System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block G: Advanced C++
6. STL & the Notion of Iterator

Eugene Zouev

Generic Programming: Introduction

Lectures 1-5: To Remind

- Function & Class Templates
- Template Type & Non-type Parameters
- Template Instantiation: Implicit & Explicit
- Explicit & Partial Specializations
- Functional Objects & Templates

Lecture 6: Plan for Today

- C++ Standard Template Library
- **The notion of iterators**

GP: Definition & Principles

Definition:

<u>**Generic Programming**</u> is an approach that provides very general principles and techniques to specify the basic programming concepts:

- Data structures
- Algorithms

Basic Principles:

Two <u>fundamental principles</u> behind GP:

- Generality
- Efficiency

The Best Example:

- **Standard Template Library (STL) for the C++ language**

References & Stages:

- A.Stepanov, D.Műsser: theoretical foundations of GP
- A.Stepanov, M.Lee, D.Műsser: first implementation of the GP approach (standalone STL library for C++)
- 1994: STL became part of the C++ Standard Library
- 1998: STL exists as Chapters 20 (partially), 23,24,25 of the ISO C++ Standard

STL as an Implementation of GP

- **STL doesn't use OOP**; GP is an orthogonal approach

  $\boxed{GP \neq OOP}$

  No encapsulation, (almost) no inheritance etc.

OOP

Class
Data Operations on data

A class may be considered as a **container** together with **algorithms** on its elements.

GP

Generic Containers
(independent from any algorithm)

Generic Algorithms
(independent from any particular container)

The difference:

- In OO libraries, algorithms are provided within the classes of things they manipulate on.
- In STL, almost all algorithms are provided externally to the container classes.

STL Structure (1)

STL has seven kinds of components

*** First-order components:**

- **Containers**
- **Algorithms**
- **Iterators**

*** Second-order components:**

- **Functional Objects**
- **Adaptors**
- **Allocators**
- **Traits**

Collection of typical **data structures** such as Vectors (expandable arrays), Lists, & Sets – and their modifications: Deques (double-ended lists), Maps (dictionaries), Multi-sets, Multi-maps (sets & maps with duplicated elements)

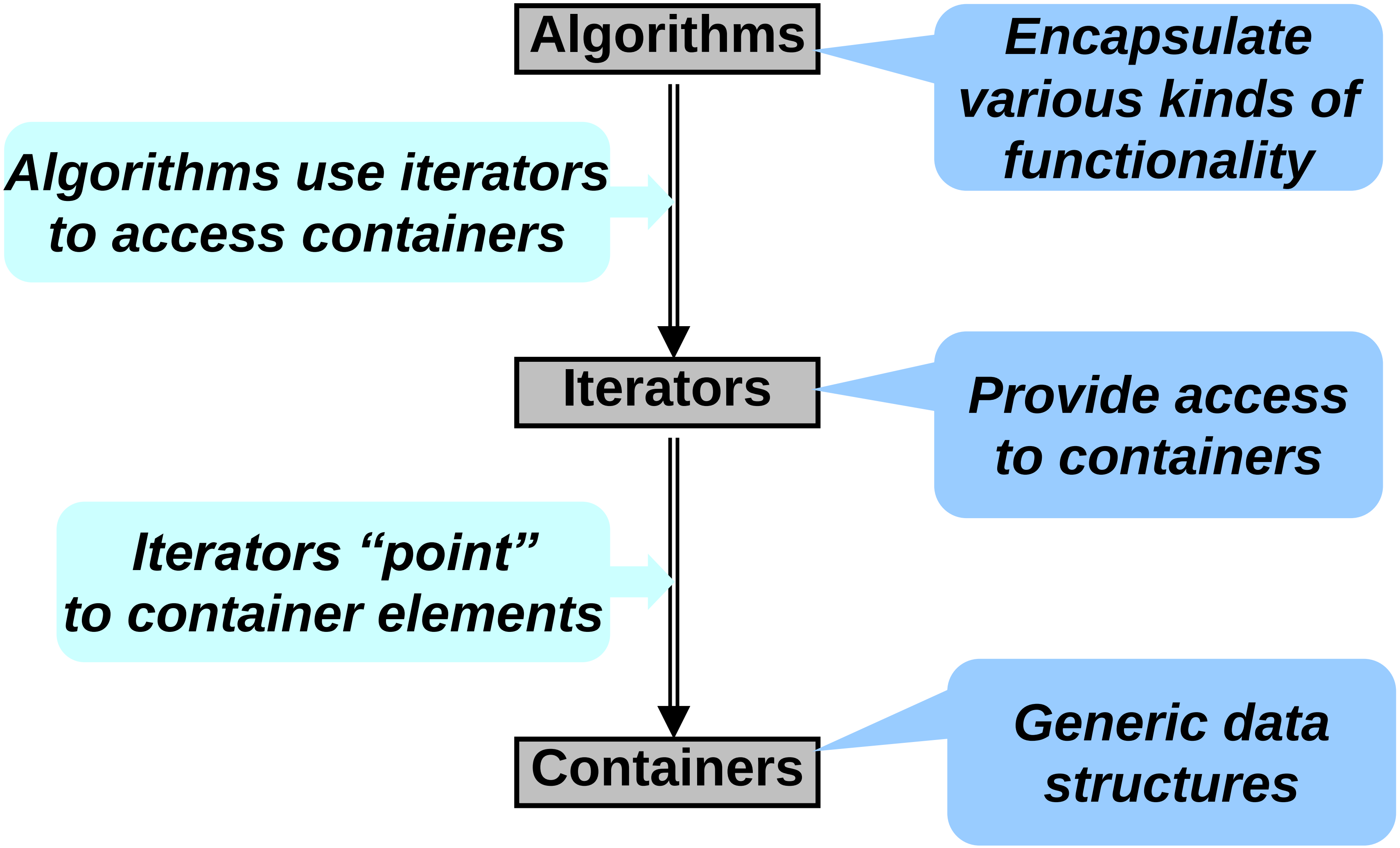
Collection of typical **functionality** on arbitrary data structures such as Creating/ copying, Searching, Sorting, Adding/ deleting/modifying container elements, Splitting/merging, Transformations: reordering/converting.

A general means to **get access to containers** and to manipulate containers and their elements.

Major characteristics:

- Iterators are (conceptually) types; but any object may play a role of iterator if its type satisfies a set of requirements.
- Iterators could be considered as a generalization of C++ pointers
- Iterators are the **interface** between containers and algorithms that manipulate on them.

GP: Iterators, Containers, Algorithms



GP: Introduction to Iterators (1)

©Andrew König (the idea)

Very simple example:

Find the first array element which <u>is equal to a given value</u>.

```
const int* find0 ( const int* array, int n, int x )
{       1                       3          4
    const int* p = array;
    for ( int i = 0; i<n; i++ )
    {
        if ( *p == x ) return p;  // success
        p++;  2
    }
    return 0;  // fail
}
```

How to weaken the restrictions?

The limitations of the algorithm:

1. It finds the *integer* value;
2. It looks through the integer *array*;
3. We must specify *the address of the first array element*.
4. We must specify the array *size*.

GP: Introduction to Iterators (2)

Step 1:

Generalize *the type* of the array elements.

```
template < typename T >
T* find1 ( T* array, int n, const T& x )
{
   T* p = array;
   for ( int i = 0; i<n; i++ )
   {
      if ( *p == x ) return p;  // success
      p++;
   }
   return 0;  // fail
}
```

A more generic function; the new assumption is that we need to have **operator==()** in T type defined.

- We removed **const** specifier assuming **const** is a part of T type parameter.
- We pass the third parameter "by reference".

GP: Introduction to Iterators (3)

The fundamental lack of the algorithm (from the generality's point of view) is that it is strongly related to the specific data structure: **<u>array</u>**.

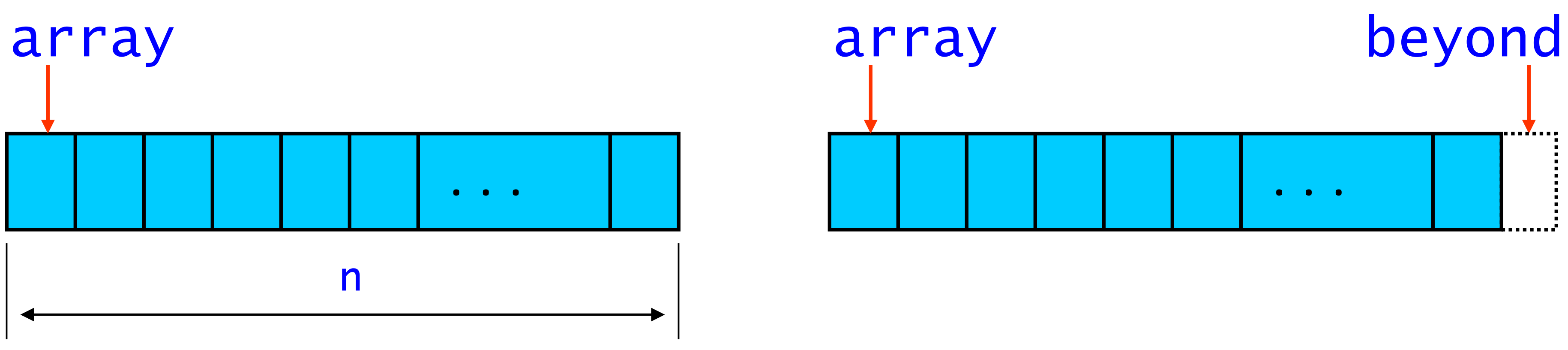
- We know *the address of the first array element.*
- We use *++ operator* to move from one array element to another.
- We use the information about the *array size* for exiting the loop.

Let's try to avoid the dependence on the array in our algorithm.

GP: Introduction to Iterators (4)

Step 2: Remove the array *size*.

```
template < typename T >
T* find2 ( T* array, T* beyond, const T& x )
{
    T* p = array;
    while ( p != beyond )
    {
        if ( *p == x ) return p;  // success
        p++;
    }
    return 0;  // fail
}
```



GP: Introduction to Iterators (5)

Step 3: Remove the zero pointer.

```
template < typename T >
T* find3 ( T* array, T* beyond, const T& x )
{
    T* p = array;
    while ( p != beyond )
    {
        if ( *p == x ) return p;  // success
        p++;
    }
    return beyond;  // instead of 0
}
```

We have changed the algorithm's specification!

Rationale:

Returning 0 is OK for arrays and pointers because integer zero literal may be converted to any pointer.

But we want *to remove* the assumption that the function works *for arrays only*; so we need to weaken the algorithm's dependence on pointers' specifics.

GP: Introduction to Iterators (6)

Step 4: Simplifying and optimizing the algorithm.

```
template < typename T >
T* find4 ( T* first, T* beyond, const T& x )
{
    T* p = first;
    while ( p != beyond && *p != x )
        p++;
    return p;  // the result
}
```

Modifications:

- We have *joined two checks* together "inverting" one of them; this should work faster.
- We excluded one of two `return` statements; the code became shorter.
- We renamed `array` by `first` because our algorithm already doesn't contain any explicit assumptions on the data structure processed!

GP: Introduction to Iterators (7)

Step 4: Conclusions.

```
template < typename T >
T* find4 ( T* first, T* beyond, const T& x )
{
    T* p = first;
    while ( p != beyond && *p != x )
        p++;
    return p;
}
```

- `find4` algorithm performs the search *in a data structure* (not in an array!).
- The data structure consists of elements of type `T`.
- **Assumption**: `T` type supports `operator!=()`.
- We access to the data structure elements using pointers `T*`
- **Assumption**: applying `operator++()` to `T*` we should *get* the pointer *to the next element* of the structure.

GP: Introduction to Iterators (8)

Step 5: Removing pointers! (the final algorithm)

```
template < typename T, typename P >
P find5 ( P first, P beyond, const T& x )
{
    P p = first;
    while ( p != beyond && *p != x )
        p++;
    return p;
}
```

Conclusions:

- `find5` algorithm performs the search *in a data structure* consisting of elements of type `T`.
- Access to the data elements is performed using object(s) of type `P`.

Requirements/Assumptions:

- Type `T` should support `operator!=()`.
- Type `P` should support `operator*()`; this operator should return a value of type `T`.
- Type `P` should support `operator++()`; the result should denote *the next element* of the data structure.
- Type `P` should support `operator!=()`.

GP: Introduction to Iterators (9)

Final Conclusion

- Type `T` (type of container elements) is called **value type**.
- Type `P` (used for accessing to container elements) is called **iterator type**, or **iterator**.

To use `find5` algorithm for searching in any container the following requirements should be met:

- **Value type** should support **`operator!=()`**.
- **Iterator type** should support **`operator*()`**; this operator should return a value of *value type*.
- **Iterator type** should support **`operator++()`**; the result should denote *the next container element*.
- **Iterator type** should support **`operator!=()`**.

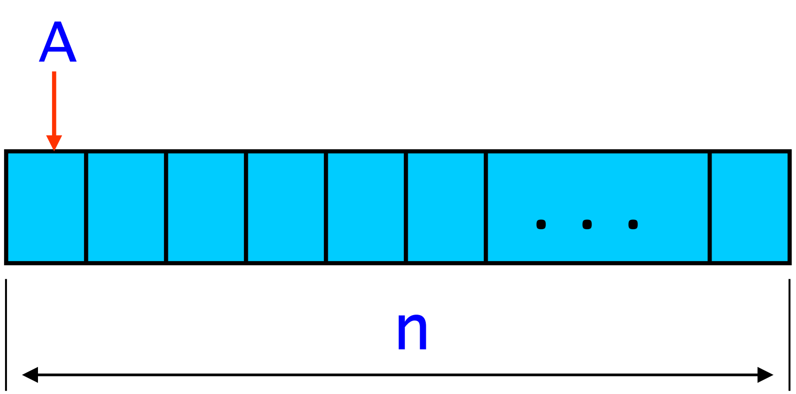
Typical features of input iterators

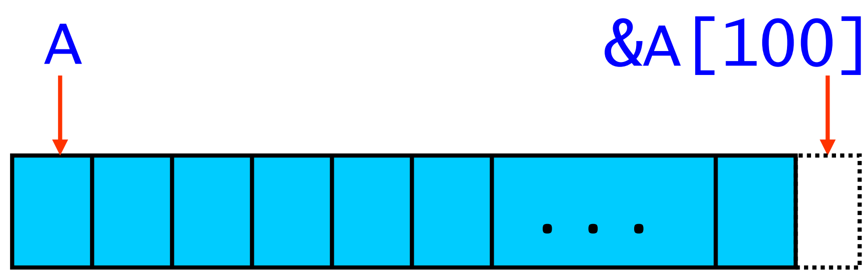
GP: Introduction to Iterators (10)

Examples of using finds

```
int A[100];
// Initializing A...

int* p1 = find1(A,100,7);
int* p2 = find5(A,&A[100],7);
```





find5 implicit instantiation:

T = int

P = int*

GP: Introduction to Iterators (11)

Why &A[100] is correct?

ISO C++ Standard

5.7 Additive operators

…

§5

… if the expression *P* points to the *i*th element of an array object, the expressions (*P*)+*N* (equivalently, *N*+(*P*)) and (*P*)*N* (where *N* has the value *n*) point to, respectively, the *i*+*n*th and *i*-*n*th elements of the array object, provided they exist.

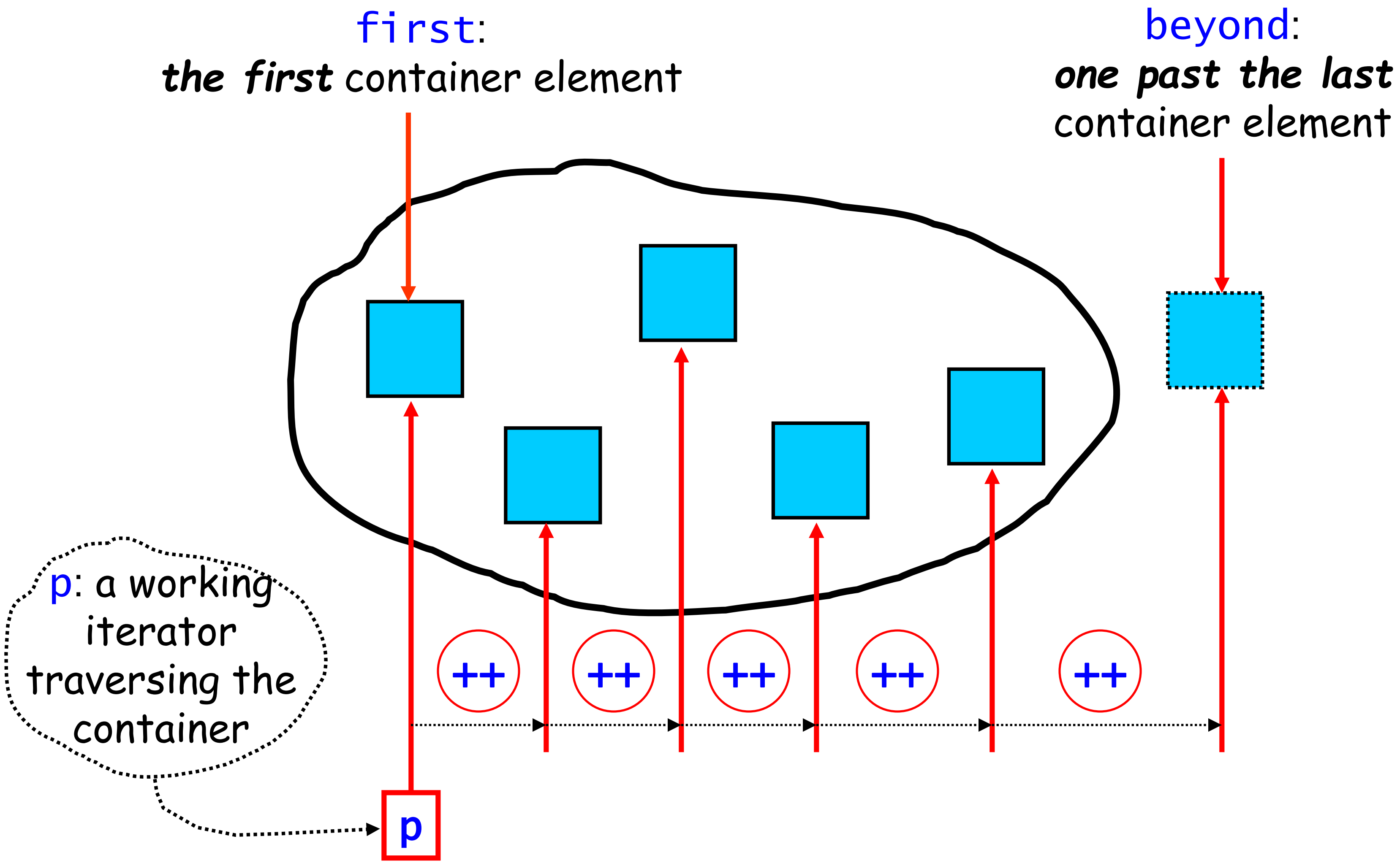
Moreover, if the expression *P* points to the last element of an array object, the expression (*P*)+1 points <u>one past the last element</u> of the array object,

and if the expression *Q* points one past the last element of an array object, the expression (*Q*)-1 points to the <u>last element</u> of the array object.

If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow…

GP: Algorithms & Iterators (1)

Using `find5`: a general scheme



The Task

The following algorithm searches ***the last*** element of the array that *satisfies a condition*:

```
int* find0 ( int* array, int n, bool (*c)(int) )
{
    int* p = array;
    for ( int i=n-1; i>=0; i-- )
    {
        if ( c(*p) ) return p;  // success
        p--;
    }
    return 0;  // fail
}
```

Task: write a semantically equivalent *generic algorithm* working with an arbitrary data structure.

(a) Use the concept of **functional types**;
(b) Apply Steps1-5 to the initial algorithm.

Write the complete list of requirements on the parameters.

GP: Algorithms & Iterators (2)

Now let's go further:
How to apply `find5` algorithm to a data structure <u>other than array</u>?

Array container:	`int A[100]`
Iterator type for accessing array:	`int*`

List container:	*Single-linked list* of integer elements
Iterator type for accessing list:	`???`

GP: Algorithms & Iterators (3)

The simple example of single-linked list implementation

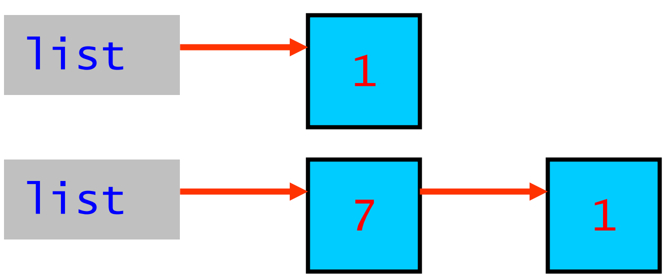
```
class NODE {
    int   value; // NODE's "contents"
    NODE* next;
public:
    NODE(int v) : value(v),next(0) { }
    void add(NODE* n)
    { n->next = this->next; this->next = n; }

            Implementation

};
```

The task (part one):
Write the full implementation of the NODE class. The implementation should meet the requirements on *value type*.

```
NODE* list = new NODE(1);
 . . .
NODE* n = new NODE(7);
list->add(n);
```



GP: Algorithms & Iterators (4)

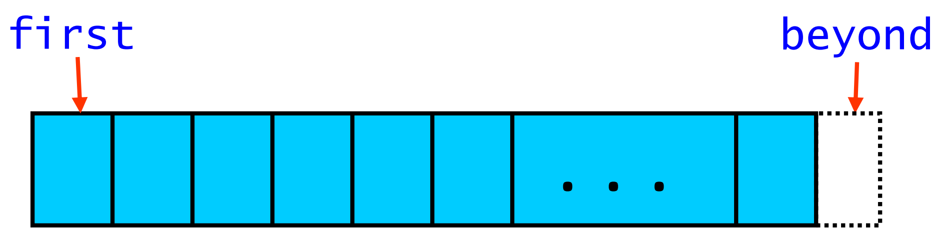
Write the iterator type for the list of NODEs

Requirements (to remind):

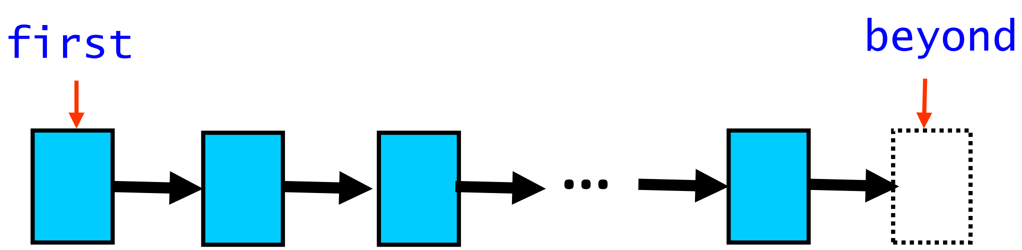
- `operator*()` for accessing the NODE's "contents"
- `operator++()` for traversing lists of NODEs
- `operator!=()` for comparing list elements

```
class iterator {
public:
    Iterator
    implementation
};
```

The effect:




```
A[100] a;
... // filling the array
find5(a,&a[100],7);
```




```
NODE* list = new NODE(1);
... // filling the list
find5(list,0,7);
```

Actually, a bit more complicated ☺

Technical Hints (1)

How to organize containers & their iterators

Container of elements of type T

```
class container_of_T
{
    . . .
};
```

Iterator for this container

```
class iterator_for_cont_of_T
{
    . . .
};
```

A generic algorithm

Container of elements of type T with its own enclosed iterator

```
class container {
    // containter
    // implementation
    class iterator {
        // iterator
        // implementation
    };
};
```

A generic algorithm

Technical Hints (2)

Final implementation: Container *with* Iterator

```
class NODE {
   int    value; // NODE's "semantics"
   NODE* next;
public:
   NODE(int v)…
   void add(NODE* n)…
   bool operator!=(NODE& n)…
   operator int()…

   class iterator;
   friend class iterator;

   class iterator {
      NODE* p;
   public:
     iterator(NODE* node)…
     int operator*()…, void operator++()…,
     bool operator!=(…)…, operator NODE*()…
   };
};
```

Implementation of the NODE container itself

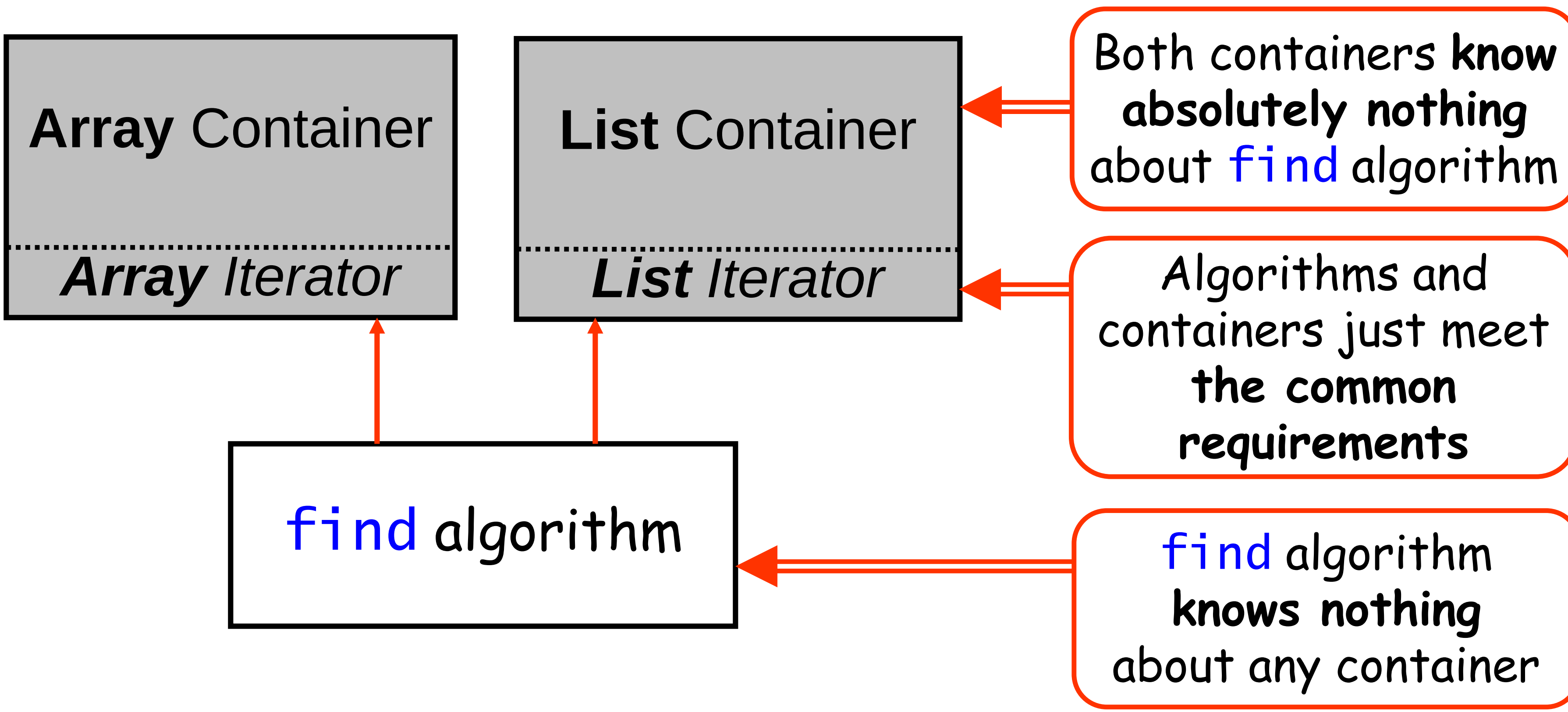
For free access to the NODE implementation from within iterator class

Implementation of the NODE's iterator

GP: Algorithms & Iterators (5)

Summary

- We have defined the **generic algorithm** for searching objects in containers
- We have applied the algorithm to two different **containers**: **array** & **list**
- We have seen that **the usage** of the algorithm for the different containers **is the same**.



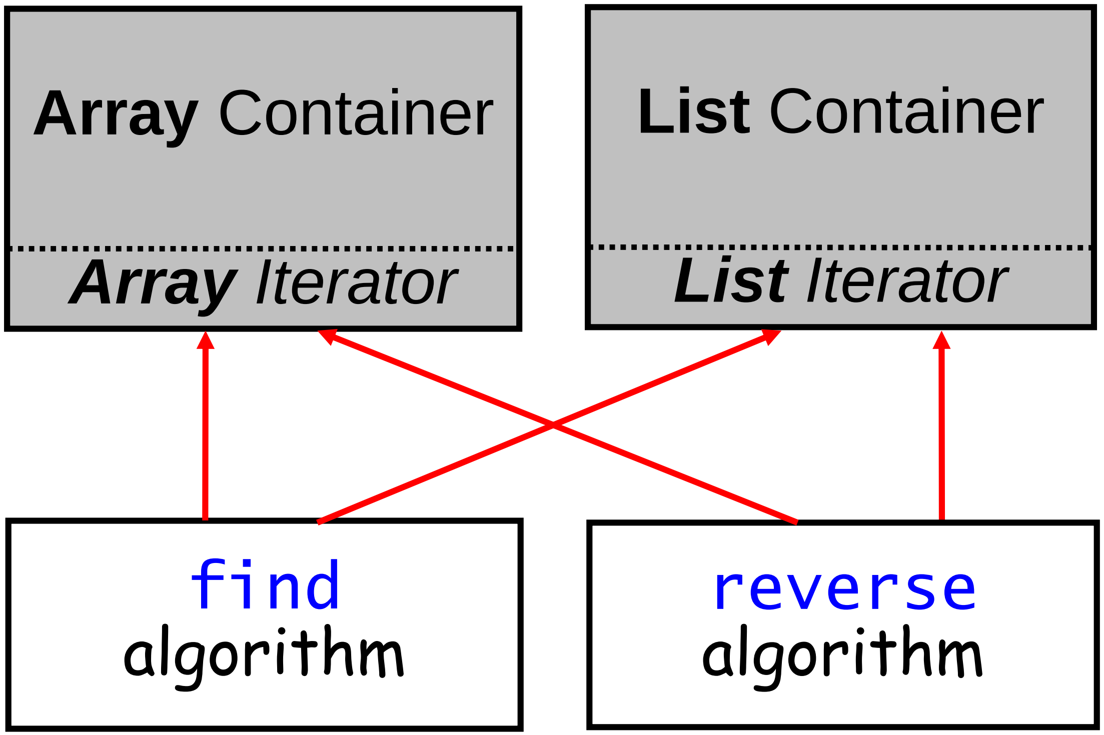
GP: Algorithms & Iterators (6)

Summary (again ☺):

- We have applied the searching algorithm to two different **containers**

Let's go further:

- Let's try **to vary the algorithm**, i.e., let's develop a different algorithm and apply it to our two containers:



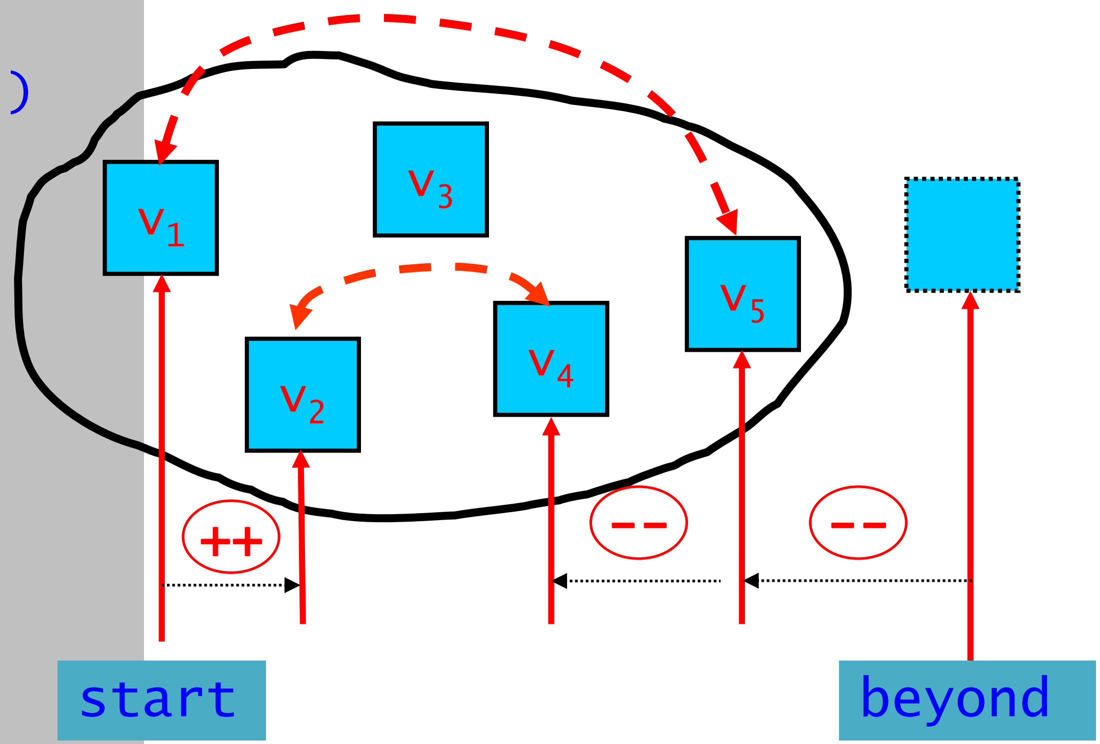
GP: Algorithms & Iterators (7)

Problem:

Change <u>the order</u> of the container elements

```
template<typename P,typename T>
void reverse(P start,P beyond)
{
   while ( start != beyond )
   {
      --beyond;
      if (start != beyond)
      {
         T t = *start;
         *start = *beyond;
         *beyond = t;
         ++start;
      }
   }
}
```

`reverse` generic algorithm
(*the idea* of the initial version)



GP: Algorithms & Iterators (8)

Problem:

Change the order of the container elements

```
template<typename P,typename T>
void reverse(P start,P beyond)
{
   while ( start != beyond )
   {
      --beyond;
      if ( start != beyond )
      {
         T t = *start;
         *start = *beyond;
         *beyond = t;
         ++start;
      }
   }
}
```

`reverse` *generic algorithm*
(*the idea* of the initial version)

Requirements on `P` iterator type:

++, **!=**, ***** operators — **The same**

-- operator — **New!**

***** operator should return *reference* but not *value* — **New!**

GP: Algorithms & Iterators (9)

Summary: Iterator requirements

Algorithm requires *Iterator type*

Algorithm	*Iterator type*	
`find`	**`operator++`**	
	`operator!=`	
	`operator*`	Should return a **value** of the *value type*
`reverse`	**`operator++`**	
	`operator--`	
	`operator!=`	
	`operator*`	Should return a **reference** to the value of the *value type*

Conclusion:

Different algorithms may require iterators with **different features**.

GP: Iterator Categories

`find`

Input Iterators
`x=*i; ++i; i++`
C++ Std, 24.1.1

Output Iterators
`*i=x; ++i; i++`
C++ Std, 24.1.2

Forward Iterators
Input ***and*** Output Iterators
C++ Std, 24.1.3

Bidirectional Iterators
Forward Iterators ***and***
`--i; i--`
C++ Std, 24.1.4

`reverse`

Random Access Iterators
Bidirectional Iterators ***and***
`i+n, i-n, i+=n, i-=n,`
`i<j, i<=j, i>j, i>=j`
C++ Std, 24.1.5

C++ pointers!

Assume:
`i`, `j` – iterators;
`x` – an object of *value type*;
`n` – an integer value

All iterators support `!=`, `==`, and `=` **operators**