Lecture 11

STL Algorithms, functions object, lambda

TDDD86: DALP

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms*
October 9th, 2023

IDA, Linköping University

11.1

Content

Contents

11.2

1 STL Algorithms

1.1 Introduction

Challenge

Write a program that reads a list of integers from a file and write the average value of the numbers.

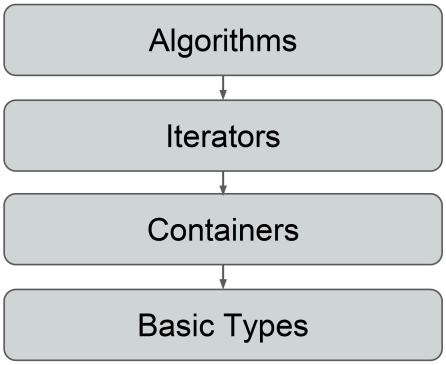
```
#include <iostream>
#include <fstream>
#include <set>
using namespace std;

int main() {
    ifstream input("data.txt");
    multiset<int> values;

    /* Read the data from the file. */
    int currValue;
    while (input >> currValue)
        values.insert(currValue);

    /* Compute the average. */
    double total = 0.0;
    for (multiset<int>::iterator itr = values.begin();
```

```
        itr != values.end(); ++itr)
        total += *itr;
    cout << "Average␣is:␣" << total / values.size() << endl;
}
```

11.3

Abstraction in STL

- **Iterators** let us abstract way from the underlying container
- Operations such as sorting, partitioning, filtering, search, etc. can be written to work with `vector`, `deque`, `set` or any other container
  - we call these operations **algorithms** in STL



11.4

1.2 `accumulate`

`accumulate`

- The second loop of the averaging program:

```
double total = 0.0;
for (multiset<int>::iterator itr = values.begin(); itr != values.end(); ++itr)
    total += *itr;
cout << total / values.size() << endl;
```

- This is functionally equivalent to:

```
cout << accumulate(values.begin(), values.end(), 0.0) / values.size() << endl;
```

- `accumulate` is defined in the header `<numeric>`
- `accumulate` takes three parameters
  - two iterators which define an interval on the elements to accumulate
    - can be any type of iterators that allow to read and iterate through the elements
  - an initial value to initialise the aggregation

```
// Calculate sum multiset elements between 42 (inclusive) and 137 (exclusive)
accumulate(values.lower_bound(42), values.upper_bound(137), 0);
```

11.5

Why STL Algorithms?

Why do we need such a complex abstraction model?

- Avoid duplication of code
- Write correct code
- Write effective code
- Write clear code

11.6

1.3 Naming convention

if

There are more than fifty algorithms in the STL (in `<algorithm>` and `<numeric>`)

- The suffix `_if` for an algorithm (t.ex. `replace_if`, `count_if`) means that the algorithm performs the task on the element only if they satisfy a given condition.
  - The condition is a predicate in the form of a unary function returning **`bool`**

```
// Print number of copies of the number 137 in vector<int> myVec
cout << count(myVec.begin(), myVec.end(), 137) << endl;

bool isEven(int value) {
    return value % 2 == 0;
}

// Print number of even numbers in myVec
cout << count_if(myVec.begin(), myVec.end(), isEven) << endl;
```

11.7

copy

- Algorithm name that contains `copy` (t.ex. `remove_copy`, `partial_sort_copy`, `unique_copy`) make a copy of a range of data and stores the result in the output

11.8

n

- Suffix `_n` for an algorithm (i.e. `generate_n`, `search_n`) performs an operation `n` times.

```
// Set every value in a deque to 0
fill(myDeque.begin(), myDeque.end(), 0);

// Set first ten values of deque to 0
fill_n(myDeque.begin(), 10, 0);
```

11.9

1.4 Reordering algorithms

sort

The most useful ordering algorithm is `sort` which sorts elements of a range in ascending order.

```
// Sort vector<int> from lowest to highest
sort(myVector.begin(), myVector.end());
```

- it requires random access iterator
  - `sort` hence it fails to work with `set`, or `map` (but this is already sorted!)
  - `list` has a `sort` member function
- by default, it uses the **`operator`**< for comparison, but a comparison can be specified
  - it should be returning a **`bool`**

```
struct placeT {
    int x;
    int y;
};

bool comparePlaces(const placeT& one, const placeT& two) {
    if(one.x != two.x)
        return one.x < two.x;
    return one.y < two.y;
}

sort(myPlaceVector.begin(), myPlaceVector.end(), comparePlaces);
```

11.10

random_shuffle

- Randomly changes the elements in a container

```
// Scramble a vector's elements
random_shuffle(myVector.begin(), myVector.end());
```

  - requires random-access iterator

11.11

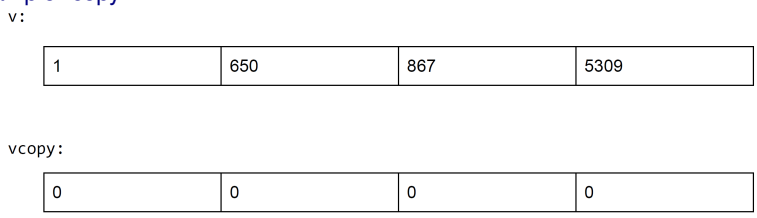
1.5 Iterator adaptor

Example: copy

```
vector<int> v;
v.push_back(1);
v.push_back(650);
v.push_back(867);
v.push_back(5309);

vector<int> vcopy(4);

copy(v.begin(), v.end(), vcopy.begin());
```

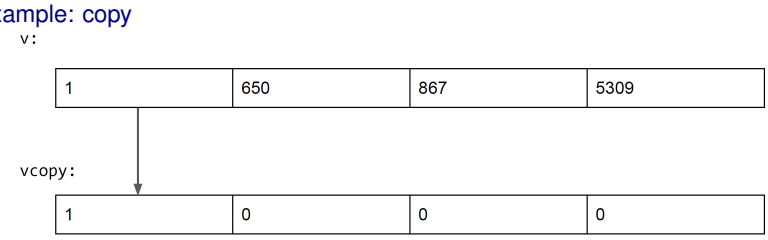
11.12

Example: copy



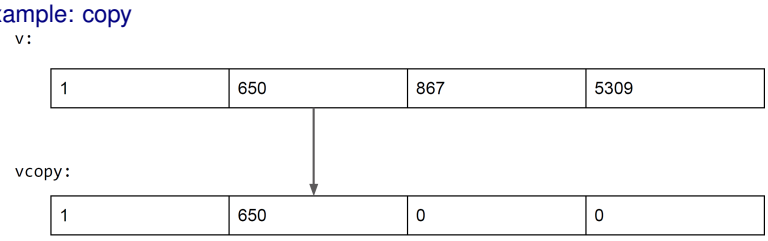
11.13

Example: copy



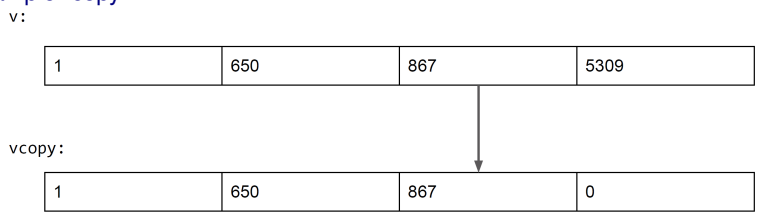
11.14

Example: copy



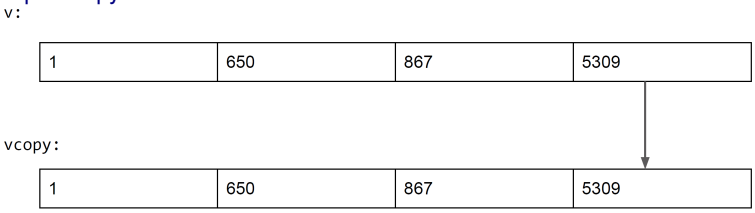
11.15

Example: copy



11.16

Example: copy



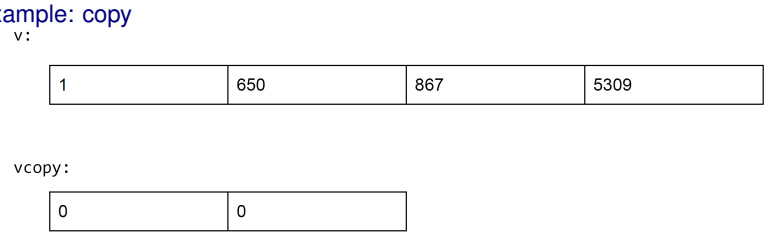
11.17

Example: copy

What happens if we have not allocated enough space?

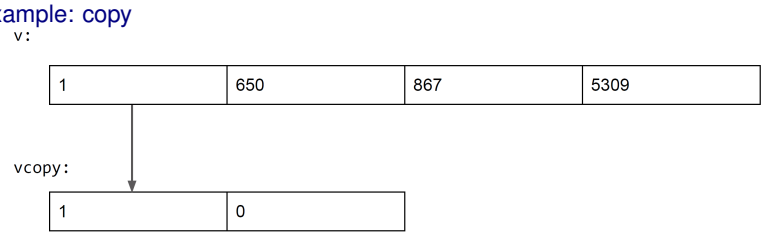
11.18

Example: copy



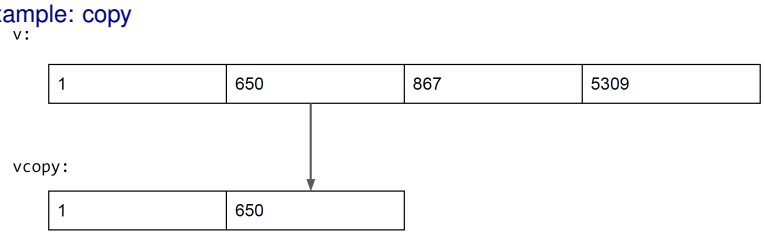
11.19

Example: copy



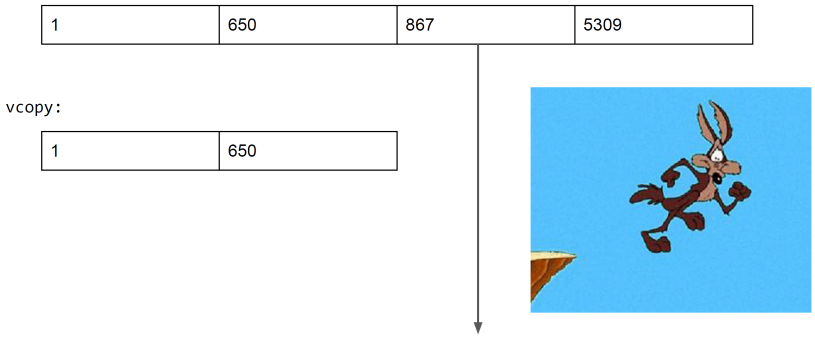
11.20

Example: copy



11.21

Example: copy



11.22

Example: copy

How can we avoid this problem?

11.23

Iterator adaptor

Iterator adaptor (defined in `<iterator>`) is an object that behaves like an iterator

- can be dereferenced with $\star$ and incremented with ++
- but they do not point to elements in a container

```
/* Declare an ostream_iterator that writes ints to cout. */
ostream_iterator<int> myItr(cout, "␣");

/* Write values to the iterator. These values will be printed to cout. */
*myItr = 137; // Prints 137 to cout
++myItr;

*myItr = 42; // Prints 42 to cout
++myItr;
```

11.24

copy (again)

What if we copy values from a container to an `ostream_iterator`?

```
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, "␣"));
```

The code above prints the elements of `myVector` separated with space! 11.25

Insert iterators

Iteratorer that when you write to them it will insert the value in the container using either `insert`, `push_back` or `push_front`.

```
vector<int> original = /* ... */
vector<int> destination;
reverse_copy(original.begin(), original.end(),
             inserter(destination,destination.begin()));
```

Syntax alternative:

```
vector<int> original = /* ... */
vector<int> destination;
copy(original.begin(), original.end(), back_inserter(destination));
copy(original.begin(), original.end(), front_inserter(destination));
```

11.26

1.6 Removal algorithm

remove

Despite the name those algorithm do not remove elements from containers

- algorithms operate on iterator and not container
- remove works by shifting elements
- to actually remove elements, you should use the `erase` function member:

```
// Remove all copies of the number 137 from a vector
myVector.erase(remove(myVector.begin(), myVector.end(), 137), myVector.end());
```

11.27

1.7 `transform`

`transform`

Apply a function to all the elements

```
int inc(int x){
    return ++x;
}

// increment each element in the container
transform(values.begin(), values.end(), values.begin(), inc);
```

- there is not requirement for `transform` other than the function returns a type compatible with the container

11.28

2 Function object

2.1 Introduction

Function

Function in C++ take a parameter and produce a return value

. . . but you already knew that. 11.29

Function object

- A function object is like a function but it can do a little more
- Function object are *classes*, not function
- Function objects define **`operator`**`()`

11.30

Function object — example

```
int ExampleFunction(string s) {
    cout << s << endl;
    return 42;
}

struct ExampleFunctor {
    int operator()(string s) {
        cout << s << endl;
        return 42;
    }
};

// Call a function
int a = ExampleFunction("Hello␣world!");

// Call a functor
ExampleFunctor f;
int b = f("Hello␣world!"); // or...
b = f.operator()("Hello␣world!");

// It's the same stuff!
assert(a == b);
```

11.31

2.2 Closures

Function object

- Function objects provide more than just complicated syntax to define functions!
- Function objects allow to emulate *closures*
- Lets look at an example of function object to count the number of words in a vector

11.32

Function — find short words

- STL gives the ability to count elements in containers with iterators and unary functions
- Suppose we want to find all words shorter than five letters in a vector

```
bool lengthIsLessThanFive(const string& str) {
    return str.length() < 5;
}

count_if(myVector.begin(), myVector.end(), lengthIsLessThanFive);
```

11.33

Function object — find short words

- Suppose we want to change the length (instead of 5, use 4?)
- Do we have to start from scratch?
- Can we still use `count_if`?

```
class ShorterThan {
public:
    /* Accept and store an int parameter */
    ShorterThan(size_t maxLength) : length(maxLength) {}

    /* Return whether the string length is less than the stored int. */
    bool operator() (const string& str) const {
        return str.length() < length;
    }

private:
    const size_t length;
};

count_if(myVector.begin(), myVector.end(), ShorterThan(length));
```

11.34

Function vs function object

Functions have:

- Local variable
- Arguments
- Global variable

Funktion objects have:

- Local variable
- Arguments
- Global variable
- *instance variables*

This tiny difference open up a lot of new opportunities.

- To define a function object that defines an infinite amount of possible functions
- Function Objects can remember information about the context in which they are called (emulates closures)
- Function objects can remember information between calls to the same function

11.35

3 Lambda

Simple lambda expression

A lambda expression is an expression that defines an anonymous function.

```
[](int x) -> int { return 2*x + 1; }
```

- begins with *lambda introduction*: `[]`
- then followed by a usual *list of arguments*
- *result type* can be specified *after* arguments list
  - if no result type is specified, it is guessed from the return type expression

    ```
    [](int x) { return 2*x + 1; }
    ```

- ends with a common *function body*
- an easy way to create simple functions

11.36

Simple lambda expression — example

```
int a[]{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

vector<int> v;

transform(begin(a), end(a), back_inserter(v), [](int x){ return 2*x + 1; });

copy(begin(v), end(v), ostream_iterator<int>{cout, "␣"});
// 1 3 5 7 9 11 13 15 17 19


int length = 5;
count_if(myVector.begin(), myVector.end(),
         [length](const string& x) { return x.length() < length}; );
```

- variable `length` is accessible from the function body

11.37

Challenge (continue)

Write a program that reads a list of integers from a file and write the average value of the numbers.

```
#include <iostream>
#include <fstream>
#include <set>
using namespace std;

int main() {
    ifstream input("data.txt");
    multiset<int> values;

    /* Read the data from the file. */
    int currValue;
    while (input >> currValue)
        values.insert(currValue);

    /* Compute the average. */
    cout << "Average␣is:␣" << accumulate(values.begin(), values.end(), 0.0)
          / values.size() << endl;
}
```

11.38

Challenge (continue)

```
#include <iostream>
#include <fstream>
#include <set>
using namespace std;

int main() {
    ifstream input("data.txt");
    int count = 0;
    /* Compute the average. */
    cout << "Average␣is:␣" << accumulate(istream_iterator<int>(input),
                    istream_iterator<int>(), 0.0,
                    [&count](int a, int b) { ++count; return a + b;}) / count
         << endl;
}
```

11.39

4 STL Algorithms: Application

Gauss-Jordan Elimination

$$\left(\begin{array}{cccc|c} 0 & 5 & 10 & -1 & -7 \\ 2 & 6 & 12 & 10 & 20 \\ 2 & 3 & 6 & 11 & 25 \end{array}\right) \text{ becomes } \left(\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 3 \\ 0 & 1 & 2 & 0 & -1 \\ 0 & 0 & 0 & 1 & 2 \end{array}\right)$$

Start in the first column and continue to the right:

- Look for a row that contains a pivot with a non-zero value in this column.
- Normalize the line so that the value in the current column is one.
- For every other row with a non-zero value in this column:
  - Normalize the line so that the value in the current column is one.
  - Subtract the rows containing the new pivot.
  - Normalize the line to restore any pivot it contains.
- Move current line above all other lines that do not already have one pivot.

11.40

Matrix inversion

$$\left(\begin{array}{ccc|ccc} 3 & 1 & 4 & 1 & 0 & 0 \\ 1 & 5 & 9 & 0 & 1 & 0 \\ 2 & 6 & 5 & 0 & 0 & 1 \end{array}\right) \text{ blir } \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 29/90 & -19/90 & 11/90 \\ 0 & 1 & 0 & -13/90 & -7/90 & 23/90 \\ 0 & 0 & 1 & 2/45 & 8/45 & -7/45 \end{array}\right)$$

11.41