Exercises week 2: Class Templates - Revision

Klaas Isaac Bijlsma s2394480
David Vroom s2309939

February 28, 2018

Exercise 9

Learn to design a member template'

In the first attempt we forgot to decrement `d_nAvailable` when required. We used the following code,

semaphore.h

```
#ifndef INCLUDED_SEMAPHORE_H_
#define INCLUDED_SEMAPHORE_H_

#include <mutex>
#include <condition_variable>
#include <utility>

class Semaphore
{
    mutable std::mutex d_mutex;
    std::condition_variable d_condition;
    size_t d_nAvailable;
    // possibly other members

    public:
        template <typename Function, typename ...Params>
        bool wait(Function &&fun, Params &&...params);
        // other members
};
```

```
20
21 template <typename Function, typename ...Params>
22 bool Semaphore::wait(Function &&fun, Params &&...params)
23 {
24     std::unique_lock<std::mutex> lk(d_mutex);
25     while (d_nAvailable == 0)
26         d_condition.wait(lk);
27
28     bool ret = fun(std::forward<Params>(params)...);
29
30     if (!ret || d_nAvailable == 0)
31         return false;
32
33     --d_nAvailable;
34     return true;
35 }
36
37 #endif
```

Exercise 13

Learn to create a generic constructor for a virtual base class

In the first attempt our return statement of the `make` function was incorrect. We added `std::move`.

We used the following code,

abc/abc.h

```
1  #ifndef INCLUDED_ABC_H_
2  #define INCLUDED_ABC_H_
3
4  #include <utility>              // std::forward
5
6  class ABC
7  {
8      public:
9          virtual ~ABC();
10
11         void interface();       // calls run
12
13         template <typename Type, typename ...Params>
14         static ABC &&make(Params &&...params);
15
16     private:
17         virtual void run() = 0;
18 };
19
20 template <typename Type, typename ...Params>
21 ABC &&ABC::make(Params &&...params)
22 {
23     return std::move(Type{std::forward<Params>(params)...});
24 }
25
26 #endif
```

abc/destructor.cc

```
1  #include "abc.h"
```

```
2
3 ABC::~ABC()
4 {}
```

abc/interface.cc

```
1 #include "abc.h"
2
3 void ABC::interface()
4 {
5     run();
6 }
```

derived1/derived1.h

```
1  #ifndef INCLUDED_DERIVED1_H_
2  #define INCLUDED_DERIVED1_H_
3
4  #include "../abc/abc.h"
5  #include <ostream>
6
7  class Derived1: public ABC
8  {
9      public:
10         Derived1(std::ostream &out);
11     private:
12         void run() override;
13 };
14
15 #endif
```

derived1/derived1.ih

```
1 #include "derived1.h"
2 #include <iostream>
3
4 using namespace std;
```

derived1/derived1.cc

```
1  #include "derived1.ih"
2
3  Derived1::Derived1(std::ostream &out)
4  {
5      cout << "Derived1 constructed\n";
6  }
```

derived1/run.cc

```
1  #include "derived1.ih"
2
3  void Derived1::run()
4  {
5      cout << "run called from Derived1\n";
6  }
```

derived2/derived2.h

```
1  #ifndef INCLUDED_DERIVED2_H_
2  #define INCLUDED_DERIVED2_H_
3
4  #include "../abc/abc.h"
5  #include <istream>
6  #include <ostream>
7
8  class Derived2: public ABC
9  {
10     public:
11         Derived2(std::istream &in, std::ostream &out);
12     private:
13         void run() override;
14 };
15
16 #endif
```

derived2/derived2.ih

```
#include "derived2.h"
#include <iostream>

using namespace std;
```

derived2/derived2.cc

```
#include "derived2.ih"

Derived2::Derived2(std::istream &in, std::ostream &out)
{
    cout << "Derived2 constructed\n";
}
```

derived2/run.cc

```
#include "derived2.ih"

void Derived2::run()
{
    cout << "run called from Derived2\n";
}
```

process/process.h

```
#ifndef INCLUDED_PROCESS_H_
#define INCLUDED_PROCESS_H_

#include "../abc/abc.h"
#include <iomanip> // std::move()

class Process
{
    ABC &&d_abc;

    public:
        Process(ABC &&abc);
```

```
13          void execute();
14  };
15
16  Process::Process(ABC &&abc)
17  :
18      d_abc(std::move(abc))
19  {}
20
21  void Process::execute()
22  {
23      d_abc.interface();
24  }
25
26  #endif
```

main.cc

```
1   #include "process/process.h"
2   #include "derived1/derived1.h"
3   #include "derived2/derived2.h"
4   #include <iostream>
5
6   using namespace std;
7
8   int main(int argc, char **argv)
9   {
10      // part 1
11      Process process{
12                  argc == 1 ?
13                  static_cast<ABC &&>(Derived1{ cerr }) :
14                  static_cast<ABC &&>(Derived2{ cin, cout })
15              };
16      // etc.
17
18      // part 2
19      Process process2{
20                  argc == 1 ?
21                  ABC::make<Derived1>(cerr) :
22                  ABC::make<Derived2>(cin, cout)
23              };
```

```
24      // etc.
25  }
```

Exercise 14

Learn to add iterators to a class

This is the first attempt.

Since the iterators have to be passed to the `sort` generic algorithm, the iterators are of iterator type `random_access_iterator`.

We used the following code,

storage.h

```
1  #ifndef INCLUDED_STORAGE_H_
2  #define INCLUDED_STORAGE_H_
3
4  #include <vector>
5  #include <iterator>
6
7  template <typename Data>
8  class Storage
9  {
10     std::vector<Data *> d_storage;
11
12     public:
13         class iterator: public
14                 std::iterator<std::random_access_iterator_tag, Data>
15         {
16             friend class Storage;
17
18             typename std::vector<Data *>::iterator d_current;
19
20             iterator(typename std::vector<Data *>::iterator const &current);
21
22             public:
23                 iterator &operator++();
24                 iterator operator++(int);
25                 iterator &operator--();
26                 iterator operator--(int);
27
28                 iterator operator+(int step) const;
29                 iterator operator-(int step) const;
30                 int operator-(iterator const &rhs) const;
```

```
31
32                  bool operator==(iterator const &other) const;
33                  bool operator!=(iterator const &other) const;
34                  bool operator<(iterator const &other) const;
35
36                  Data &operator*() const;
37                  Data *operator->() const;
38          };
39
40          typedef std::reverse_iterator<iterator> reverse_iterator;
41
42          Storage<Data>::iterator begin();
43          Storage<Data>::iterator end();
44
45          Storage<Data>::reverse_iterator rbegin();
46          Storage<Data>::reverse_iterator rend();
47  };
48
49      // implementations of member functions of Storage<Data>::iterator
50  template <typename Data>
51  inline Storage<Data>::iterator::iterator(
52           typename std::vector<Data *>::iterator const &current)
53  :
54      d_current(current)
55  {}
56
57  template <typename Data>
58  inline typename Storage<Data>::iterator &
59                      Storage<Data>::iterator::operator++()
60  {
61      ++d_current;
62      return *this;
63  }
64
65  template <typename Data>
66  inline typename Storage<Data>::iterator
67                      Storage<Data>::iterator::operator++(int)
68  {
69      return iterator(d_current++);
70  }
71
```

```
72  template <typename Data>
73  inline typename Storage<Data>::iterator &
74                      Storage<Data>::iterator::operator--()
75  {
76      --d_current;
77      return *this;
78  }
79
80  template <typename Data>
81  inline typename Storage<Data>::iterator
82                      Storage<Data>::iterator::operator--(int)
83  {
84      return iterator(d_current--);
85  }
86
87  template <typename Data>
88  inline typename Storage<Data>::iterator
89                      Storage<Data>::iterator::operator+(int step) const
90  {
91      return iterator(d_current + step);
92  }
93
94  template <typename Data>
95  inline typename Storage<Data>::iterator
96                      Storage<Data>::iterator::operator-(int step) const
97  {
98      return iterator(d_current - step);
99  }
100
101 template <typename Data>
102 inline int Storage<Data>::iterator::operator-(iterator const &rhs) const
103 {
104     return d_current - rhs.d_current;
105 }
106
107 template <typename Data>
108 inline bool Storage<Data>::iterator::operator==(iterator const &other) const
109 {
110     return d_current == other.d_current;
111 }
112
```

```
113 template <typename Data>
114 inline bool Storage<Data>::iterator::operator!=(iterator const &other) const
115 {
116     return d_current != other.d_current;
117 }
118
119 template <typename Data>
120 inline bool Storage<Data>::iterator::operator<(iterator const &other) const
121 {
122     return d_current < other.d_current;
123 }
124
125 template <typename Data>
126 inline Data &Storage<Data>::iterator::operator*() const
127 {
128     return **d_current;
129 }
130
131 template <typename Data>
132 inline Data *Storage<Data>::iterator::operator->() const
133 {
134     return *d_current;
135 }
136
137     // implementations of member functions of Storage<Data>
138 template <typename Data>
139 inline typename Storage<Data>::iterator Storage<Data>::begin()
140 {
141     return iterator(d_storage.begin());
142 }
143
144 template <typename Data>
145 inline typename Storage<Data>::iterator Storage<Data>::end()
146 {
147     return iterator(d_storage.end());
148 }
149
150 template <typename Data>
151 inline typename Storage<Data>::reverse_iterator Storage<Data>::rbegin()
152 {
153     return reverse_iterator(d_storage.end());
```

```
154 }
155
156 template <typename Data>
157 inline typename Storage<Data>::reverse_iterator Storage<Data>::rend()
158 {
159     return reverse_iterator(d_storage.begin());
160 }
161
162 #endif
```