32 8
33 1
35 8
36 ?
37 ?
38 0

C++ Exercises
Set 5

Author(s): David Vroom and Klaas Isaac Bijlsma

January 3, 2018

32

8

Purpose of this exercise: learn the implications of types of inheritance.

The given code has a design flaw in the class CSVLines. It inherits privately from std::vector<std::vector<std::string>> and it uses its begin and end functions. These however, return a (non const !) iterator. This is not what you want, because then the data can be modified. We solved it by making sure that const_iterators are returned.

Listing 1: main.cc

```
#include <vector>
#include <string>
#include <iostream>

class CSVLines: private std::vector<std::vector<std::string>>
{
    typedef std::vector<std::string> StrVector;

    public:
        CSVLines();

        void read() const;

        using std::vector<StrVector>::const_iterator;

        const_iterator begin() const;
        const_iterator end() const;
};

inline CSVLines::CSVLines()
{
    std::cout << "Constructor CSVLines called\n";
    assign(2, std::vector<std::string>{ "one", "two" });
}

inline void CSVLines::read() const
{
    std::cout << "read() called\n";
}

inline CSVLines::const_iterator CSVLines::begin() const
{
    return std::vector<StrVector>::cbegin();
}

inline CSVLines::const_iterator CSVLines::end() const
{
    return std::vector<StrVector>::cend();
}


void process(std::vector<std::string> const &strVect)
{
    std::cout << *strVect.begin()
```

```
                  << '\n'
                  << *(strVect.end()-1)
                  << '\n';
}

int main()
{
    CSVLines csvLines;
    csvLines.read();
    for (auto &next: csvLines)
        process(next);
}
```

33 8

Purpose of this exercise: learn to define an constructor accepting an initializer_list.

A constructor that accepts an initializer_list is added to the class Derived.

The following command is used to compile the code:
g++ -Wall -Werror -std=c++17 -o main main.cc
The compiler gave no output.

Listing 2: main.cc

```
#include <vector>
#include <string>

using namespace std;          !

class Derived: private vector<string>
{
    public:
        Derived() = default;
        Derived(initializer_list<string> iniList);
};

Derived::Derived(initializer_list<string> iniList)
:
    vector(iniList)
{}

int main()
{
    Derived derived{ "Klaas", "David", "Jurjen", "Frank" };
}
```

35 $\rightarrow$ 8

Purpose: learn to use containers to solve complex tasks.

A program is constructed that prints a sorted list of all different words appearing at its standard input.

Listing 3: main.cc

```
#include <iostream>
#include <set>
#include <string>

using namespace std;

int main()
{
    set<string> object;

    cout << "Enter a sequence of words (enter \"done\" when done):\n";
    string word;
    while (cin >> word)
    {
        if (word == "done")
            break;
        object.insert(word);
    }

    for (auto &next: object)
        cout << next << '\n';
}
```
[illegible]

36 ?

Purpose: learn to select the right container for the task at hand.

Now, a program is constructed that prints a sorted list of all different words appearing at its standard input, together with the number of times is was entered as input.

Listing 4: main.cc

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
    multiset<string> object;

    cout << "Enter a sequence of words (enter \"done\" when done):\n";
    string word;
    while (cin >> word)
    {
        if (word == "done")
            break;
        object.insert(word);
    }

    for (auto it = object.begin(), prev = object.end(); it != object.end(); prev
        = it, ++it)
        if (*it != *prev)
            cout << *it << ' ' << object.count(*it) << '\n';
}
```

Een multiset werkt, maar zorgt voor een nodeloos complexe afhandeling v.d. bepaling v. h. aantal

gebruik een container die direct kan doen wat je wilt

37 ?

Purpose: learn to shed excess capacity from a vector.

The output of the program of exercise 35 is now stored in a vector. Then one additional word is added to the vector. If needed, its excess capacity is shedded. Below is our code and the output it generates when initally two (different) input words are given. It can be seen that the capacity is nicely reduced from 4 to 3.

Listing 5: main1.cc

```
#include <iostream>
#include <set>
#include <vector>

using namespace std;

int main()
{
    set<string> object;

    cout << "Enter a sequence of words (enter \"done\" when done):\n";
    string word;
    while (cin >> word)
    {
        if (word == "done")
```

TC: initialize the vector with the set's elements

```
            break;
        object.insert(word);
    }

    vector<string> vecObj;
    for (auto &next: object)
        vecObj.push_back(next);

    cout << "1. Size: " << vecObj.size() << '\n'
         << "1. Capacity: " << vecObj.capacity() << '\n';

    vecObj.emplace_back("new");

    cout << "2. Size: " << vecObj.size() << '\n'
         << "2. Capacity: " << vecObj.capacity() << '\n';

    vector<string>(vecObj).swap(vecObj);

    cout << "3. Size: " << vecObj.size() << '\n'
         << "3. Capacity: " << vecObj.capacity() << '\n';
}
```

Where can see push-back?

```
Enter a sequence of words (enter "done" when done):
1. Size: 2
1. Capacity: 2
2. Size: 3
2. Capacity: 4
3. Size: 3
3. Capacity: 3
```

Now essentially the same is done, but by using a class VectorData that has a vector<string> data member. The output (from input "2 1 done") shows that again excess capacity is shedded.

Listing 6: vectordata/vectordata.h

```
#ifndef EX37B_VECTORDATA_H
#define EX37B_VECTORDATA_H

#include <vector>

class VectorData
{
    std::vector<std::string> d_data;

    public:
        void add(std::string const &word);
        void swap(VectorData &other);
        size_t size() const;
        size_t capacity() const;
};

inline void VectorData::add(std::string const &word)
{
    d_data.push_back(word);
}

inline void VectorData::swap(VectorData &other)
{
    d_data.swap(other.d_data);
}

inline size_t VectorData::size() const
{
    return d_data.size();
}

inline size_t VectorData::capacity() const
{
    return d_data.capacity();
}

#endif
```

Listing 7: main.cc

```
#include <iostream>
#include <set>
#include "vectordata/vectordata.h"

using namespace std;

int main()
{
    set<string> object;

    cout << "Enter a sequence of words (enter \"done\" when done):\n";
    string word;
    while (cin >> word)
    {
        if (word == "done")
            break;
        object.insert(word);
    }

    VectorData vecObj;                       // initialize
    for (auto &next: object)                 //   vecObj
        vecObj.add(next);

    cout << "1. Size: " << vecObj.size() << '\n'
         << "1. Capacity: " << vecObj.capacity() << '\n';

    vecObj.add("new");

    cout << "2. Size: " << vecObj.size() << '\n'
         << "2. Capacity: " << vecObj.capacity() << '\n';

    VectorData(vecObj).swap(vecObj);

    cout << "3. Size: " << vecObj.size() << '\n'
         << "3. Capacity: " << vecObj.capacity() << '\n';
}
```

```
Enter a sequence of words (enter "done" when done):
1. Size: 2
1. Capacity: 2
2. Size: 3
2. Capacity: 4
3. Size: 3
3. Capacity: 3
```

38 0

Purpose of this exercise: learn to fine-tune the unordered_multimap::count member.

We made a function that determines the number of unique keys in an unordered_multimap, by giving the keys a weight-factor inversely proportional to the number of times they occur.

Listing 8: main.cc

```
#include <iostream>
#include <string>
#include <unordered_map>

using namespace std;

size_t countKeys(unordered_multimap<string, string> &container)
{
```

```cpp
    double count = 0;

    for (auto& element: container)
        count += 1.0/container.count(element.first);

    return static_cast<size_t>(count);
}

int main(int argc, char **argv)
{
    typedef unordered_multimap<string, string> container;

    container object({{ "Netherlands", "Amsterdam" },
                      { "Belgium", "Brussels" },
                      { "Belgium", "Antwerp" },
                      { "France", "Nice" },
                      { "France", "Paris" }});

    size_t nUniqueKeys = countKeys(object);

    cout << "There are " << nUniqueKeys
         << " unique keys in the container\n";
}
```

VC.

Er is ee functie die je 't aantal gelijke keys oplevert (in feite ee range v. iterators) je hoeft 't dus niet op deze complexe wijze zelf te doen.