T3

Exercises week 1: Function Templates

Klaas Isaac Bijlsma
s2394480

David Vroom
s2309939

February 15, 2018

Exercise 1

(0) Header files have no guards

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 ints 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

```
template <typename Type>

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

Personally, I like no vertical space between template header and function header.

Bot - 0

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

    fun();
}
```

?

Exercise 2

Learn to embed a function template in a function template

Not too informative.

We used the following code,

mind your naming.

as.h

```
template <typename Type1, typename Type2>

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

Fails if constructor of Type1 takes "value" by reference (non-const).

hint: Use better type deduction.

main.cc

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

using namespace std;

int main()
{
    int chVal = 'X';

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

Exercise 3

?

Learn to construct a generic index operator

We used the following code,

storage.h

?!? No include guards ?

```
#include <vector>
#include <initializer_list>


class Storage
{
    std::vector<size_t> d_data;

    public:
        Storage() = default;
        Storage(std::initializer_list<size_t> const &list);

        template <typename Type>
        size_t operator[](Type const &idx) const;

        template <typename Type>
        size_t &operator[](Type const &idx);
};

template <typename Type>
inline size_t Storage::operator[](Type const &idx) const
{
    return d_data[static_cast<size_t>(idx)];
}

template <typename Type>
inline size_t &Storage::operator[](Type const &idx)
{
    return d_data[static_cast<size_t>(idx)];
}

inline Storage::Storage(std::initializer_list<size_t> const &list)
:
    d_data(list)
{}
```

Exercise 4

?

Learn to design and use a function template

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

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
```

Only works if += is defined on string and Type.

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,

no guards :(

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);
}
```

Exercise 7

1

Gain some experience with the function selection mechanism

source

```
#include <iostream>

using namespace std;

template <typename Type>
inline Type const &max(Type const &left, Type const &right)
{
    return left > right ? left : right;
}


int main()
{
    cout << ::max(3.5, 4) << endl;
}
```

or a header included by it...

Why is the scope resolution operator required when calling max()? Appar~~a~~entnly, there is another function template for a function max in the header file iostream, that also expects two arguments that are a const & to the same formal type. The function selection mechanism will find a draw between this template function and ours on all criteria and therefore end the process with an ambiguity. To specify that we call the function max for which we defined a template above main, we need the scope resolution operator.

When compiling this function the compiler complains with a message like: `max.cc:13: error: no matching function for call to 'max(double, int)'` **Why doesn't the compiler generate a max(double, double) function in this case?** The standard conversion from int to double is only allowed for template non-type parameters. It is not part of the three allowed types of parameter type transformations. Since we deal with template type parameters, it is not possible.

Assume we add a function double max(double const &left, double const &right) to the source. Explain why this solves the problem. Now we have added a normal function (not a function template), for which the compiler is allowed to make the implicit conversion from int to double to fit the arguments to the parameters.

Assume we would then call `::max('a', 12)`**. Which** `max()` **function is then used and why?**

Now again the normal, non-template function is used. Both arguments are converted to a `double`.

Remove the additional `max` **function. Without using casts or otherwise changing the argument list of the function call** `max(3.5, 4)`**, how can we get the compiler to compile the source properly?**

By calling `::max<double>(3.5, 4)`.

Specify a general characteristic of the answer to the previous question (i.e., can the approach always be used or are there certain limitations?).

This only works if a standard conversion exists to convert the arguments to the type that is specified between pointy brackets.