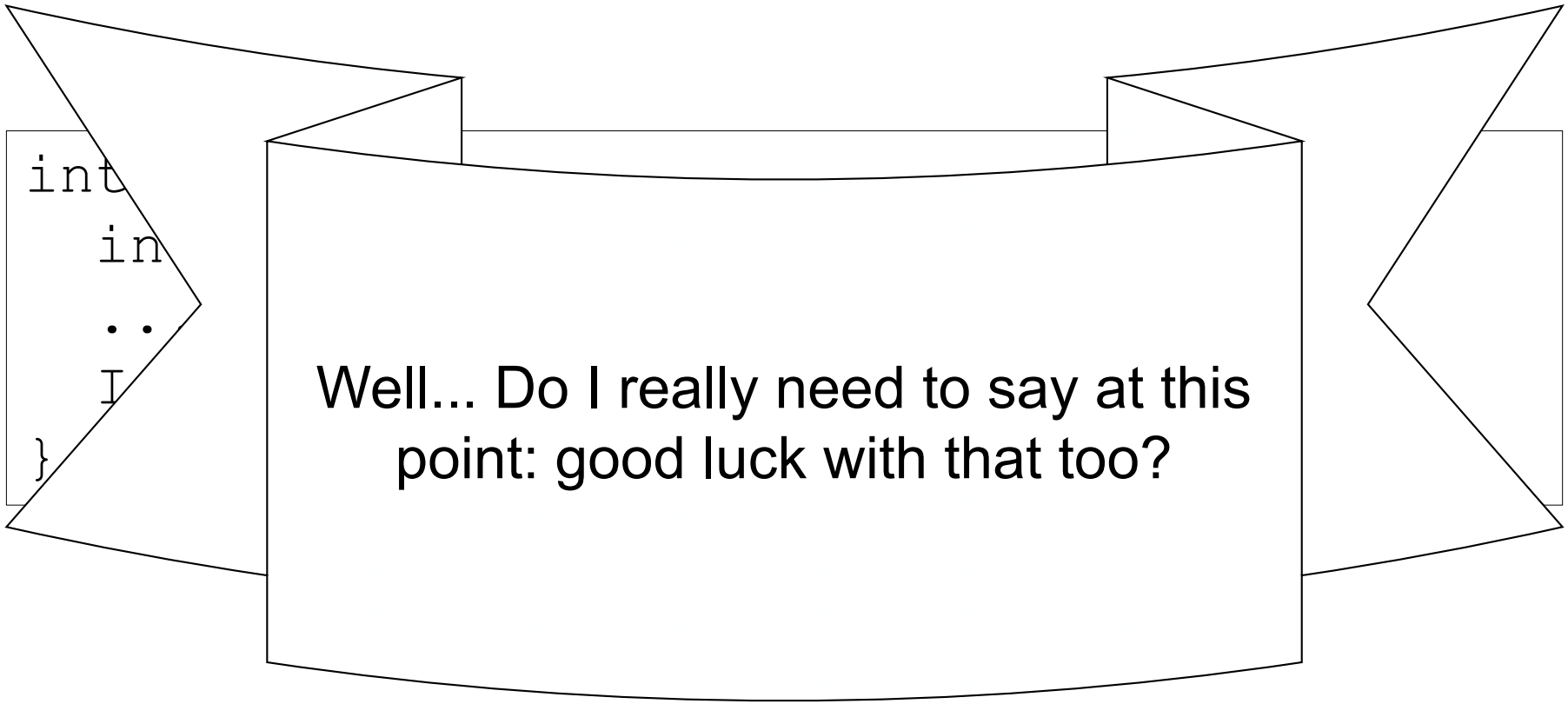
- **Eventually... try to enlarge/reduce the array size.**

```
int main() {
   int v[10];
   ...
   I_want_to_resize_v(v);
}
```


Rel. 24/04/2016

Vector vs Array

- **Eventually... try to enlarge/reduce the array size.**



Outline

- The Standard Template Library
- The STL's Components
- Containers definition
- STL Containers
- STL Containers: The Vector
- STL Iterators
- STL Algorithms

Iterators Definition

- **An iterator is an extension to the pointer**
  - **It implements the standard pointer operators**
- **It gives you the ability to cycle through the contents of the container like a pointer to cycle through an array**
- **Iterators used by the algorithms to move through the containers.**

```
class_name<template_parameters>::iterator name;
```

Iterators Definition

- name - name of the iterator (like a variable name);
- class_name - name of the STL container;
- template_parameters - parameters to the template;

array

- **Iterators used by the algorith ve through the containers.**

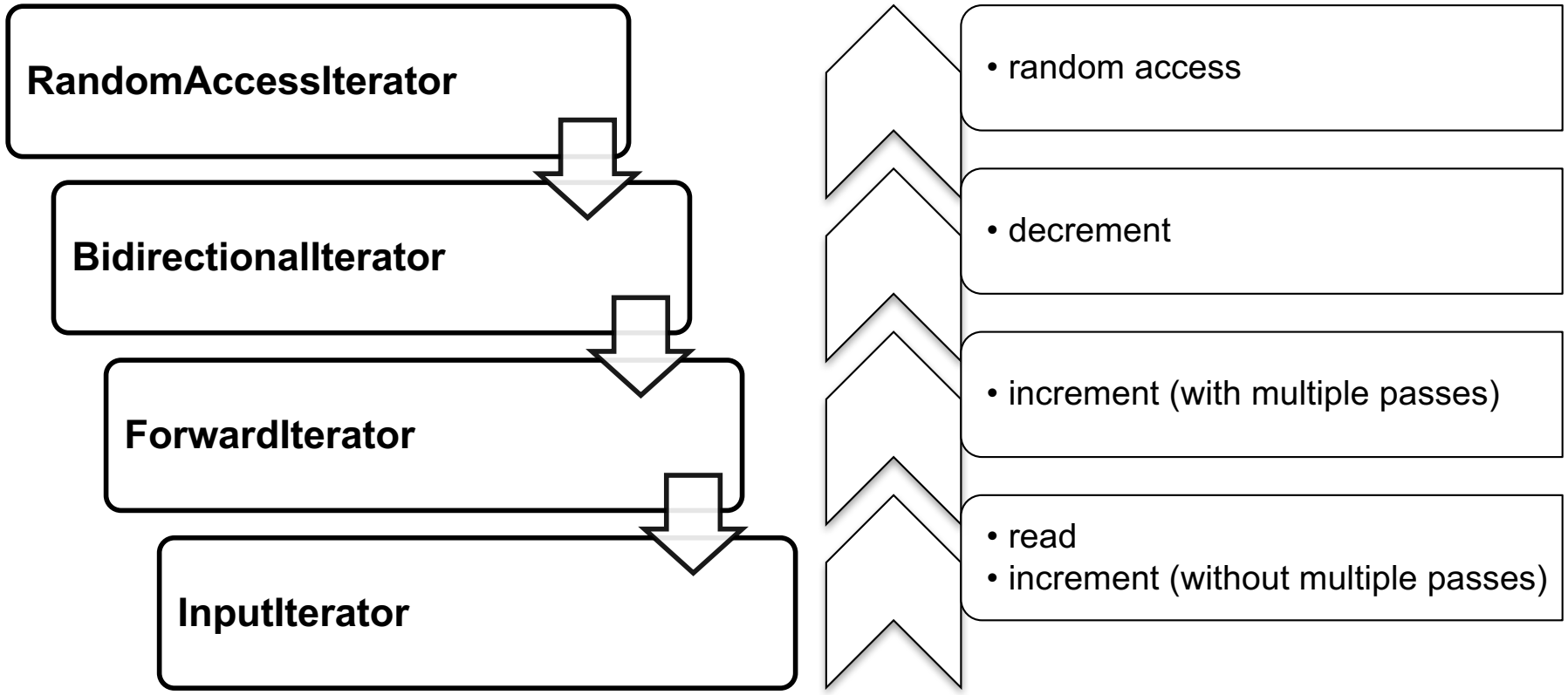
```
class_name<template_parameters>::iterator name;
```

Iterators Properties

- **Each container class in STL has a corresponding iterator that functions appropriately for the container**
  - **E.g., an iterator in a vector class allows random access while a list class does not allow that.**

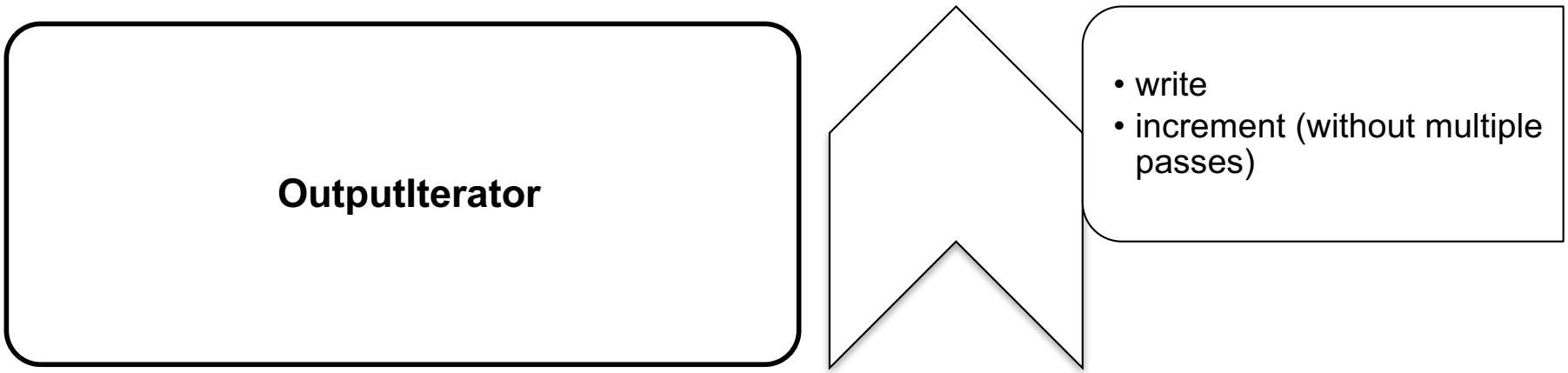
Iterators Hierarchy

- **Iterators are instantiation of classes.**
  - **It exist a hierarchy of Iterators.**
    - **They differ by the kind of operations allowed.**



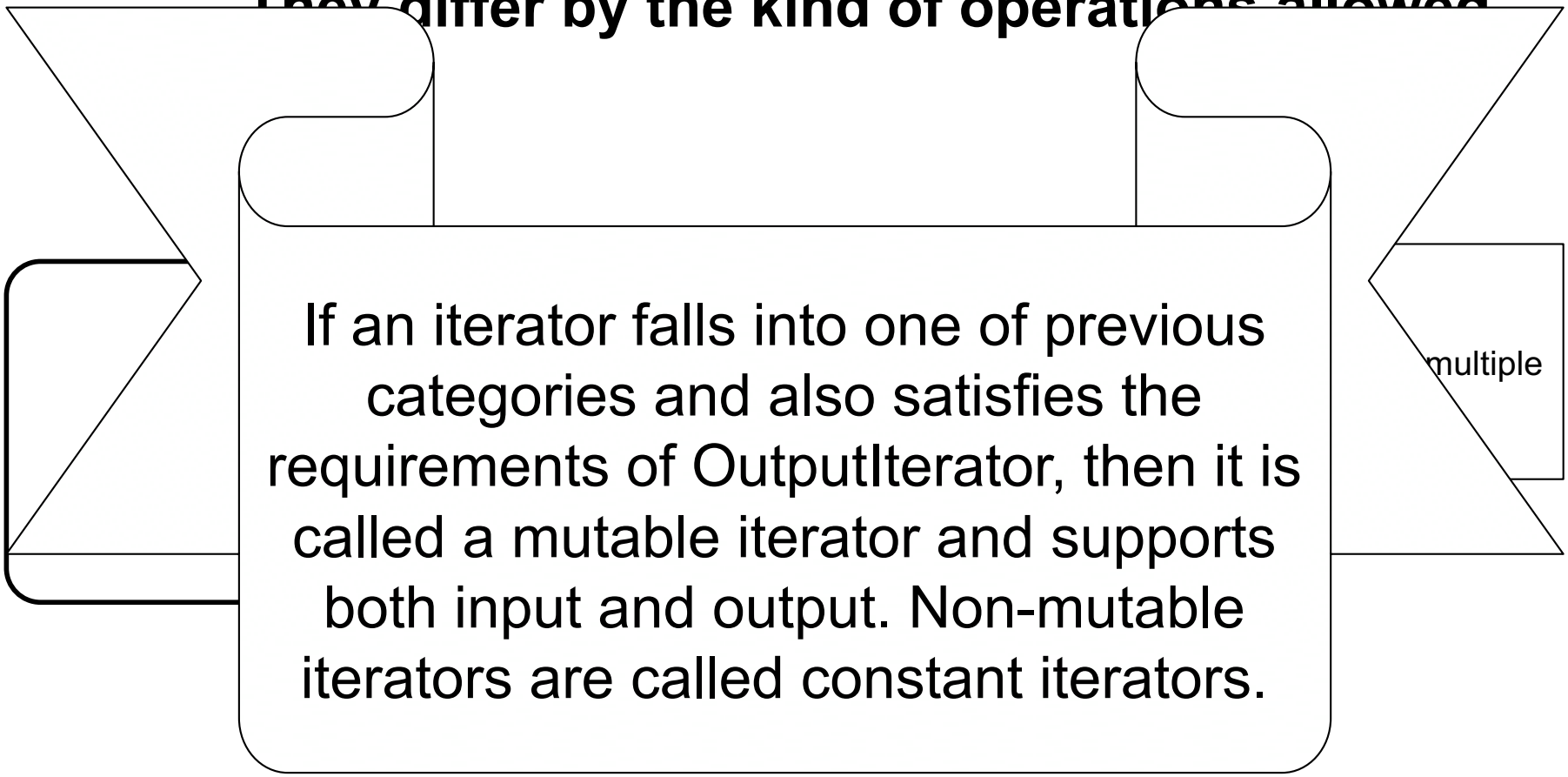
Iterators Hierarchy

- **Iterators are instantiation of classes.**
  - **It exist a hierarchy of Iterators.**
    - **They differ by the kind of operations allowed.**



Iterators Hierarchy

- **Iterators are instantiation of classes.**
  - **It exist a hierarchy of Iterators.**
    They differ by the kind of operations allowed



Iterators Basic Operations

- **General operations allow to:**
  - **assign an iterator a value, i.e., the start of a container;**
  - **access the value that an iterator "points to";**
  - **increment the iterator to point to the next value;**
  - **check if the iterator has reached a predetermined value, i.e., the end of a container;**

Iterators Basic Operations

- **General operations allow to:**
  - **assign an iterator a value, i.e., the start of a container;**
    - **. = operator;**
  - **access the value that an iterator "points to";**
  - **increment the iterator to point to the next value;**
  - **check if the iterator has reached a predetermined value, i.e., the end of a container;**

Iterators Basic Operations

- **General operations allow to:**
  - **assign an iterator a value, i.e., the start of a container;**
  - **access the value that an iterator "points to";**
    - *** operator**
  - **increment the iterator to point to the next value;**
  - **check if the iterator has reached a predetermined value, i.e., the end of a container;**

Iterators Basic Operations

- **General operations allow to:**
  - **assign an iterator a value, i.e., the start of a container;**
  - **access the value that an iterator "points to";**
  - **increment the iterator to point to the next value;**
    - **+, +=, ++ operators**
  - **check if the iterator has reached a predetermined value, i.e., the end of a container;**

Iterators Basic Operations

- **General operations allow to:**
  - **assign an iterator a value, i.e., the start of a container;**
  - **access the value that an iterator "points to";**
  - **increment the iterator to point to the next value;**
    - **+, +=, ++ operators**
  - **check if the iterator has reached a predetermined value, i.e., the end of a container;**

when the iterator supports decrement, include -, -=, – operators!

Iterators Basic Operations

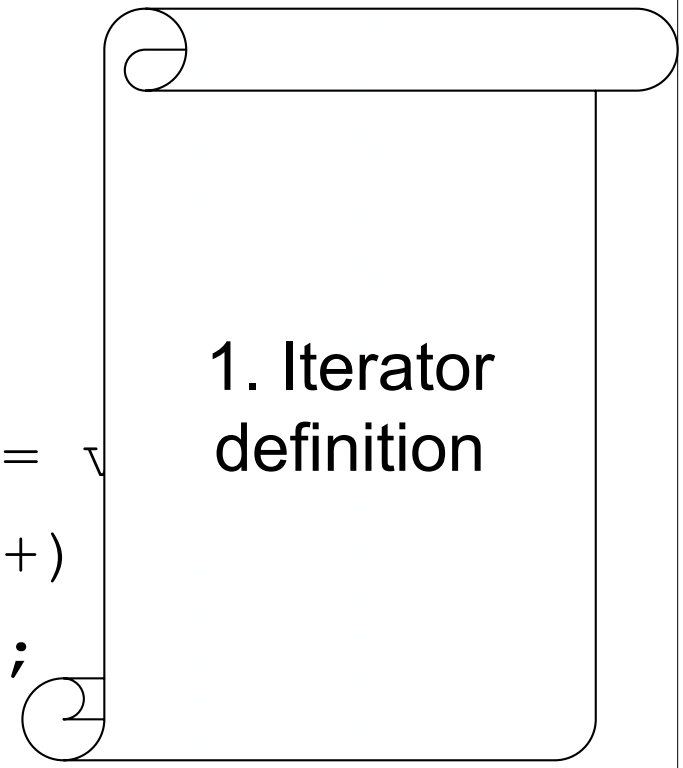
- **General operations allow to:**
  - **assign an iterator a value, i.e., the start of a container;**
  - **access the value that an iterator "points to";**
  - **increment the iterator to point to the next value;**
  - **check if the iterator has reached a predetermined value, i.e., the end of a container;**
    - **!=, == operators**

Iterators Example

```
#include <vector>
#include <iterator>
int main()
{
  vector<int> v(6, 0);
  vector<int>::iterator it = v.begin();
  for (; it != v.end(); it++)
     cout << (*it)++ << " ";
  cout << endl;
}
```

Iterators Example

```
#include <vector>
#include <iterator>
int main()
{
  vector<int> v(6, 0);
  vector<int>::iterator it = v
  for (; it != v.end(); it++)
     cout << (*it)++ << " ";
  cout << endl;
}
```



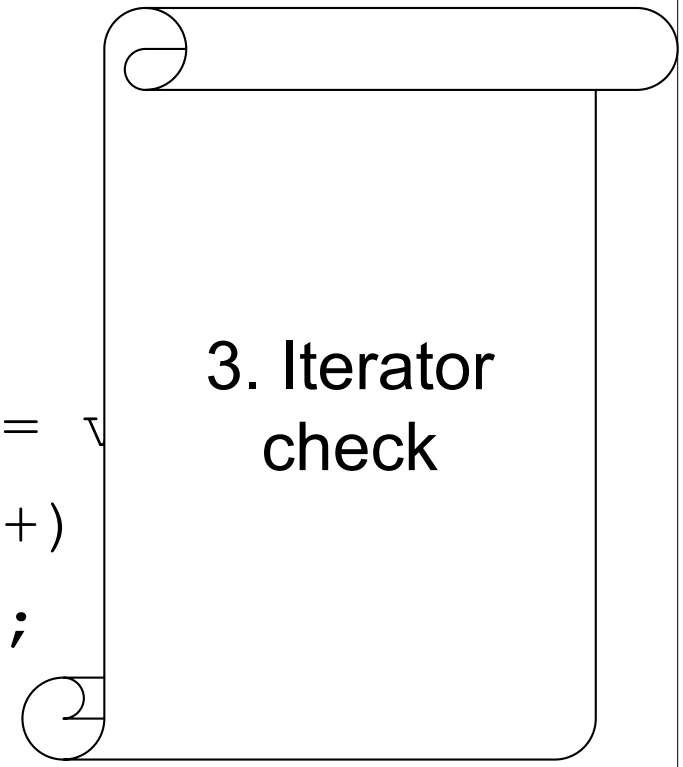
Iterators Example

```
#include <vector>
#i                    or>
ir
{
                      v(6, 0);
                      :iterator it = v.begin();
                       v.end(); it++)
                      (*it)++ << " ";
                      ;
}
```

2. Iterator assignment

Iterators Example

```
#include <vector>
#include <iterator>
int main()
{
  vector<int> v(6, 0);
  vector<int>::iterator it = v
  for (; it != v.end(); it++)
     cout << (*it)++ << " ";
  cout << endl;
}
```



Iterators Example

```
#include <vector>
#i                    or>
ir
{
                      v(6, 0);
                      :iterator it = v.begin();
                      v.end(); it++)
                      (*it)++ << " ";
                      ;
}
```

4. Iterator increment

Iterators Example

```
#include <vector>
#include <iterator>
int main()
{
  vector<int> v(6, 0);
  vector<int>::iterator it = v
  for (; it != v.end(); it++)
     cout << (*it)++ << " ";
  cout << endl;
}
```

5. Iterator dereferenced access

Iterators Example

```
#include <vector>
#include <iterator>
int main()
{
  vector<int> v(6, 0);
  vector<int>::iterator it = v.begin();
  for (; it != v.end(); it++)
     cout << (*it)++ << " ";
```

v.begin() and v.end() return the start and the end of the container data.

if the container is empty v.begin() == v.end()

Iterators Example

```
#include <vector>
#include <iterator>
int main()
{
  vector<int> v(6, 0);
  vector<int>::iterator it = v.begin();
  for (; it != v.end(); it++)
     cout << (*it)++ << " ";
```

v.begin() and v.end() "point" to valid addresses.

there is no null pointer value for an iterator!

Iterators Example

```
#include <vector>
#include <iterator>
int main()
{
  vector<int> v(6, 0);
  vector<int>::iterator it = v.begin();
  for (; it != v.end(); it++)
     cout << (*it)++ << " ";
```

v.end() returns an iterator which is the past-the-end value for the container.

there is no null pointer value for an iterator!

Outline

- The Standard Template Library
- The STL's Components
- Containers definition
- STL Containers
- STL Containers: The Vector
- STL Iterators
- STL Algorithms

STL Algorithms

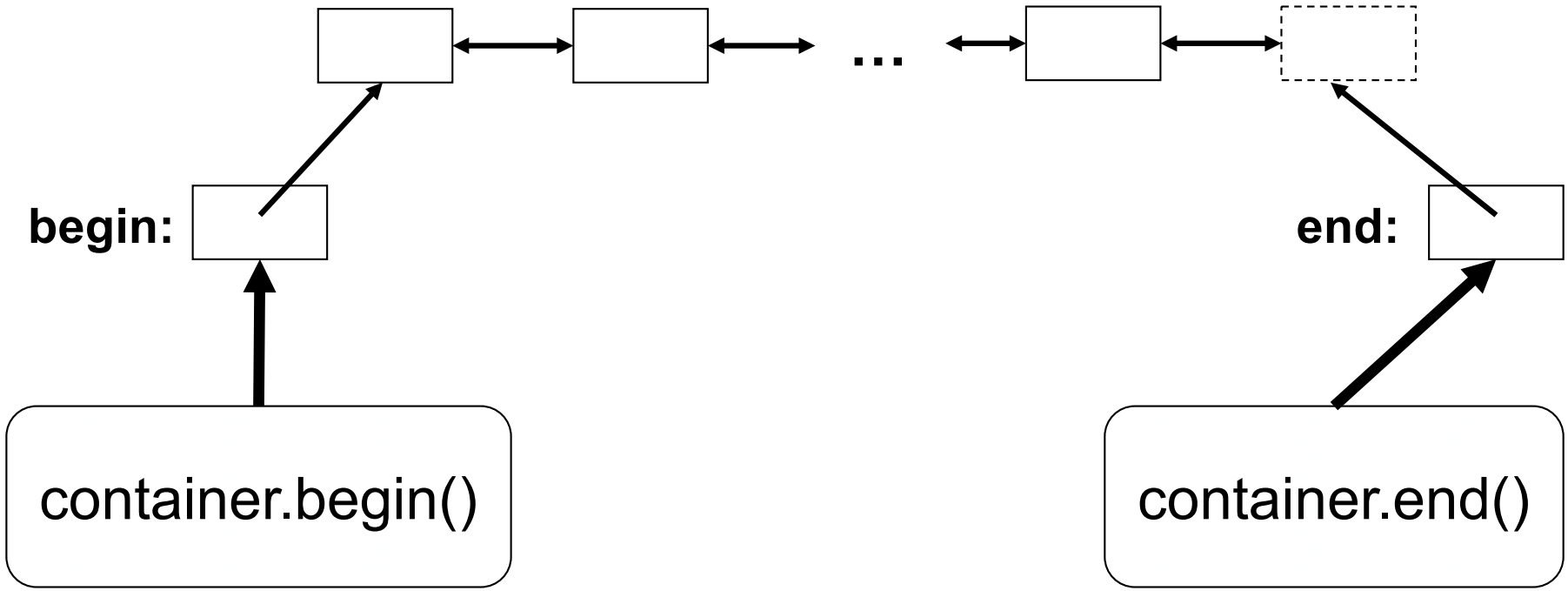
- **Used generically across a variety of containers.**
- **Many algorithms operate on sequence of elements defined by pairs of iterators**
  - **Start and End**
- **It is possible to create new algorithms that operate in a similar fashion so they can be used with the STL containers and iterators.**

STL Algorithms (II)

- **Mutating Sequence Algorithms**
  - **like copy(), remove(), replace(), fill(), swap(), etc.,**
- **Non Modifying sequence Algorithms**
  - **like find(), count(),search(), mismatch(), and equal()**
- **Numerical Algorithms**
  - **accumulate(), partial_sum(), inner_product(), and adjacent_difference()**

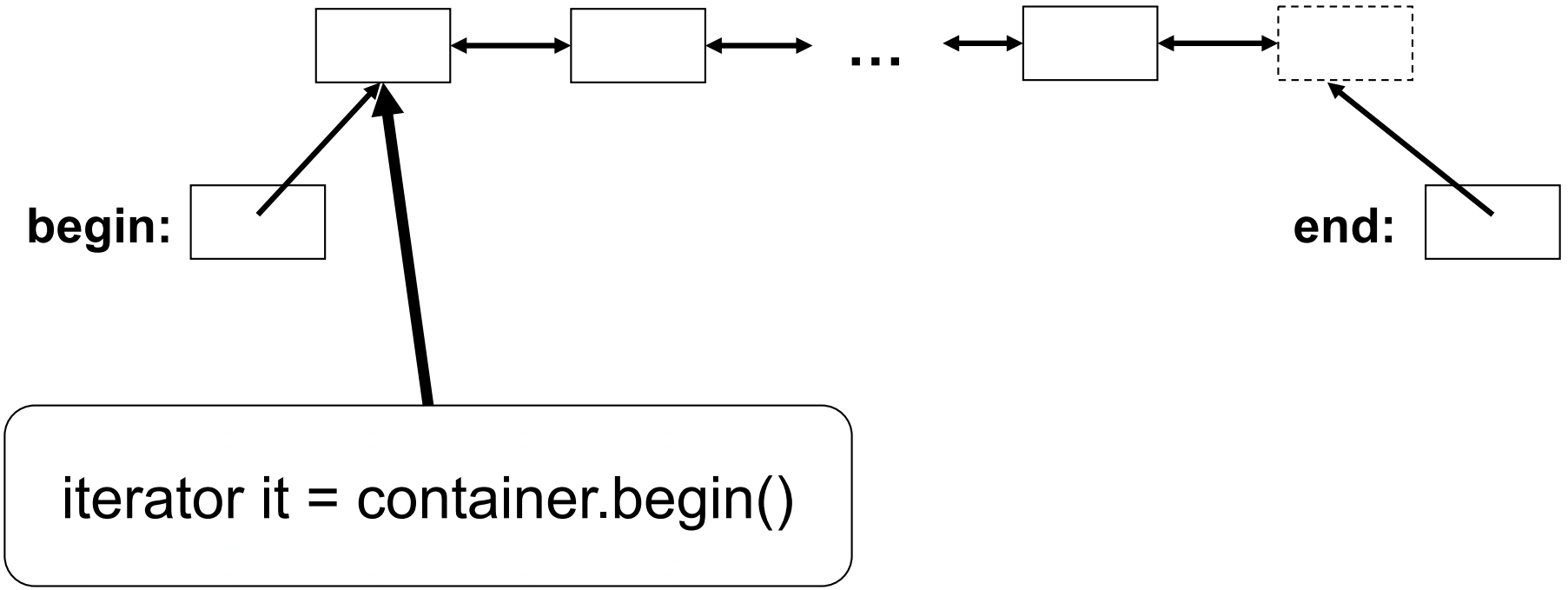
STL Algorithms: Principles

- **A generic container organizes its items in the following way:**



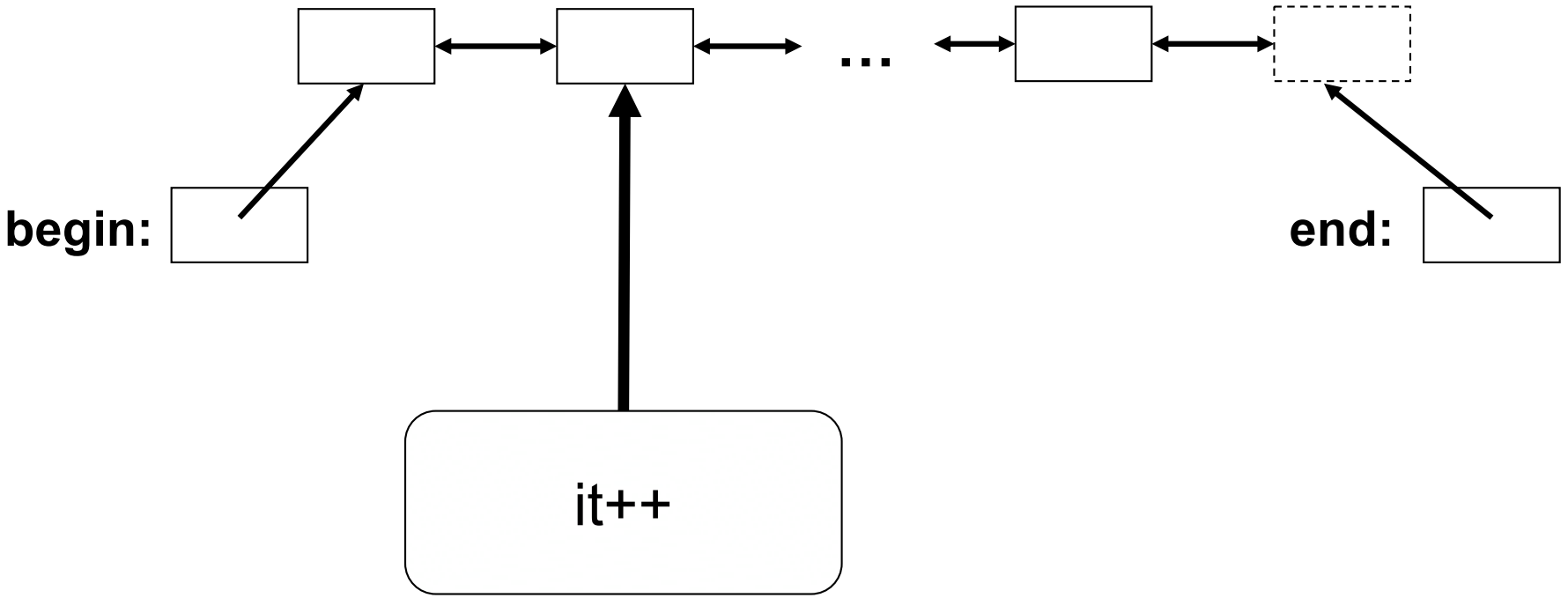
STL Algorithms: Principles

- **Then a generic algorithm is able to move across the data by using a proper iterator**



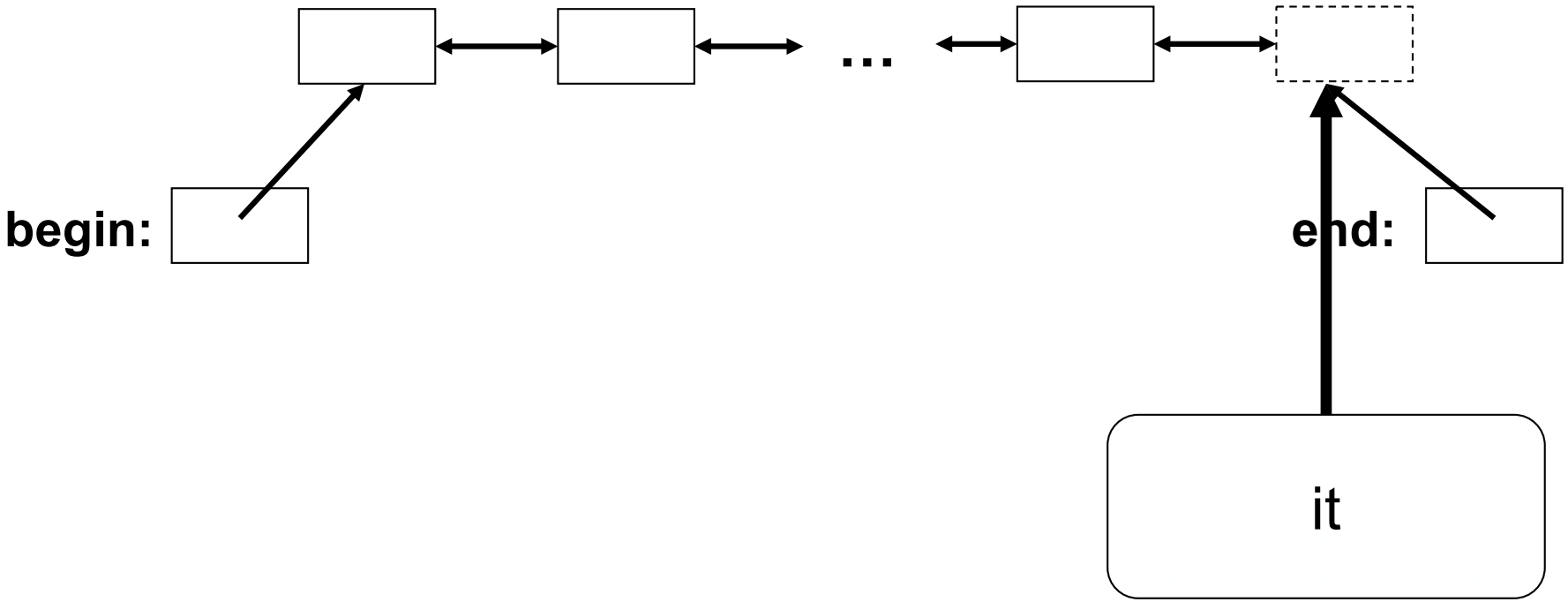
STL Algorithms: Principles

- **Then a generic algorithm is able to move across the data by using a proper iterator.**



STL Algorithms: Principles

- **It may continue until it reaches the end.**
  - **And/or the algorithm finds what it was looking for.**



STL Algorithms: find

- **The most simple algorithm in the STL is the find function**
  - **Like all functions is in the form of a template function**

```
template <class InputIterator, class T>
InputIterator find (InputIterator first,
InputIterator last, const T& val);
```

STL Algorithms: find

- **The most simple algorithm in the STL is the find function**
  - **Like all functions is in the form of a template function**
  - **Returns an InputIterator**

```
template <class InputIterator, class T>
InputIterator find (InputIterator first,
InputIterator last, const T& val);
```

STL Algorithms: find

- **The most simple algorithm in the STL is the find function**
  - **Like all functions is in the form of a template function**
  - **Returns an InputIterator**
  - **Receives the first and the last elements for the search**

```
template <class InputIterator, class T>
InputIterator find (InputIterator first,
InputIterator last, const T& val);
```

STL Algorithms: find

- **The most simple algorithm in the STL is the find function**
  - **Like all functions is in the form of a template function**
  - **Returns an InputIterator**
  - **Receives the first and the last elements for the search**
  - **Expect a third argument with the value to search (still in the form of a template value)**

```
template <class InputIterator, class T>
InputIterator find (InputIterator first,
InputIterator last, const T& val);
```

STL Algorithms: find example

- **Notice the function usage and the post-function check**

```
#include <algorithm>
int main ()
{
 vector<int> myvector (5);
 ...
 vector<int>::iterator it;
 it = find (myvector.begin(),
            myvector.end(), 3);
 if (it != myvector.end())
      cout << ”I found 3!" << endl;
 return 0;
}
```

STL Algorithms: other examples

- **Lots of common algorithm are already at your disposal.**

```
#include <algorithm>

// return num elements equal to 7
int i = count(c.begin(), c.end(), 7);

// fill the container with elements set to 7
fill(c.begin(), c.end(), 7);

// find min or max element (with default comparison)
c::iterator min_it = min_element(c.begin(), c.end());
c::iterator max_it = max_element(c.begin(), c.end());
```

STL Algorithms: Advanced

- **Most of the algorithms can work with Function Object.**
  - **Template function that can be programmed as callback functions**
    - ***Remember the pointer-to-function concept.***
  - **Some are already defined in the <functional> part of the STL library.**

Rel. 24/04/2016

STL Algorithms: Advanced

```
...
#include <algorithm>
#include <functional>
int increase_int (int i) { return ++i; }
int main () {
  std::vector<int> vect1, vect2;
  for (int i=1; i<6; i++)
    vect1.push_back (i);

  vect2.resize(vect1.size());
  transform (vect1.begin(), vect1.end(),
             vect2.begin(), increase_int);
```

STL Algorithms: Advanced

```
...
#include <algorithm>
#include <functional>
int increase_int (int i) { return ++i; }
int main () {
  std::vector<int> vect1, vect2;
  for (int i=1; i<6; i++)
    vect1.push_back (i);

  vect2.resize(vect1.size());
  transform (vect1.begin(), vect1.end
             vect2.begin(), increase_int);
```

the two libraries required (plus vector, etc.)

STL Algorithms: Advanced

```
...
#include <algorithm>
#include <functional>
int increase_int (int i) { return ++i; }
int main () {
  std::vector<int> vect
  for (int i=1; i<6; i+
    vect1.push_back (i)

  vect2.resize(vect1.si
  transform (vect1.begi                    ,
             vect2.beg                     nt);
```

my custom function to increase an int number

STL Algorithms: Advanced

```
...
#include <algorithm>
#include <functional>
int increase_int (int i) { return ++i; }
int main () {
  std::vector<int> vect1, vect2;
  for (int i=1; i<6; i++)
    vect1.push_back (i);

  vect2.resize(vect1.size());
  transform (vect1.begin(), vect1.end(),
             vect2.begin(), increase_int);
```

Resize the second vector about the size of the first one so they match in size!

STL Algorithms: Advanced

```
template <class InputIterator, class OutputIterator,
          class UnaryOperation>
OutputIterator transform (InputIterator first1,
                          InputIterator last1,
                          OutputIterator result,
                          UnaryOperation op);
```

```
transform (vect1.begin(), vect1.end(),
           vect2.begin(), increase_int);
```

STL Algorithms: Advanced

```
..
#i
#i
int
int
```

The point is transforming vect1 but saving the transformation in vect2.
The transformation operation is provided by the last parameter (which is a function)

```
transform (vect1.begin(), vect1.end(),
           vect2.begin(), increase_int);
```

STL Algorithms: Advanced

```
  transform (vect1.begin(), vect1.end(),
             vect2.begin(), vect1.begin(),
             plus<int>());
  cout << "Vect1 now contains:";
  for (vector<int>::iterator it=vect1.begin();
       it!=vect1.end();
       ++it)
       std::cout << ' ' << *it;
  std::cout << endl;
  return 0;
}
```

STL Algorithms: Advanced

```
transform (vect1.begin(), vect1.end(),
           vect2.begin(), vect1.begin(),
           plus<int>());
}
```

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class BinaryOperation>
OutputIterator transform (InputIterator1 first1,
                          InputIterator1 last1,
                          InputIterator2 first2,
                          OutputIterator result,
                          BinaryOperation binary_op);
```

STL Algorithms: Advanced

```
transform (vect1.begin(), vect1.end(),
           vect2.begin(), vect1.begin(),
           plus<int>());
}
```

The point is transforming vect1 but, this time, the overloaded function expect a second input (vect2) and the output will be in vect1.
The transformation operation is provided by the last parameter (which is a function object of the library and requires a template specification)

STL Algorithms: Advanced

```
  transform (vect1.begin(), vect1.end(),
             vect2.begin(), vect1.begin(),
             plus<int>());
  cout << "Vect1 now contains:";
  for (vector<int>::iterator it=vect1.begin();
       it!=vect1.end();
       ++it)
       std::cout << ' ' << *it;
  std::cout << endl;
  return 0;
}
```


Малые Автюхи, Калинковичский район, Республики Беларусь