C++ Object-Oriented Prog.
Unit 6: Generic Programming

CHAPTER 20: GENERIC PROGRAMMING

DR. ERIC CHOU

IEEE SENIOR MEMBER

Generic Programming

- Programming/developing algorithms with the abstraction of types
- The uses of the abstract type define the necessary operations needed when instantiation of the algorithm/data occurs

```
template <class T>
T Add(const T &t1, const T &t2)
{
    return t1 + t2;
}
```

STL (Standard Template Library)

A library of class and function templates

Components:

1. Containers:

   Generic "off-the-shelf" class templates for storing collections of data

1. Algorithms:

   Generic "off-the-shelf" function templates for operating on containers

1. Iterators:

   Generalized "smart" pointers provide a generic way to access container elements

LECTURE 1

Templates

Templates

Function templates

- Specify entire range of related (overloaded) functions
- Function-template specializations

Class templates

- Specify entire range of related classes
  - Class-template specializations

Function Templates

Overloaded functions

- Similar operations
  - Different types of data

Function templates

- Identical operations
  - Different types of data
- Single function template
  - Compiler generates separate object-code functions
- Unlike Macros they allow Type checking

Function Templates

Function-template definitions

- Keyword **`template`**
- List formal type parameters in angle brackets (**<** and **>**)
  - Each parameter preceded by keyword **`class`** or **`typename`**
    - **`class`** and **`typename`** interchangeable
    - **`template< class T >`**
    - **`template< typename ElementType >`**
    - **`template< class BorderType, class FillType >`**
  - Specify types of
    - Arguments to function
    - Return type of function
    - Variables within function

```cpp
#include <iostream>
using namespace std;

// function template printArray definition
template< class T > void printArray( const T *array, const int count ){
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";
   cout << endl;
} // end function printArray

int main(){
   const int aCount = 5, bCount = 7, cCount = 6;
   int a[ aCount ] = { 1, 2, 3, 4, 5 };
   double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ cCount ] = "HELLO";  // 6th position for null
   cout << "Array a contains:" << endl;
   // call integer function-template specialization
   printArray( a, aCount );
   cout << "Array b contains:" << endl;
   // call double function-template specialization
   printArray( b, bCount );
   cout << "Array c contains:" << endl;
   // call character function-template specialization
   printArray( c, cCount );
   return 0;
} // end main
```

Function template definition; declare single formal type parameter **T**.

T is type parameter; use any valid identifier.

If **T** is user-defined type, stream-insertion operator must be overloaded for class **T**.

Creates complete function-template specialization for printing array of **int**s:

```cpp
void printArray( const int *array, const int count
)
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " "
   cout << endl;
} // end function printArray
```

Compiler infers **T** is **double**; instantiates function-template specialization where **T** is **double**.

Compiler infers **T** is **char**; instantiates function-template specialization where **T** is **char**.

Demo Program: function1.cpp

Go Notepad++!!!

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\function1>function1
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Overloading Function Templates

Related function-template specializations

- Same name
  - Compiler uses overloading resolution

Function template overloading

- Other function templates with same name
  - Different parameters
- Non-template functions with same name
  - Different function arguments
- Compiler performs matching process
  - Tries to find precise match of function name and argument types
  - If fails, function template
    - Generate function-template specialization with precise match

LECTURE 2

Generic Class

Class Templates

Stack

- LIFO (last-in-first-out) structure

Class templates

- Generic programming
- Describe notion of stack generically
  - Instantiate type-specific version
- Parameterized types
  - Require one or more type parameters
    - Customize “generic class” template to form class-template specialization

Demo Program: call_stack.cpp + stack.h

Go Notepad++!!!

stack.h

```
1  #ifndef TSTACK1_H
2  #define TSTACK1_H
3  template< class T >
4  class Stack {
5    public:
6     Stack( int = 10 );  // default cor
7      // destructor
8     ~Stack() { delete [] stackPtr; }
9
10    bool push( const T& );  // push
11    bool pop( T& );         // pop an element off the stack
12    // determine whether Stack is empty
13    bool isEmpty() const { return top == -1; } // end function isEmpty
14    // determine whether Stack is full
15    bool isFull() const { return top == size - 1; } // end function isFull
16
17   private:
18     int size;     // # of elements in the s
19     int top;      // location of the top ele
20     T *stackPtr;  // pointer to the stack
21 }; // end class Stack
```

```
23 template< class T >
24 Stack< T >::Stack( int s ){
25    size = s > 0 ? s : 10;
26    top = -1;  // Stack initially empty
27    stackPtr = new T[ size ]; // al
28 } // end Stack constructor
29
30 // push element onto stack;
31 // if successful, return true; otherwise, return false
32 template< class T >
33 bool Stack< T >::push( const T &pushValue ){
34    if ( !isFull() ) {
35       stackPtr[ ++top ] = pushValue;  // place item on Stack
36       return true;  // push successful
37    } // end if
38    return false;  // push unsuccessful
39 }
40
41 // pop element off stack;
42 // if successful, return true; o
43 template< class T >
44 bool Stack< T >::pop( T &pop
45    if ( !isEmpty() ) {
46       popValue = stackPtr[ top-- ];  // remove item from Stack
47       return true;  // pop successful
48    } // end if
49    return false;  // pop unsuccessful
50 } // end function pop
51 #endif
```

Constructor creates array of type **T**. For example, compiler generates

```
stackPtr = new T[ size ];
```

for class-template specialization **Stack< double >**.

Member functions preceded with header

```
template< class T >
```

Member function preceded with header

```
template< class T >
```

Function parameters of type **T**.

Specify class-template definition; type parameter **T** indicates type of **Stack** class to be created.

Array of elements of type **T**.

Use binary scope resolution operator (**::**) with class-template name (**Stack< T >**) to tie definition to class template's scope.

Use binary scope resolution operator (**::**) with class-template name (**Stack< T >**) to tie definition to class template's scope.

call_stack.cpp

```cpp
#include <iostream>
#include "stack.h"  // Stack class template definition
using namespace std;

int main(){
   Stack< double > doubleStack( 5 );
   double doubleValue = 1.1;
   cout << "Pushing elements onto doubleStack\n";
   while ( doubleStack.push( doubleValue ) ) {
      cout << doubleValue << ' ';
      doubleValue += 1.1;
   } // end while
   cout << "\nStack is full. Cannot push " << doubleValue
      << "\n\nPopping elements from doubleStack\n";
   while ( doubleStack.pop( doubleValue ) ) cout << doubleValue << ' ';
   cout << "\nStack is empty. Cannot pop\n";
   Stack< int > intStack;
   int intValue = 1;
   cout << "\nPushing elements onto intStack\n";
   while ( intStack.push( intValue ) ) {
      cout << intValue << ' ';
      ++intValue;
   } // end while
   cout << "\nStack is full. Cannot push " << intValue
      << "\n\nPopping elements from intStack\n";
   while ( intStack.pop( intValue ) )  cout << intValue << ' ';
   cout << "\nStack is empty. Cannot pop\n";
   return 0;
} // end main
```

Link to class template definition.

Instantiate object of class **`Stack< double >`**.

Invoke function **`push`** of class-template specialization **`Stack< double >`**.

Invoke function **`pop`** of class-template specialization **`Stack< double >`**.

Note similarity of code for **`Stack< int >`** to code for **`Stack< double >`**.

Execution Result:

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\stack>call_stack
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Demo Program: call_stack2.cpp

Go Notepad++!!!

```cpp
#include <iostream>
using namespace std;

#include "stack.h"  // Stack class template definition

// function template to manipulate Stack< T >
template< class T >
void testStack(Stack< T > &theStack,  T value,  T increment,  const char *stackName ){
   cout << "\nPushing elements onto " << stackName << '\n';
   while ( theStack.push( value ) ) {
      cout << value << ' ';
      value += increment;
   } // end while
   cout << "\nStack is full. Cannot push " << value
        << "\n\nPopping elements from " << stackName << '\n';
   while ( theStack.pop( value ) )
      cout << value << ' ';
   cout << "\nStack is empty. Cannot pop\n";
} // end function testStack

int main(){
   Stack< double > doubleStack( 5 );
   Stack< int > intStack;
   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );
   return 0;
} // end main
```

Function template to manipulate **`Stack< T >`** eliminates similar code from previous file for **`Stack< double >`** and **`Stack< int >`**.

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\stack>call_stack2

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Class Templates and Non-type Parameters

Class templates

- **Nontype parameters**
  - Default arguments
  - Treated as consts
  - Example:

```
template< class T, int elements >
Stack< double, 100 > mostRecentSalesFigures;
```

    - Declares object of type `Stack< double, 100>`
- **Type parameter**
  - Default type
    - Example:

```
template< class T = string >
```

Class Templates and Non-type Parameters

Overriding class templates

- Class for specific type
  - Does not match common class template
- Example:

```
template<>
Class Array< Martian > {
 // body of class definition
};
```

LECTURE 3

Template and Inheritance

Templates and Inheritance

Several ways of relating templates and inheritance:

- Class template derived from class-template specialization
- Class template derived from non-template class
- Class-template specialization derived from class-template specialization
- Non-template class derived from class-template specialization

Templates and Friends

Friendships between class template and

- Global function
- Member function of another class
- Entire class

Friendship

- In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to "friends".
- Friends are functions or classes declared with the friend keyword.
- A non-member function can access the private and protected members of a class if it is declared a **friend** of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword **friend**.

Templates and Friends

`friend` functions

- Inside definition of **`template< class T > class X`**
  - **`friend void f1();`**
    - **`f1()`** **`friend`** of all class-template specializations
  - **`friend void f2( X< T > & );`**
    - **`f2( X< float > & )`** **`friend`** of **`X< float >`** only,
      `f2( X< double > & )` **`friend`** of **`X< double >`** only,
      `f2( X< int > & )` **`friend`** of **`X< int >`** only,
      …
  - **`friend void A::f4();`**
    - Member function **`f4`** of class **`A`** **`friend`** of all class-template specializations

Templates and Friends

`friend` functions

- Inside definition of **`template< class T > class X`**
  - **`friend void C< T >::f5( X< T > & );`**
    - Member function **`C<float>::f5( X< float> & )`** **`friend`** of **`class X<float>`** only

`friend` classes

- Inside definition of **`template< class T > class X`**
  - **`friend class Y;`**
    - Every member function of **`Y`** friend of every class-template specialization
  - **`friend class Z<T>;`**
    - **`class Z<float>`** **`friend`** of class-template specialization **`X<float>`**, etc.

Demo Program: friend.cpp

Go Notepad++!!!

friend.cpp

```cpp
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\friend>friend
24
```

Templates and static Members

Non-template class

- **`static`** data members shared between all objects

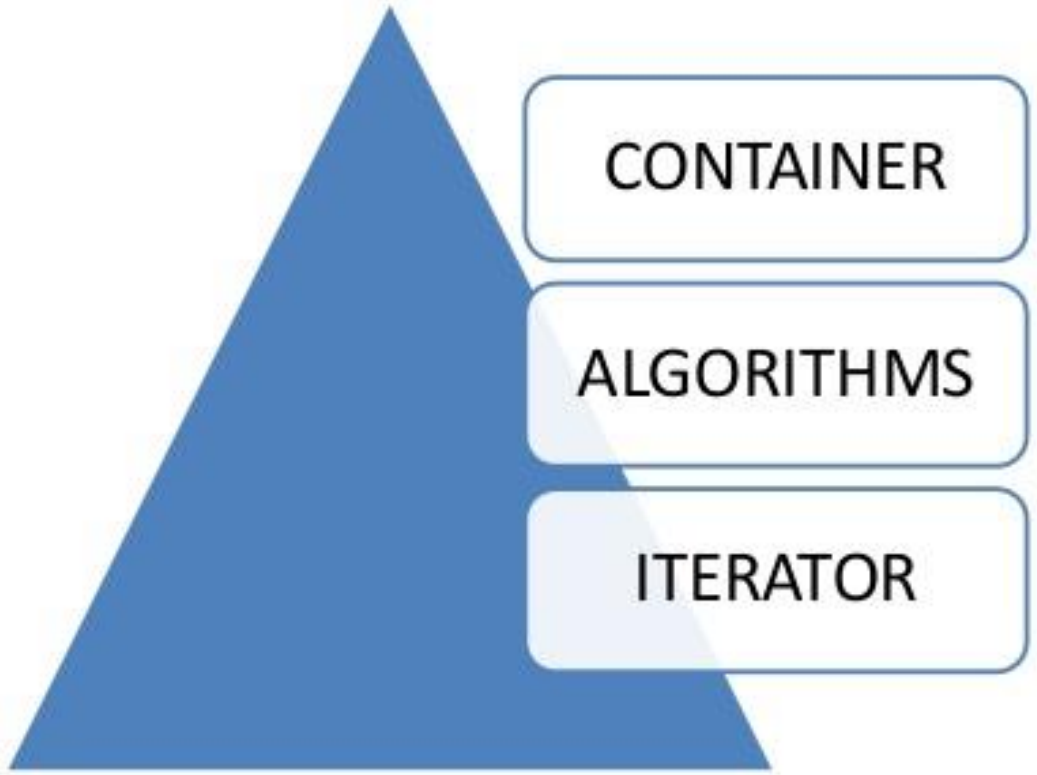
Class-template specialization

- Each has own copy of **`static`** data members
- **`static`** variables initialized at file scope
- Each has own copy of **`static`** member functions

LECTURE 4

Standard Template Library

Components of STL

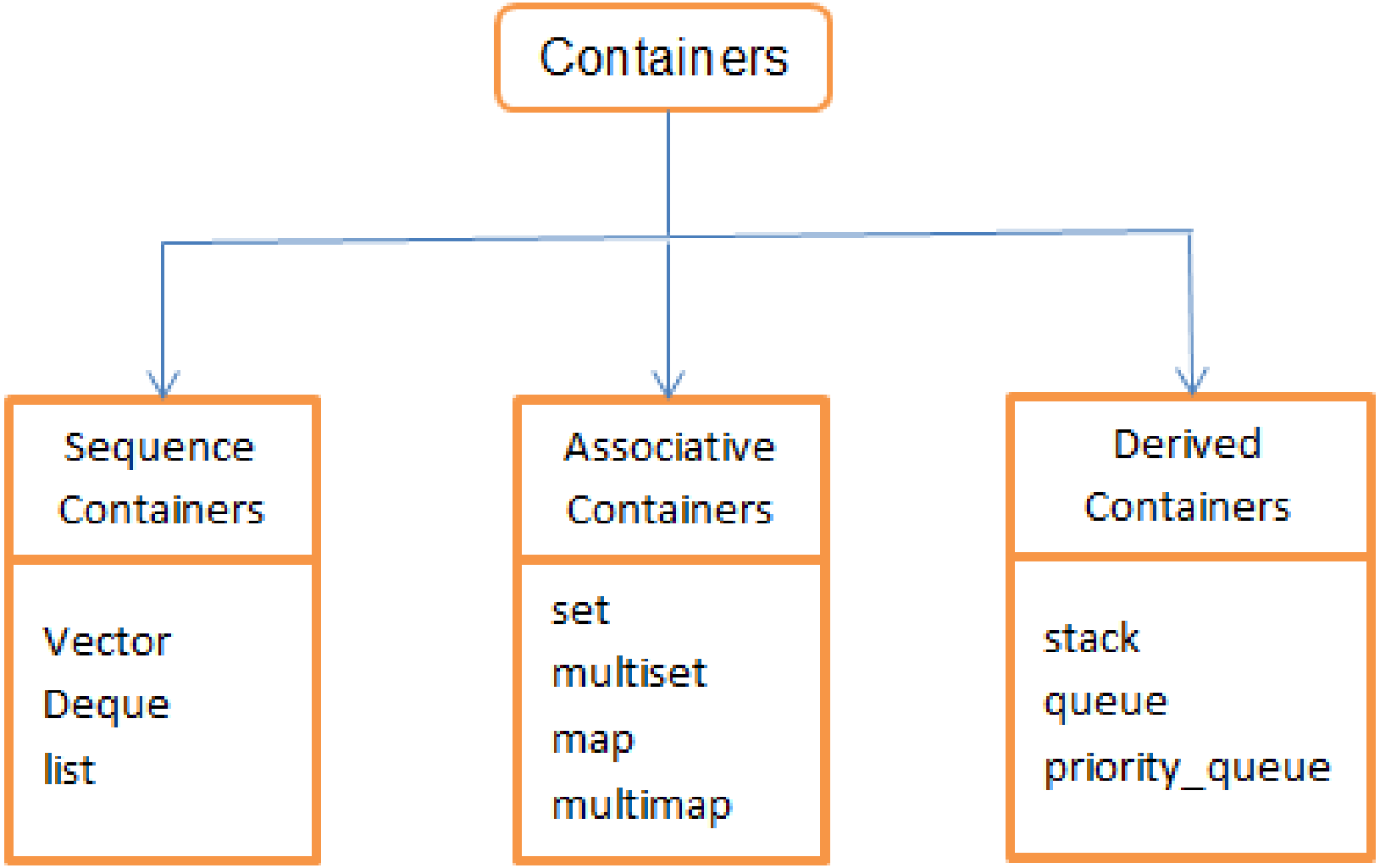


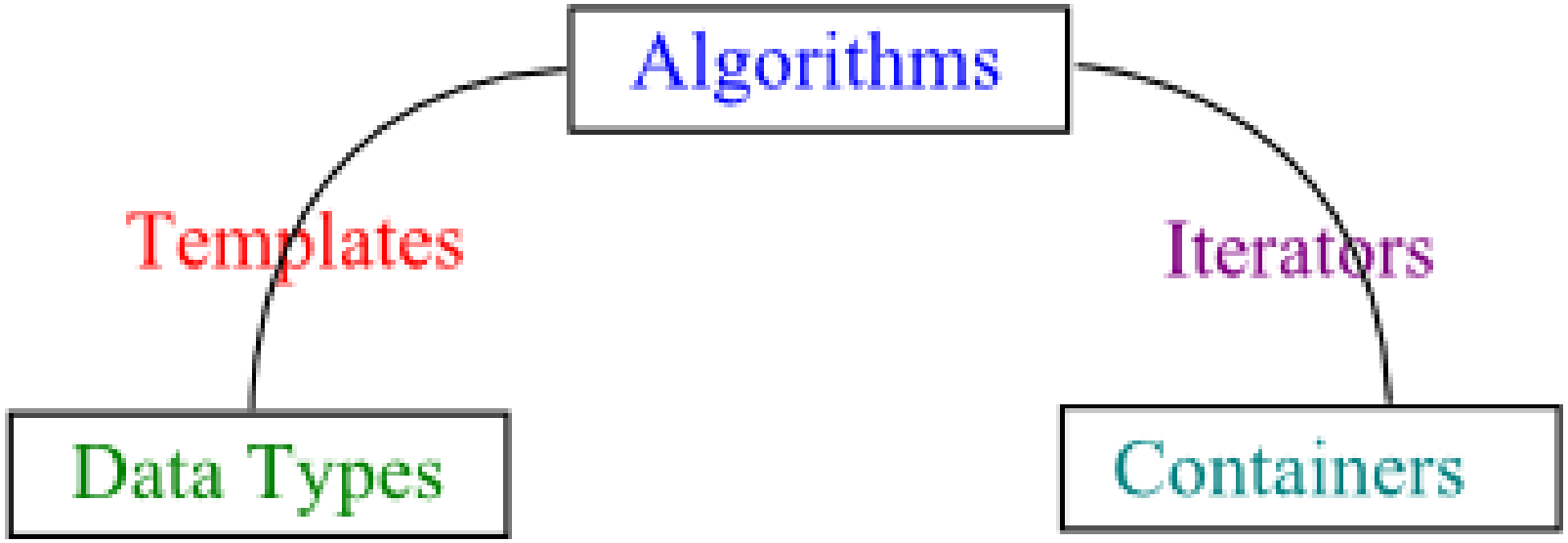
- STL including containers and the algorithms
- Strings
- I/O Streams

- Move semantic
- Unified initialization
- `auto` and `decltype`
- Lambda functions
- Multithreading
- Regular expressions
- Smart pointers
- Hash tables
- `std::array`

- Reader-writer locks
- Generalized lambdas

- Fold expressions
- constexpr if
- Initializers in `if` and `switch` statements
- Structured binding declarations
- Template deduction of constructors
- Guaranteed copy elision
- `auto_ptr` and trigraphs removed
- `string_view`
- Parallel algorithm of the STL
- The filesystem library
- `std::any`
- `std::optional`
- `std::variant`

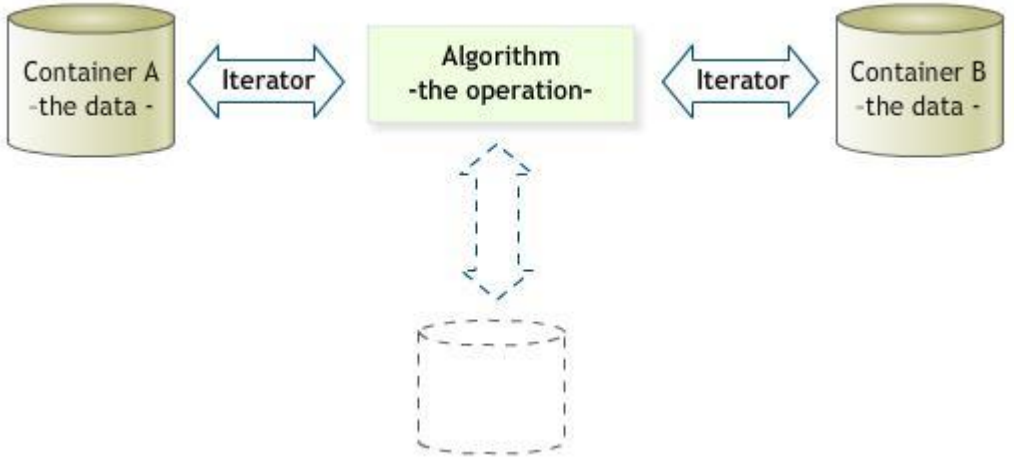

Containers
Sequence Containers
Vector
Deque
list
Associative Containers
set
multiset
map
multimap
Derived Containers
stack
queue
priority_queue

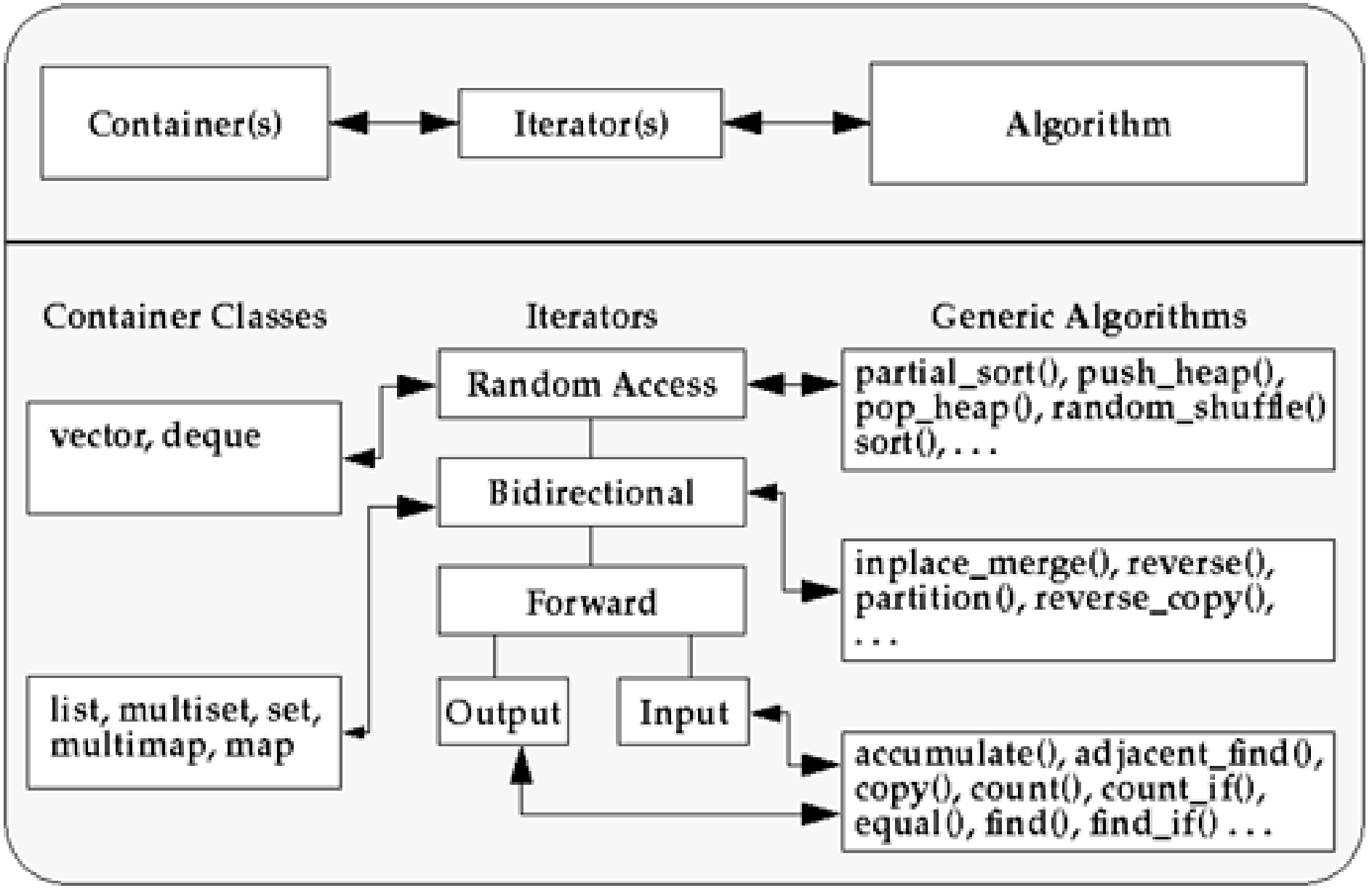


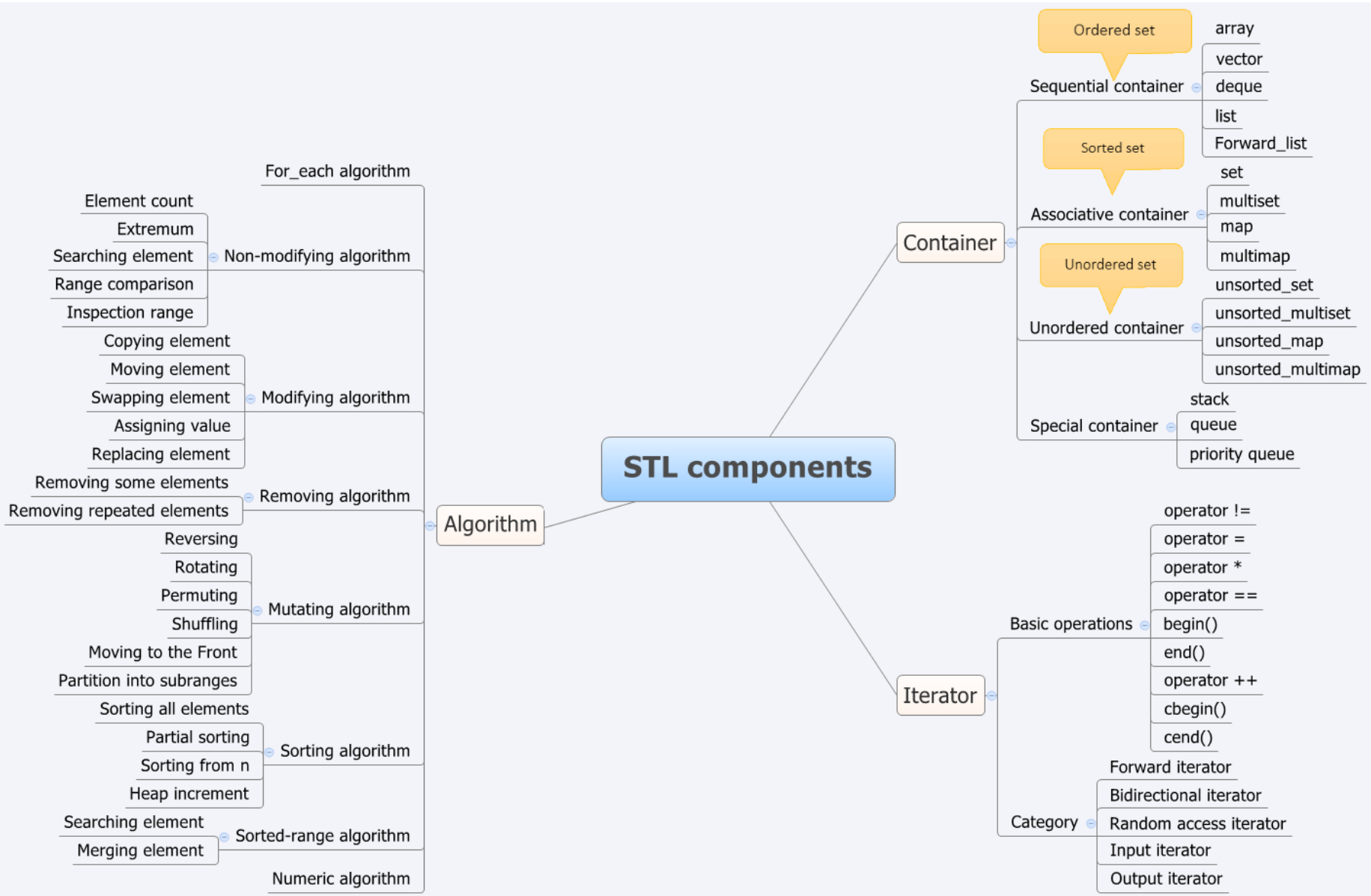
1. Templates
   make algorithms independent of the data types
2. Iterators
   make algorithms independent of the containters

What Is Algorithms?

- Used to process the elements of collections. For example, algorithms can search, sort and modify. Algorithms use iterators. Thus, an **algorithm** has to be written only once to work with arbitrary containers because the iterator interface for iterators is common for all container types.
- We can use a general algorithm to suit our needs even if that need is very special or complex. You will find in the program examples later, most of the member functions for processing the elements or data are common for various containers.
- **The data and operations in STL are decoupled.** Container classes manage the data, and the operations are defined by the algorithms, used together with the iterators.
- **Conceptually, iterators are the linker between these two components.** They let any algorithm interact with any container, graphically shown below.




Container(s)
Iterator(s)
Algorithm
Container Classes
Iterators
Generic Algorithms
Random Access
partial_sort(), push_heap(), pop_heap(), random_shuffle() sort(), . . .
vector, deque
Bidirectional
inplace_merge(), reverse(), partition(), reverse_copy(), . . .
Forward
list, multiset, set, multimap, map
Output
Input
accumulate(), adjacent_find(), copy(), count(), count_if(), equal(), find(), find_if() . . .


STL components
Container
Ordered set
Sequential container
array
vector
deque
list
Forward_list
Sorted set
Associative container
set
multiset
map
multimap
Unordered set
Unordered container
unsorted_set
unsorted_multiset
unsorted_map
unsorted_multimap
Special container
stack
queue
priority queue
Iterator
Basic operations
operator !=
operator =
operator *
operator ==
begin()
end()
operator ++
cbegin()
cend()
Category
Forward iterator
Bidirectional iterator
Random access iterator
Input iterator
Output iterator
Algorithm
For_each algorithm
Non-modifying algorithm
Element count
Extremum
Searching element
Range comparison
Inspection range
Modifying algorithm
Copying element
Moving element
Swapping element
Assigning value
Replacing element
Removing algorithm
Removing some elements
Removing repeated elements
Mutating algorithm
Reversing
Rotating
Permuting
Shuffling
Moving to the Front
Partition into subranges
Sorting algorithm
Sorting all elements
Partial sorting
Sorting from n
Heap increment
Sorted-range algorithm
Searching element
Merging element
Numeric algorithm

Header files

- **Containers**

<vector> <list> <deque>
<queue> <stack> <map>
<set> <bitset>

- **General utilities**

<utility> <functional>
<memory> <ctime>

- **Iterators**

<iterator>

- **Algorithms**

<algorithm> <cstdlib>

- **Diagnostics**

<stdexcept> <cassert>
<cerrno>

- **Strings**

<string> <cctype>
<cwtype> <cstring>
<cwstring> <cstdlib>

LECTURE 5

Standard Template Library – General Issues

Performance

- Personal experience 1:
  - STL implementation was **40% slower** than hand-optimized version.
    - STL: used deque
    - Hand Coded: Used “circular buffer” array;
  - Spent several days debugging the hand-coded version.
  - In my case, not worth it.
  - Still have prototype: way to debug fast version.

Performance

- Personal experience 2
- Application with STL list ~5% slower than custom list.
- Custom list “intrusive”

```
struct foo {
  int a;
  foo * next;
};
```

- Can only put foo in one list at a time ☹

Pitfalls

- Accessing an invalid `vector<>` element.

```
vector<int> v;
v[100]=1;  // whoops!
```

- **Solutions:**
  - use `push_back()`
  - Preallocate with constructor.
  - Reallocate with `reserve()`
  - Check `capacity()`

Pitfalls

- Inadvertently inserting into map<>.

```
if (foo[”bob”]==1)
//silently created entry “bob”
```

  - Use count() to check for a key without creating a new entry.

```
if ( foo.count(“bob”)  )
```

Pitfalls

- Not using `empty()` on `list<>` .
  - Slow

```
if ( my_list.count() == 0 ) { ... }
```

  - Fast

```
if ( my_list.empty() ) {...}
```

Pitfalls

Using invalid iterator

```
list<int> L;
list<int>::iterator li;
li = L.begin();
L.erase(li);
++li;                    // WRONG
```

Use return value of erase to advance

```
li = L.erase(li);  // RIGHT
```

Common Compiler Errors

```
vector<vector<int>> vv;
```

missing space

lexer thinks it is a right-shift.

any error message with **`pair<...>`**

`map<a,b>` implemented with **`pair<a,b>`**

STL versus Java Containters

STL	Java Containers
•Holds any type	•Holds things derived from Object
•No virtual function calls	•Virtual Function Call overhead
•Static type-checking	•No Static type-checking

More Generic Programming

- GTL : Graph Template Library
- BGL : Boost Graph Library
- MTL : Matrix Template Library
- ITL : Iterative Template Library

LECTURE 6

STL Generic Data Structures (Containers)

Standard Template Library

- The standard template library (STL) contains
  - Containers
  - Algorithms
  - Iterators
- A ***container*** is a way that stored data is organized in memory, for example an array of elements.
- ***Algorithms*** in the STL are procedures that are applied to containers to process their data, for example search for an element in an array, or sort an array.
- ***Iterators*** are a generalization of the concept of pointers, they point to elements in a container, for example you can increment an iterator to point to the next element in an array

The three parts of STL

- **Containers**
- Algorithms
- Iterators

Three types of containers

- Sequence containers
  - Linear data structures (vectors, linked lists)
  - First-class container
- Associative containers
  - Non-linear, can find elements quickly
  - Key/value pairs
  - First-class container
- Container adapters
  - Near containers
  - Similar to containers, with reduced functionality
- Containers have some common functions

STL Container Classes

- Sequence containers
  - `vector`
  - `deque`
  - `list`
- Associative containers
  - `set`
  - `multiset`
  - `map`
  - `multimap`
- Container adapters
  - `stack`
  - `queue`
  - `priority_queue`

Common STL Member Functions

- Member functions for all containers
  - Default constructor, copy constructor, destructor
  - **`empty`**
  - **`max_size`**, **`size`**
  - **`= < <= > >= == !=`**
  - **`swap`**
- Functions for first-class containers
  - **`begin`**, **`end`**
  - **`rbegin`**, **`rend`**
  - **`erase`**, **`clear`**

Common STL typedefs

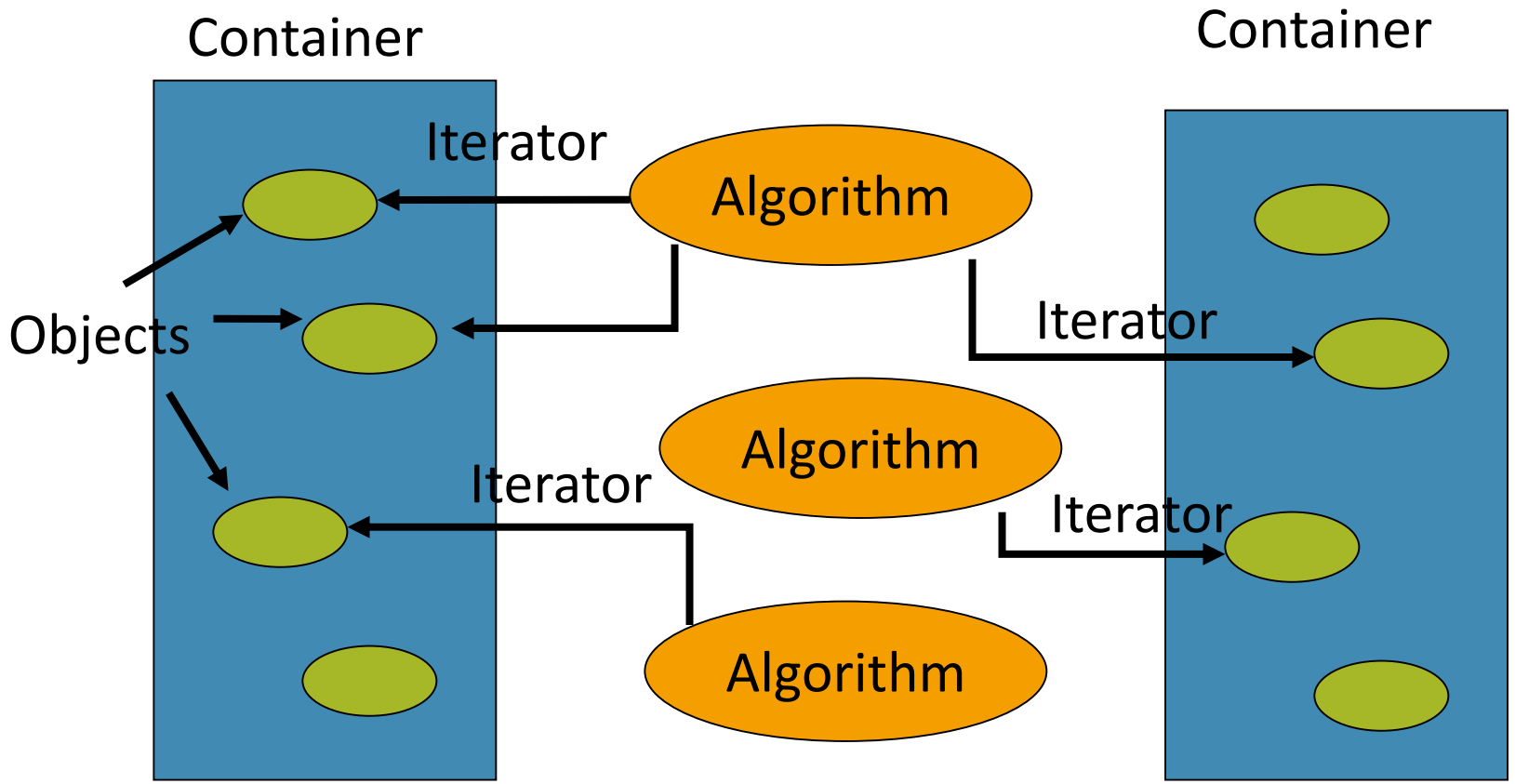
- `typedef`s for first-class containers
  - `value_type`
  - `reference`
  - `const_reference`
  - `pointer`
  - `iterator`
  - `const_iterator`
  - `reverse_iterator`
  - `const_reverse_iterator`
  - `difference_type`
  - `size_type`

LECTURE 7

Three Container Types

Containers, Iterators, Algorithms

- Algorithms use iterators to interact with objects stored in containers



Containers

A container is a way to store data, either built-in data

types like int and float, or class objects

The STL provides several basic kinds of containers

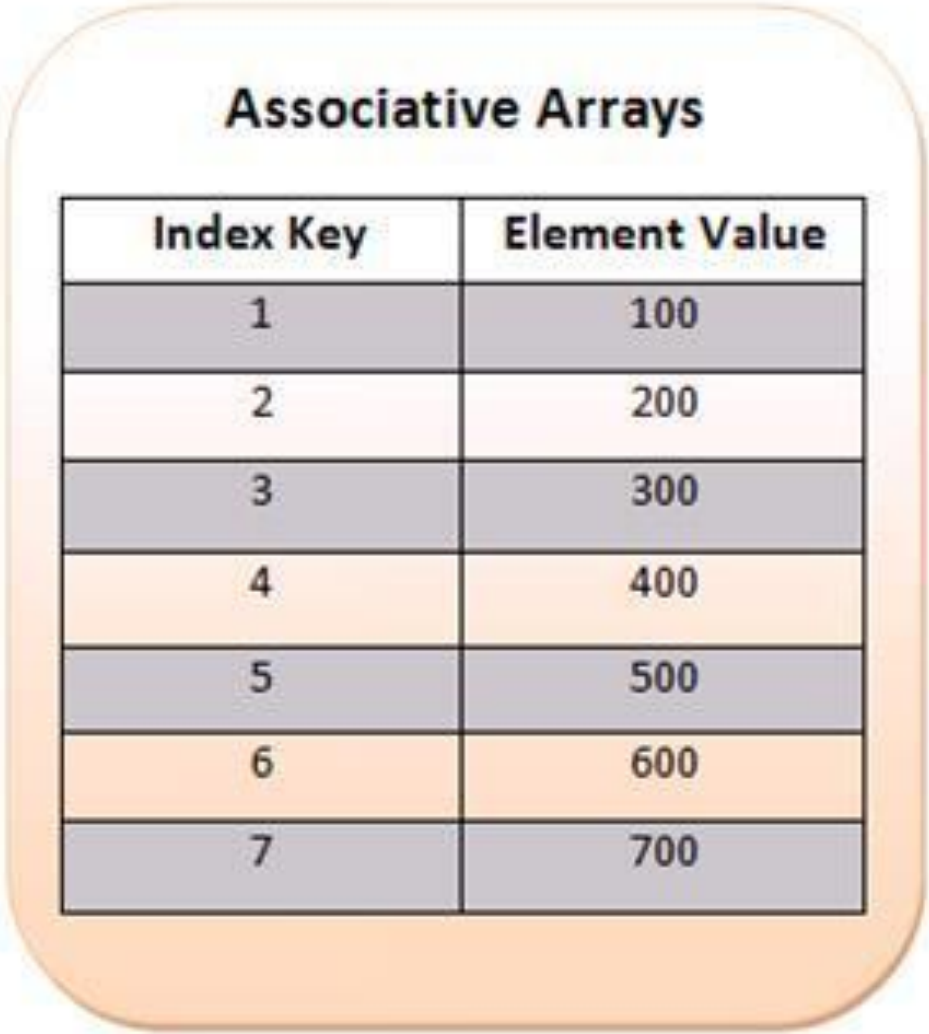
- <vector> : one-dimensional array
- <list> : double linked list
- <deque> : double-ended queue
- <queue> : queue
- <stack> : stack
- <set> : set
- <map> : associative array

Sequence Containers

- A sequence container stores a set of elements in sequence, in other words each element (except for the first and last one) is preceded by one specific element and followed by another, <vector>, <list> and <deque> are sequential containers
- In an ordinary C++ array the size is fixed and cannot change during run-time, it is also tedious to insert or delete elements.
- Advantage: quick random access <vector> is an expandable array that can shrink or grow in size, but still has the disadvantage of inserting or deleting elements in the middle.

Sequence Containers

- <list> is a double linked list (each element has points to its successor and predecessor), it is quick to insert or delete elements but has slow random access
- <deque> is a double-ended queue, that means one can insert and delete elements from both ends, it is a kind of combination between a stack (last in
- first out) and a queue (first in first out) and constitutes a compromise between a <vector> and a <list>



Associative Containers

- An associative container is non-sequential but uses a *key* to access elements. The keys, typically a number or a string, are used by the container to arrange the stored elements in a specific order, for example in a dictionary the entries are ordered alphabetically.

Associative Containers

- A <set> stores a number of items which contain keys The keys are the attributes used to order the items, for example a set might store objects of the class Person which are ordered alphabetically using their name.
- A <map> stores pairs of objects: a key object and an associated value object. A <map> is somehow similar to an array except instead of accessing its elements with index numbers, you access them with indices of an arbitrary type.
- <set> and <map> only allow one key of each value, whereas <multiset> and <multimap> allow multiple identical key values

Containers and “almost containers”

Sequence containers
- **vector, list, deque**

Associative containers
- **map, set, multimap, multiset**

“almost containers”
- array, **string, stack, queue, priority_queue, bitset**

New C++11 standard containers
- **unordered_map** (a hash table), **unordered_set,** …

For anything non-trivial, consult documentation
- Online
  - SGI, RogueWave, Dinkumware
- Other books
  - Stroustrup: The C++ Programming language 4th ed. (Chapters 30-33, 40.6)
  - Austern: Generic Programming and the STL
  - Josuttis: The C++ Standard Library

Sample Code	Operation
con, con1 and con2 are containers.	
ContainerType con e.g. vector<int> vec0	Creates an empty container without any element.
ContainerType con1(con2) e.g. vector<int> vec0(vec1)	Copies a container of the same type.
ContainerType con(begin,end) e.g. vector<int> vec0(p.begin(),p.end())	Creates a container and initializes it with copies of all elements of [begin, end).
con.~*ContType*()	Deletes all elements and frees the memory.
con.size()	Returns the actual number of elements.
con.empty()	Returns whether the container is empty, equivalent to size()==0, but might be faster.
con.max_size()	Returns the maximum number of elements possible.
con1 == con2	Returns whether con1 is equal to con2.
con1 != con2	Returns whether con1 is not equal to con2, equivalent to !(con1==con2)
con1 < con2	Returns whether con1 is less than con2
con1 > con2	Returns whether con1 is greater than con2, equivalent to con2 < con1.
con1 <= con2	Returns whether con1 is less than or equal to con2, equivalent to !(con2<con1).
con1 >= con2	Returns whether con1 is greater than or equal to con2, equivalent to !(con1<con2).
con1 = con2	Assignment, assigns all elements of con1 to con2.

Sample Code	Operation
con, con1 and con2 are containers.	
con1.swap(con2)	Swaps the data of con1 and con2.
swap(con1,con2)	Same but a global function.
con.begin()	Returns an iterator for the first element.
con.end()	Returns an iterator for the position after the last element.
con.rbegin()	Returns a reverse iterator for the first element of a reverse iteration.
con.rend()	Returns a reverse iterator for the position after the last element of a reverse iteration.
con.insert(position,element)	Inserts a copy of element.
con.erase(begin,end)	Removes all elements of the range [begin, end), some containers return next element not removed.
con.clear()	Removes all elements, making the container empty.
con.get_allocator()	Returns the memory model of the container.

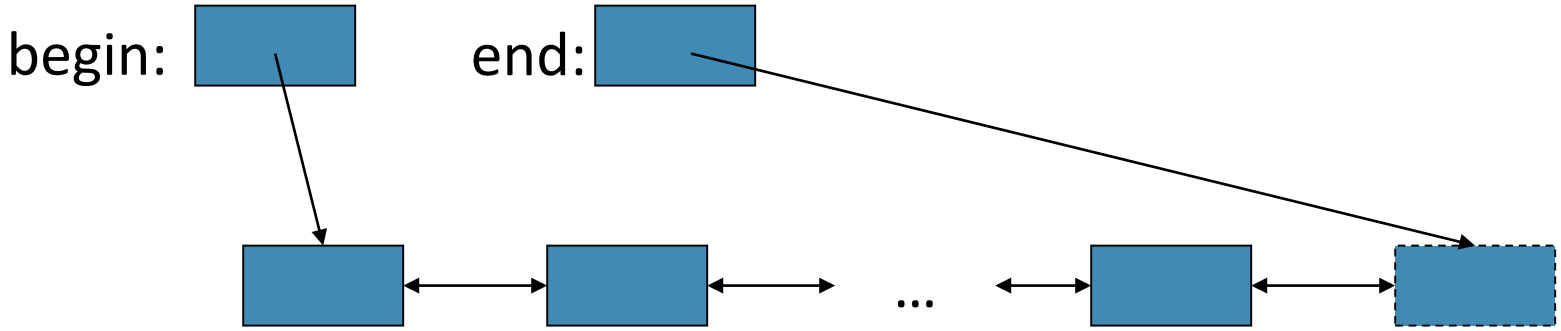
LECTURE 8

Iterators

Basic model

A pair of iterators defines a sequence

- The beginning (points to the first element – if any)
- The end (points to the one-beyond-the-last element)



- An iterator is a type that supports the "iterator operations" of
  - ++ Point to the next element
  - * Get the element value
  - == Does this iterator point to the same element as that iterator?
- Some iterators support more operations *(e.g.*, --, +, and [ ])

Introduction to Iterators

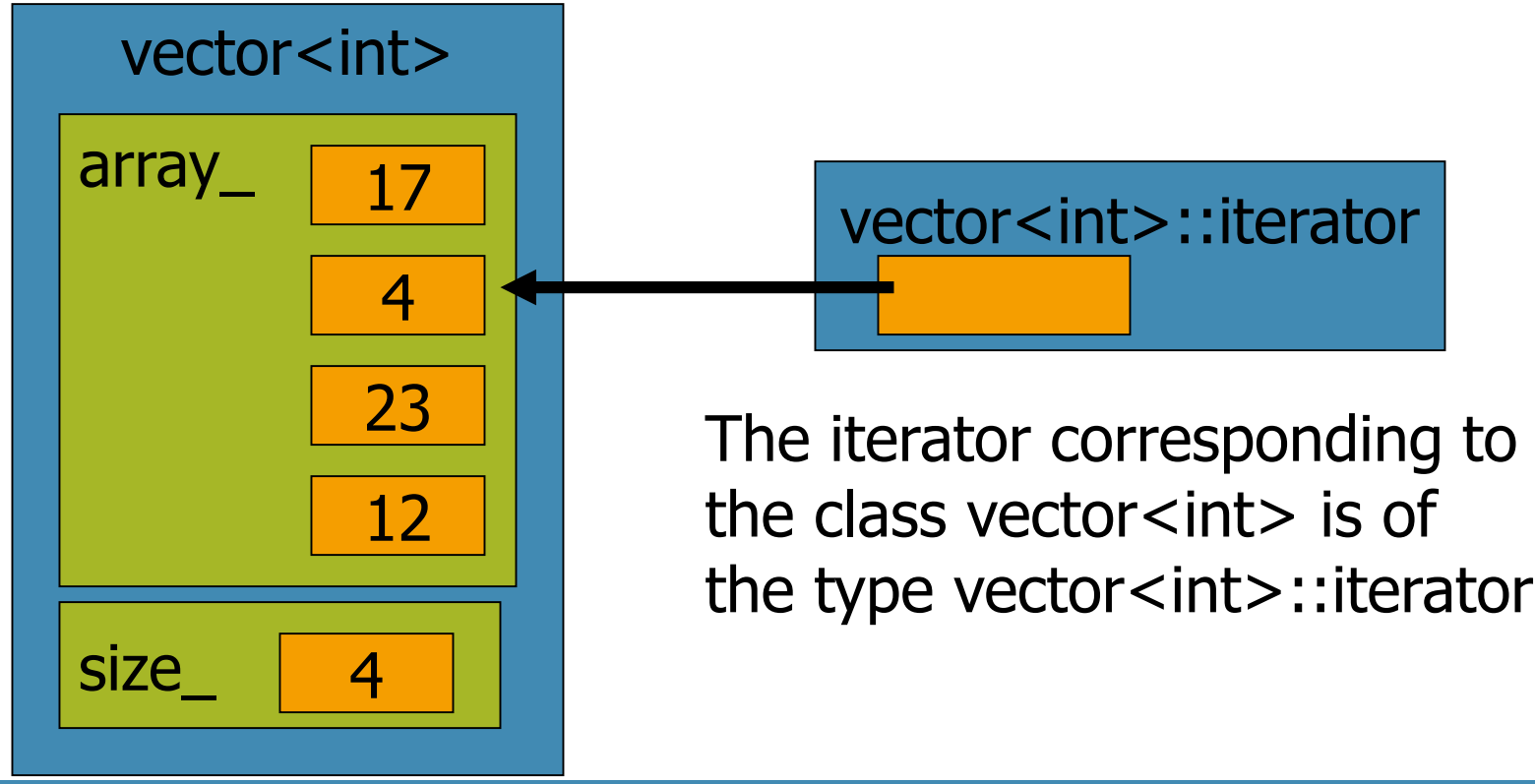
- Iterators similar to **pointers**
  - Point to first element in a container
  - Iterator operators same for all containers
    - `*` dereferences
    - `++` points to next element (equivalent to next() function in Java)
    - `begin()` returns iterator to first element
    - `end()` returns iterator to last element
  - Use iterators with sequences (ranges)
    - Containers
    - Input sequences: `istream_iterator`
    - Output sequences: `ostream_iterator`

Introduction to Iterators

- **Usage**
  - `std::istream_iterator< int > inputInt( cin )`
    - Can read input from `cin`
    - `*inputInt`
      - Dereference to read first `int` from `cin`
    - `++inputInt`
      - Go to next `int` in stream
  - `std::ostream_iterator< int > outputInt(cout)`
    - Can output `int`s to `cout`
    - `*outputInt = 7`
      - Outputs 7 to `cout`
    - `++outputInt`
      - Advances iterator so we can output next `int`

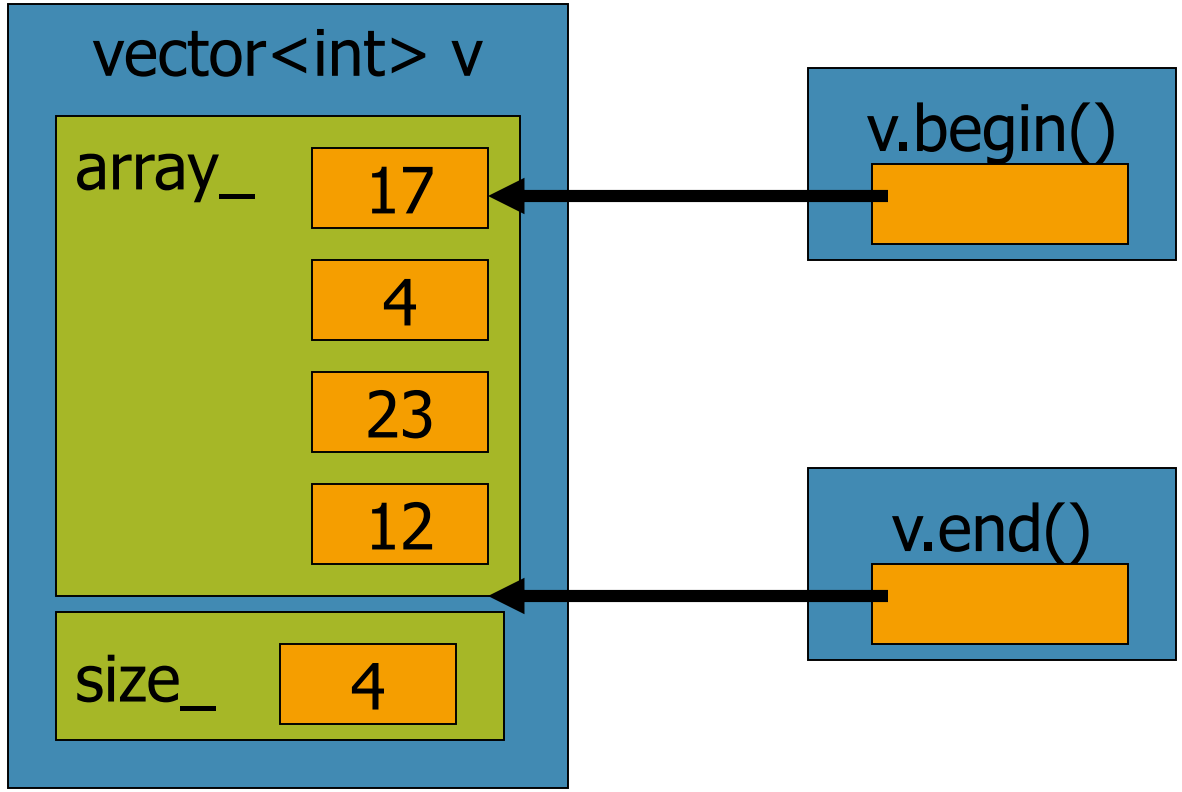
Iterators

- Iterators are pointer-like entities that are used to access individual elements in a container.
- Often they are used to move sequentially from element to element, a process called *iterating* through a container.



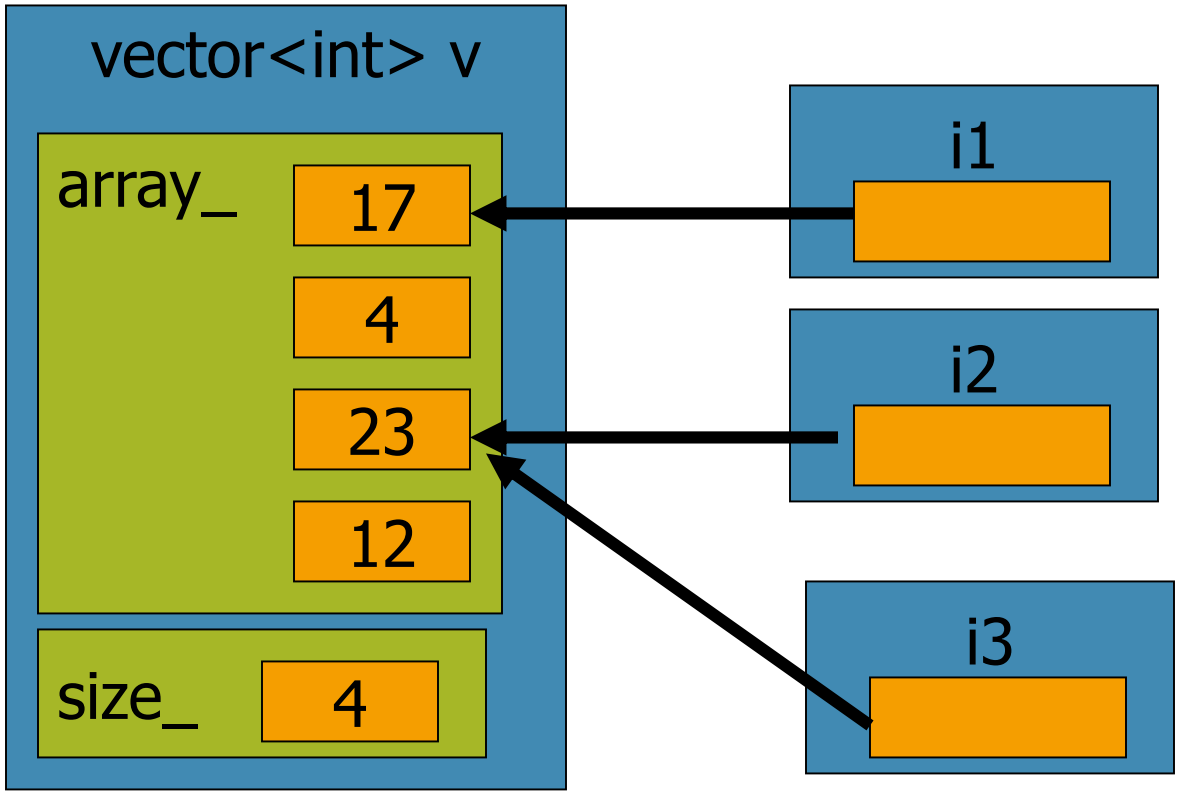
Iterators

- The member functions begin() and end() return an iterator to the first and past the last element of a container



Iterators

- One can have multiple iterators pointing to different or identical elements in the container



Iterator Categories

- Input
  - Read elements from container, can only move forward
- Output
  - Write elements to container, only forward
- Forward
  - Combines input and output, retains position
  - Multi-pass (can pass through sequence twice)
- Bidirectional
  - Like forward, but can move backwards as well
- Random access
  - Like bidirectional, but can also jump to any element

Iterator Types Supported

- Sequence containers
  - **`vector`**: random access
  - **`deque`**: random access
  - **`list`**: bidirectional
- Associative containers (all bidirectional)
  - **`set`**
  - **`multiset`**
  - **`Map`**
  - **`multimap`**
- Container adapters (no iterators supported)
  - **`stack`**
  - **`queue`**
  - **`priority_queue`**

Iterator Operations

- All
  - `++p, p++`
- Input iterators
  - `*p`
  - `p = p1`
  - `p == p1, p != p1`
- Output iterators
  - `*p`
  - `p = p1`
- Forward iterators
  - Have functionality of input and output iterators

Iterator Operations

- Bidirectional
  - `--p, p--`
- Random access
  - `p + i, p += i`
  - `p - i, p -= i`
  - `p[i]`
  - `p < p1, p <= p1`
  - `p > p1, p >= p1`

LECTURE 9

Iterator Example

Iterators

Demo Program: fifth_iterator.cpp

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4  int main(){
5    int arr[] = { 12, 3, 17, 8 };  // standard C array
6    vector<int> v(arr, arr+4);  // initialize vector with C array
7    for (vector<int>::iterator i=v.begin(); i!=v.end(); i++) {
8      cout << *i << " ";
9    }
10   cout << endl;
11   return 0;
12 }
```

Go Notepad++!!!

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\iterator>fifth_iterator
12 3 17 8
```

Iterators

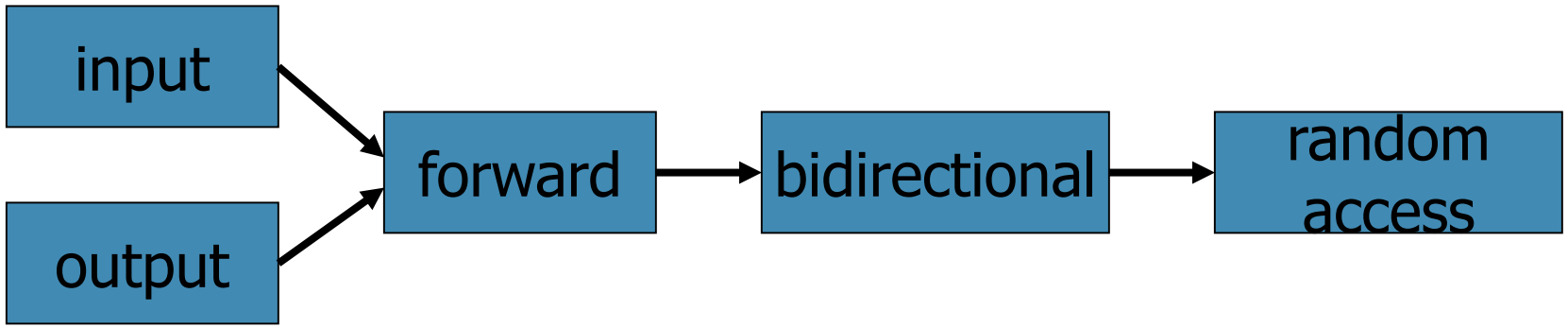
Demo Program: sixth_iterator.cpp

- Operating the iterator.
- Iterator is an pointer.
- Iterable object can be used for **for-each** loop using iterators.

```cpp
#include <vector>
#include <iostream>
using namespace std;
int max(vector<int>::iterator start, vector<int>::iterator stop) {
   int m=*start;
   while(start != stop)
    {
       if (*start > m)
         m=*start;
       ++start;
    }
    return m;
}
int main(){
  int arr[] = { 12, 3, 17, 8 };                  // standard C array
  vector<int> v(arr, arr+4);                     // initialize vector with C array
  vector<int>::iterator iter=v.begin();          // iterator for class vector
  // define iterator for vector and point it to first element of v
  cout << "first element of v=" << *iter;        // de-reference iter
  iter++;                                        // move iterator to next element
  iter=v.end()-1;                                // move iterator to last element
  cout << "max of v = " << max(v.begin(),v.end()) << endl ;
  return 0;
}
```

Iterator Categories

- Not every iterator can be used with every container for example the list class provides no random access iterator
- Every algorithm requires an iterator with a certain level of capability for example to use the [] operator you need a random access iterator
- Iterators are divided into five categories in which a higher (more specific) category always subsumes a lower (more general) category, e.g. An algorithm that accepts a forward iterator will also work with a bidirectional iterator and a random access iterator



Demo Program: first_iterator.cpp

```cpp
#include <iostream>
using namespace std;

#include <iterator>  // ostream_iterator and istream_iterator
int main(){
    cout << "Enter two integers: ";

    // create istream_iterator for reading int values from cin
    std::istream_iterator< int > inputInt( cin );

    int number1 = *inputInt; // read int from standard input
    ++inputInt;         // move iterator to next input value
    int number2 = *inputInt;  // read int from standard input
    // create ostream_iterator for writing int values to cout
    std::ostream_iterator< int > outputInt( cout );

    cout << "The sum is: ";
    *outputInt = number1 + number2;  // output result to cout
    cout << endl;
     return 0;
} // end main
```

Go Notepad++!!!

Note creation of **`istream_iterator`**. For compilation reasons, we use **`std::`** rather than a **`using`** statement.

Access and assign the iterator like a pointer.

Create an **`ostream_iterator`** is similar. Assigning to this iterator outputs to **`cout`**.

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\iterator>first_iterator
Enter two integers: 3 35
The sum is: 38
```

LECTURE 10

Functions for Iterators

Introduction to Iterator-based Algorithms

- STL has algorithms used generically across containers
  - Operate on elements indirectly via iterators
  - Often operate on sequences of elements
    - Defined by pairs of iterators
    - First and last element
  - Algorithms often return iterators
    - **`find()`**
    - Returns iterator to element, or **`end()`** if not found
  - Premade algorithms save programmers time and effort

For_Each() Algorithm

Demo Program: third_iterator.cpp

```
1  #include <vector>
2  #include <algorithm>
3  #include <iostream>
4  using namespace std;
5  void show(int n) {
6    cout << n << " ";
7  }
8  int main(){
9     int arr[] = { 12, 3, 17, 8 };  // standard C array
10    vector<int> v(arr, arr+4);  // initialize vector with C array
11   for_each (v.begin(), v.end(), show); // apply function show
12            // to each element of vector v
13   return 0;
14 }
```

Go Notepad++!!!

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\iterator>third_iterator
12 3 17 8
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\iterator>_
```

Find() Algorithm

Demo Program: second_iterator.cpp

```
1  #include <vector>
2  #include <algorithm>
3  #include <iostream>
4  using namespace std;
5  int main(){
6    int key;
7    int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C array
8    vector<int> v(arr, arr+7);  // initialize vector with C array
9    vector<int>::iterator iter;
10   cout << "enter value :";
11   cin >> key;
12   iter=find(v.begin(),v.end(),key); // finds integer key in v
13   if (iter != v.end()) // found the element
14     cout << "Element " << key << " found" << endl;
15   else
16     cout << "Element " << key << " not in vector v" << endl;
17   return 0;
18 }
```

Go Notepad++!!!

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\iterator>second_iterator
enter value :3
Element 3 found
```

Count_If() Algorithm

Demo Program: fourth_iterator.cpp

```
1  #include <vector>
2  #include <algorithm>
3  #include <iostream>
4  using namespace std;
5  bool mytest(int n) { return (n>14) && (n <36); };
6  int main(){
7   int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C array
8   vector<int> v(arr, arr+7);  // initialize vector with C array
9   int n=count_if(v.begin(),v.end(),mytest);
10    // counts element in v  for which mytest is true
11  cout << "found " << n << " elements" << endl;
12  return 0;
13 }
```

Go Notepad++!!!

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\iterator>fourth_iterator
found 2 elements
```