TDDE18 & 726G77

Standard Templated Library – Algorithms

Different types of iterator

- Single pass iterator can only advance over the list a single element at a time, and once an item has been iterated, it will never be iterated again.
- Multi-pass iterators can “go back” to previous character, but you might not be able to do so from the iterator object itself

Single-pass and multi-pass iterators

Single-pass iterators	Multi-pass iterators
InputIterator	ForwardIterator
OutputIterator	BidirectionalIterator
	RandomAccessIterator

Single pass iterators

- InputIterator
  - Can read from the pointed-to element
  - Only guarantee validity for single pass algorithms
- OutputIterator
  - Can write to the pointed-to element
  - Only guarantee validity for single pass algorithms

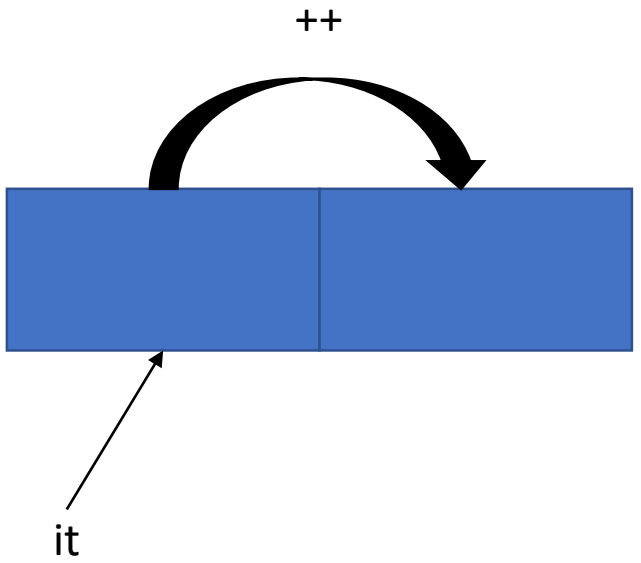
Multi-pass iterators

- ForwardIterator
  - Can read data from pointed-to element
  - Can be used in multipass algorithms
- BidirectionalIterator
  - Is a ForwardIterator in both directions
  - Can be incremented and decremented
- RandomAccessIterator
  - Is a BirectionalIterator
  - Can be moved to point to any element in constant time

ForwardIterator

- be dereferenced
- be incremented
- be compared with another iterator

```
// dereferencing
*it
it->
//incrementing
++it
it++
//compared
it == other_it
it != other_it
```

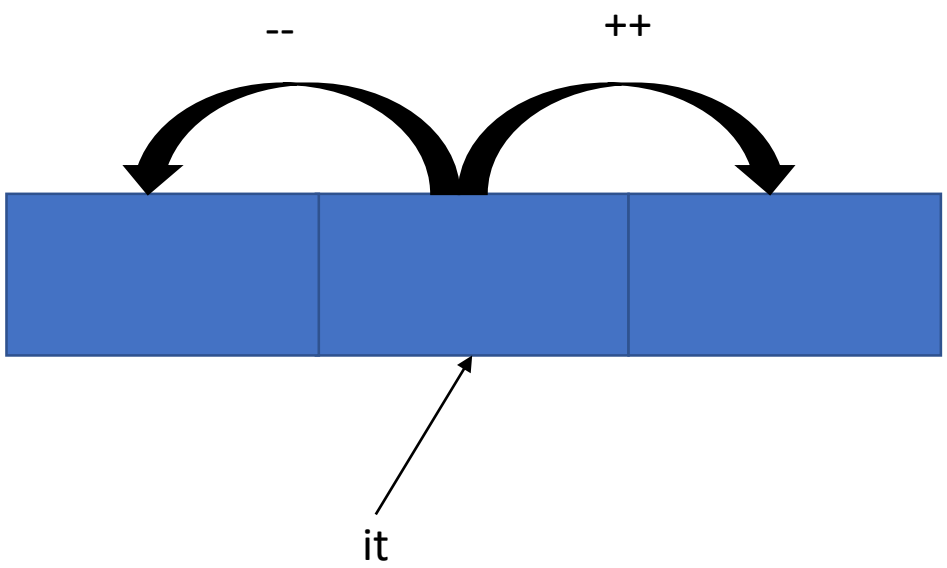


BidirectionalIterator

- Is a ForwardIterator
- With the added ability to decrement

// decrement

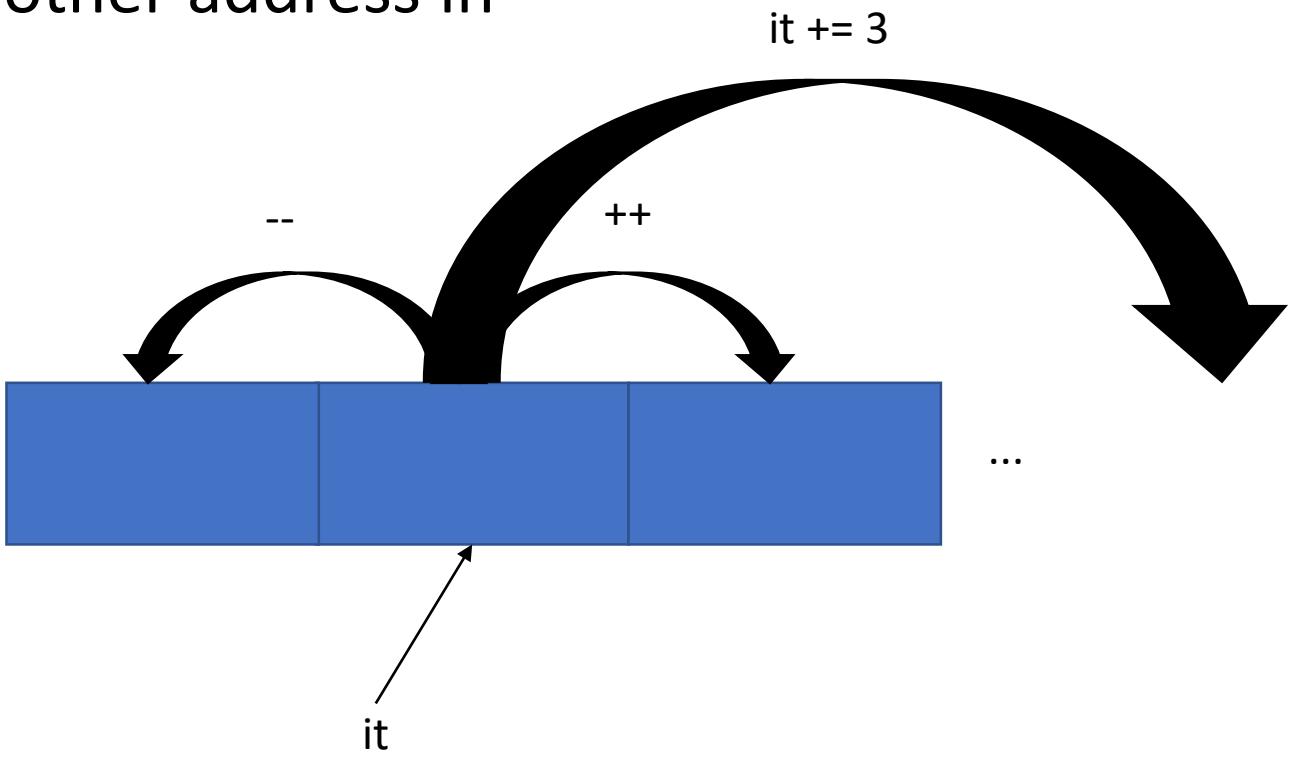
```
it--
--it
```



RandomAccessIterator

- Is a BidirectionalIterator
- With the ability to jump to another address in constant time

```
// Random Access
it += 3
it -= 5
```



Containers and their iterator type

ForwardIterator	Bidirectional	RandomAccess
forward_list	list	vector
	map	string
	set	

Algorithm requires different iterator type

std::sort

Defined in header <algorithm>

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );
```
(1)

std::min_element

Defined in header <algorithm>

```
template< class ForwardIt >
ForwardIt min_element( ForwardIt first, ForwardIt last );
```

Different types of functions

- STL algorithms uses different types of functions (function object) as an input argument.
  - UnaryOperation
  - BinaryOperation
  - Predicate
  - Comparison

std::min_element

```
template< class ForwardIt, class Compare >
ForwardIt min_element( ForwardIt first, ForwardIt last, Compare comp );
```

std::copy_if

```
template< class InputIt, class OutputIt, class UnaryPredicate >
OutputIt copy_if( InputIt first, InputIt last,
                  OutputIt d_first,
                  UnaryPredicate pred );
```

UnaryOperation

- UnaryOperation – unary operation function object that will be applied.

```
Ret fun(T [const&] a);      // signature
```

Ret must be a type that OutputIterator can reference to
const& are optional

```
char upperChar(char c) {
    return std::toupper(c);
}
```

BinaryOperation

- BinaryOperation – binary operation function object will be applied

```
Ret fun(T1 [const&] a, T2 [const&] b);     // signature
```

Ret must be a type that OutputIterator can reference to

const& are optional

```
int sum(int i, int j) {
    return i + j;
}
```

Predicate

- Predicate – returns true for the required elements

```
bool pred(T [const&] a);    // signature

const& are optional
```

```
bool less_than_five(int a) {
    return a < 5;
}
```

Comparison

- Comparison function object – which returns *true* if the first argument is *less* than (i.e is ordered *before*) the second

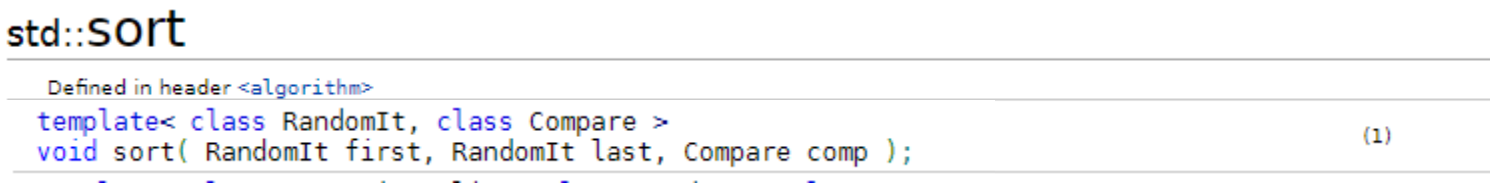
```
bool cmp(T1 [const&] a, T2 [const&] b);      // signature

const& are optional
```

```
bool larger(int a, int b) {
    return a > b;
}
```

How to use this?

```
bool larger(int a, int b) {
    return a > b;
}
```



vector<int> a{3, 4, 5, 6, 7, 8};

sort(begin(a), end(a), larger);

std::transform

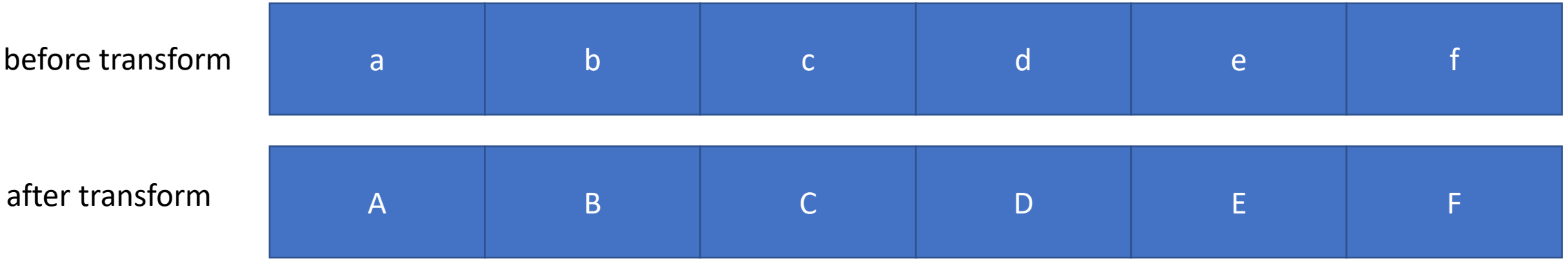
- transform applies the given function Operation operation to a range and store the result in another range, beginning at d_first

```
ForwardIterator transform(InputIterator first, InputIterator last, OutputIterator
d_first, UnaryOperation operation);
```

std::tranform

```
char toUpper(char c) {
    return std::toupper(c);
}

string s{“abcdef”};
transform(begin(s), end(s), begin(s), toUpper);
```



std::for_each

- Applies the given function object f to the result of dereferencing every iterator in the range [first, last), in order

```
void for_each(InputIterator first, InputIterator last, UnaryFunction f);
```

std::for_each

```
void print_out(int n) {
        cout << “ “ << n;
}

set<int> s{5, 4, 3, 99, 0, 1, 2};
for_each(begin(s), end(s), print_out);
```

Iterator adaptors for streams (1)

- Sometimes a class have the functionality you seek but not the right interface for accessing that functionality.
  - copy() algorithm requires a pair of input iterators as its first two parameters.
- An istream object can act as a source of such data values but it does not have any iterators that the copy algorithm can use.

Iterator adaptors for streams (2)

- Sometimes a class have the functionality you seek but not the right interface for accessing that functionality.
  - copy() algorithm also have a version where it takes in three arguments. The third of which is an output iterator that directs the copied values to their proper destination.
- An ostream object can act as a destination of such data values but output streams do not directly provide any output iterator

Iterator adaptors for streams (3)

- An adaptor class is one that acts like a “translator” by “adapting” the messages you want to send to produce messages that the other class object wants to receive.
- Iterator adaptors that iterates through streams (filestream, standard input/output etc)
  - istream_iterator – provides the interface that the copy() algorithm expects for input
  - ostream_iterator – provides the interface that the copy() algorithm expects for output

ostream_iterator

- Is a single-pass iterator that writes successive object of type T
- Writes to ostream by calling operator<<
- Optional delimiter string is written to the output stream after every write.

ostream_iterator

```
vector<int> v{1, 2, 3, 4, 5};
ostream_iterator<int> oos{cout, “ “};
copy(begin(v), end(v), oos);
```

istream_iterator

- Single-pass input iterator that reads successive object of type T
- Read from an istream object by calling operator>>
- Default constructor is known as the *end-of-stream* iterator.

istream_iterator

```
istream_iterator<int> iis{cin};
istream_iterator<int> eos{};

ostream_iterator<int> oos{cout, " "};

copy(iis, eos, oos);
```

Iterator adaptors for insertion (1)

- Inserters (also called “insert iterators”) are “iterator adaptors” that permit algorithms to operate in insert mode rather than overwrite mode.
- They solve the problem that crops up when an algorithm tries to write element to a destination container not already big enough to hold them.
- They make the container larger if needed

Iterator adaptors for insertion (2)

- There are three kinds of inserters
  - back_inserter – which can be used if the recipient container supports the push_back() member function
  - front_inserter – which can be used if the recipient container supports the push_front() member function
  - inserter – which can be used if the recipient container supports the insert() member function.

back_inserter

- Call the push_back function of the container

```
vector<int> v1{1, 2, 3, 4, 5, 6};
vector<int> v2{};

copy(begin(v1), end(v1), back_inserter(v2));
```

front_inserter

- Call the push_front member function of the container

```
vector<int> v{1, 2, 3, 4, 5, 6};
list<int> l{};

copy(begin(v), end(v), front_inserter(l));
```

Useful iterator functions

- C++ give you a set of general functions that works on all iterator that have those attributes
  - advance
  - distance
  - next
  - prev – only work on bidirectional and random access iterators

std::advance

- Increments the given iterator by n elements
- If n is negative, the iterator is decremented. The iterator must be a BidirectionalIterator

```
vector<int> v{ 3, 1, 4 };
auto vi{v.begin()};
advance(vi, 2);
cout << *vi << '\n';
```

std::next

- Return the n-th successor of iterator it

```
vector<int> v{ 3, 1, 4 };

auto it{v.begin()}
it = next(it, 2);
cout << *it << endl;

// print out 4
```

std::prev

- Return the n-th predecessor of iterator it
- Iterator must be BidirectionalIterator or RandomAccessIterator

```
vector<int> v{ 3, 1, 4 };
auto it{v.end()};
it = prev(it, 2);
cout << *it << endl;

// print out 1
```

std::distance

- Returns the number of hops from *first* to *last*
- The value may be negative if random-access iterators are used and first is reachable from last

```
difference_type distance(InputIterator first, InputIterator last);
```

```
vector<int> v{ 3, 1, 4 };
distance(begin(v), end(v));    // return 3
distance(end(v), begin(v));    // return -3
```

Lambda function

- Constructs an unnamed function object
- Able to capture variables in scope
- You can see this as an anonymous function

// empty lambda function that have no capture, no argument and nothing in function body

[](){}

// if you want to call the lambda function as is then add parentheses after

[](){}()

Lambda function – return type

- The return type is deduced from return statements

```
[]() { return 1; }                          // returns data type int
[](double d) { return d}                    // return data type double
[]() { return new Node; }                   // return data type Node *
[](Person & p) { p.updateName(“Sam”); } // return data type void
```

Lambda function – how to use

```
vector<int> v{1, 2, 3, 4, 5};

sort(begin(v), end(v), [](int a, int b) { return a > b; });

// equivalent to

bool larger(int a, int b) {
    return a > b;
}

sort(begin(v), end(v), larger);
```

Lambda function – capture variables

- Lambda functions cannot reach variables outside of its function body scope

```
vector<int> v{};

[] () { v.push_back(5); }()   // error v is not captured

[v]() { v.push_back(5); }()   // is a copy of v and its const

[=v]() { v.push_back(5); }()  // captures v by copy

[&v]() { v.push_back(5); }()  // captures v by reference
```

std::copy_if

- Copies the element in the range [first, last) to another range beginning at *d_first*. Only copies the element in the range for which the predicate *pred* returns true.
- The relative order of the elements that are copied is preserved.
- The behavior is undefined if the source and the destination ranges overlap.

```
ForwardIterator copy_if(ForwardIterator from_first, ForwardIterator from_last,
  ForwardIterator d_first, UnaryPredicate pred);
```

How to copy from cin to a vector all numbers that are larger than 5?