Exercises week 1: Function Templates

Klaas Isaac Bijlsma
s2394480

David Vroom
s2309939

February 14, 2018

Exercise 1

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

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 the linker prevents 'code bloat'.

add.h

```
template <typename Type>

Type add(Type const &lhs, Type const &rhs)
{
    return lhs + rhs;
}
```

pointerunion.h

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

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 };

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

Exercise 2

Learn to embed a function template in a function template

We used the following code,

as.h

```
template <typename Type1, typename Type2>

Type1 as(Type2 const &value)
{
    return static_cast<Type1>(value);
}
```

main.cc

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

using namespace std;

int main()
{
    int chVal = ’X’;

    cout << as<char>(chVal) << ’\n’;
}
```

Exercise 3

Exercise 4

Learn to design and use a function template

We used the following code,

exception/exception.h

```
#ifndef INCLUDED_EXCEPTION_
#define INCLUDED_EXCEPTION_

#include <string>
#include <exception>

class Exception: public std::exception
{
    template <typename Type>
    friend Exception &&operator<<(Exception &&in, Type const &txt);

    std::string d_what;

    public:
        Exception() = default;

        char const *what() const noexcept(true) override;
};

template <typename Type>
inline Exception &&operator<<(Exception &&in, Type const &txt)
{
    in.d_what += txt;
    return std::move(in);
}

#endif
```

exception/exception.ih

```
#include "exception.h"
```

exception/what.cc

```
#include "exception.ih"

char const *Exception::what() const noexcept(true)
{
    return d_what.c_str();
}
```

main.cc

```
#include <iostream>
#include "exception/exception.h"

using namespace std;

int main(int argc, char **argv)
try
{
    throw Exception{} << "insert anything that's ostream-insertable: "
                         "strings, values, argc, etc.";
}
catch (exception const &ex)
{
    cout << ex.what() << '\n';
}
```

Exercise 5

Learn to design a generic function template

We used the following code,

forwarder/forwarder.h

```
template <typename Function, typename ...Params>
void forwarder(Function fun, Params &&...params)
{
    fun(std::forward<Params>(params)...);
}
```

main.cc

```
#include "main.ih"

void fun(int first, int second)
{
    cout << "fun(" << first << ", " << second << ")\n";
}

void fun(Demo &&dem1, Demo &&dem2)
{
    cout << "fun(dem1, dem2)\n";
}

int main()
{
                                // inserts 'fun(dem1, dem2)' to cout
    forwarder<void(Demo &&, Demo &&)>(fun, Demo{}, Demo{});

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