Exercises week 1: Function Templates - Revision

Klaas Isaac Bijlsma s2394480
David Vroom s2309939

February 21, 2018

Exercise 1

Show that templates don't result in 'code bloat'

In the first attempt we forgot to include the guards in the header files. This is now fixed.

A function template `add` and a union `PointerUnion` were defined in separate header files. We use this union to print the address of the function `add`. There are two source files, one for `fun` and one for `main`. The function `fun`, which includes `add.h`, instantiates `add` for `int`s and prints its address. Then, in `main` the same happens and `fun` is called. When the two source files of `fun` and `main` are compiled to object modules, they both contain an instantiation of `add`. Then they are linked to obtain an executable. The output of this executable gives two identical addresses, which means that only one instantiation of `add` is present. So it can be concluded that the linker prevents 'code bloat'.

add.h

```
1  #ifndef INCLUDED_ADD_
2  #define INCLUDED_ADD_
3
4  template <typename Type>
5  Type add(Type const &lhs, Type const &rhs)
6  {
7      return lhs + rhs;
8  }
9
10 #endif
```

pointerunion.h

```
#ifndef INCLUDED_POINTERUNION_
#define INCLUDED_POINTERUNION_

union PointerUnion
{
    int (*fp)(int const &, int const &);
    void *vp;
};

#endif
```

fun.cc

```
#include <iostream>
#include "add.h"
#include "pointerunion.h"

void fun()
{
    PointerUnion pu = { add };

    std::cout << pu.vp << '\n';
}
```

main.cc

```
#include <iostream>
#include "add.h"
#include "pointerunion.h"

void fun();

int main()
{
    PointerUnion pu = { add };
    std::cout << pu.vp << '\n';

    fun();
```

```
13 }
```

Exercise 2

Learn to embed a function template in a function template

In the first attempt we forgot to include the guards in the header files. This is now fixed. Also our formal typenames were not too informative, this is changed. Most importantly, we altered the parameter type of `value`.

We used the following code,

as.h

```
1  #ifndef INCLUDED_AS_
2  #define INCLUDED_AS_
3
4  template <typename newType, typename oldType>
5  newType as(oldType &&value)
6  {
7      return static_cast<newType>(value);
8  }
9
10 #endif
```

main.cc

```
1  #include <iostream>
2  #include "as.h"
3
4  using namespace std;
5
6  int main()
7  {
8      int chVal = 'X';
9
10     cout << as<char>(chVal) << '\n';
11 }
```

Exercise 3

Learn to construct a generic index operator

In the first attempt we forgot to include the guards in the header file. This is now fixed.

We used the following code,

storage.h

```
1  #ifndef INCLUDED_STORAGE_
2  #define INCLUDED_STORAGE_
3
4  #include <vector>
5  #include <initializer_list>
6
7  class Storage
8  {
9      std::vector<size_t> d_data;
10
11     public:
12         Storage() = default;
13         Storage(std::initializer_list<size_t> const &list);
14
15         template <typename Type>
16         size_t operator[](Type const &idx) const;
17
18         template <typename Type>
19         size_t &operator[](Type const &idx);
20 };
21
22 template <typename Type>
23 inline size_t Storage::operator[](Type const &idx) const
24 {
25     return d_data[static_cast<size_t>(idx)];
26 }
27
28 template <typename Type>
29 inline size_t &Storage::operator[](Type const &idx)
30 {
31     return d_data[static_cast<size_t>(idx)];
```

```
32 }
33
34 inline Storage::Storage(std::initializer_list<size_t> const &list)
35 :
36     d_data(list)
37 {}
38
39 #endif
```

Exercise 4

Learn to design and use a function template

In the first attempt we didn't document that an operator+= is needed for `std::string` and `Type`. This is now fixed. We also added a specialization for `Type == int`.

The code below is based on the solution of exercise 48 of part II of the C++ course.

exception/exception.h

```
1  // Type should have defined an operator+= for a std::string and Type
2  // A specialization is made for Type == int
3
4  #ifndef INCLUDED_EXCEPTION_
5  #define INCLUDED_EXCEPTION_
6
7  #include <string>
8  #include <exception>
9
10 class Exception: public std::exception
11 {
12     template <typename Type>
13     friend Exception &&operator<<(Exception &&in, Type const &txt);
14
15     std::string d_what;
16
17     public:
18         Exception() = default;
19
20         char const *what() const noexcept(true) override;
21 };
22
23 template <typename Type>
24 inline Exception &&operator<<(Exception &&in, Type const &txt)
25 {
26     in.d_what += txt;
27     return std::move(in);
28 }
29
30 template <>
```

```
31 inline Exception &&operator<<<int>(Exception &&in, int const &integer)
32 {
33     in.d_what += std::to_string(integer);
34     return std::move(in);
35 }
36
37 #endif
```

exception/what.cc

```
1 #include "exception.h"
2
3 char const *Exception::what() const noexcept(true)
4 {
5     return d_what.c_str();
6 }
```

Exercise 5

Learn to design a generic function template

In the first attempt we forgot to include the guards in the header files. This is now fixed.

We used the following code,

forwarder/forwarder.h

```
1  #ifndef INCLUDED_FORWARDER_
2  #define INCLUDED_FORWARDER_
3
4  template <typename Function, typename ...Params>
5  void forwarder(Function fun, Params &&...params)
6  {
7      fun(std::forward<Params>(params)...);
8  }
9
10 #endif
```

main.cc

```
1  #include "main.ih"
2
3  void fun(int first, int second)
4  {
5      cout << "fun(" << first << ", " << second << ")\n";
6  }
7
8  void fun(Demo &&dem1, Demo &&dem2)
9  {
10     cout << "fun(dem1, dem2)\n";
11 }
12
13 int main()
14 {
15                                  // inserts 'fun(dem1, dem2)' to cout
16     forwarder<void(Demo &&, Demo &&)>(fun, Demo{}, Demo{});
17
```

```
18                                      // inserts 'fun(1, 3)' to cout
19      forwarder<void(int, int)>(fun, 1, 3);
20  }
```