C++ Exercises
Set 1

Author(s): Pau Lopez, Peter Versluis

–

22:50

February 13, 2025

1

Showing that multiple instantiations of a template with the same arguments don't result in 'code bloat' when linking.

Listing 1: `main.ih`

```
#include "sum.hh"
#include <iostream>
using namespace std;

void print1();
void print2();
```

Listing 2: `sum.hh`

```
#ifndef SUM_HH_
#define SUM_HH_

template <typename Type>
Type add(Type const &lhs, Type const &rhs)
{
        return lhs + rhs;
}

#endif
```

Listing 3: `print1.cc`

```
#include "main.ih"

string (*sum1)(string const &, string const &) = add;

void print1()
{
    cout << reinterpret_cast<void *>(sum1) << '\n';
}
```

Listing 4: `print2.cc`

```
#include "main.ih"

string (*sum2)(string const &, string const &) = add;

void print2()
{
    cout << reinterpret_cast<void *>(sum2) << '\n';
}
```

Listing 5: main.cc

```
#include "main.ih"

int main()
{
    print1();
    print2();
}
```

```
0x55620531f2ba
0x55620531f2ba
```

2

Embedding the static_cast template into a custom template.

Listing 6: as.hh

```
#ifndef AS_HH_
#define AS_HH_

template <typename RetType, typename ArgType>
RetType as(ArgType const &arg)
{
    return static_cast<RetType>(arg);
}

#endif
```

3

Simplifying the use of void * returning functions/operators using templates.

Listing 7: rawCapacity.hh

```
#ifndef RAW_CAPACITY_HH_
#define RAW_CAPACITY_HH_

#include <memory>

template <typename T>
T *rawCapacity(size_t cap)
{
    return static_cast<T *>(operator new(cap * sizeof(T)));
}

#endif
```

4

A class forwarder to call a function with perfectly forwarded arguments is designed.

Listing 8: forwarder.hh

```
#ifndef FORWARDER_HH_
#define FORWARDER_HH_

#include <utility>

template <typename Fun, typename ...Args>
void forwarder(Fun &&fun, Args &&...args)
{
    // Call the function with the perfectly forwarded arguments
    fun(std::forward<Args>(args)...);
}

#endif
```

Listing 9: main.cc

```
// No main.ih for simplicity
#include "forwarder.hh"
#include <iostream>
using namespace std;

class Demo {};

// Skipping 1F1F rule for demonstration purposes
void fun(int first, int second)
{
    cout << "funny int\n";
}

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

int main()
{
    // passing fun to forwarder recieving ints using pointers to functions
    void (*intFun)(int, int) = fun;
    forwarder(intFun, 1, 3);

    // passing fun to forwarder recieving Demos using pointers to functions
    void (*demoFun)(Demo &&, Demo &&) = fun;
    forwarder(demoFun, Demo{}, Demo{});

    // passing fun to forwarder recieving ints using static_cast
    forwarder(static_cast<void (*)(int, int)>(fun), 1, 3);

    // passing fun to forwarder recieving Demos using static_cast
    forwarder(static_cast<void (*)(Demo &&, Demo &&)>(fun), Demo{}, Demo{});
}
```

5

Exercise 5
Build a template to construct a generic index operator

Listing 10: storage/storage.hh

```
#ifndef STORAGE_HH_
#define STORAGE_HH_

#include <vector>

enum class TcpUdp
{
    SECONDS = 1,
    MU_SECONDS,
    PROTOCOL,
    SRC,
    DST,
    SPORT,
    DPORT,
    SENTPACKETS,
    SENTBYTES,
    RECVDPACKETS,
    RECVDBYTES,
    nFields,
};

enum class Icmp
{
    SECONDS = 1,
    MU_SECONDS,
    SRC,
    DST,
```

```
    ID,
    SENTPACKETS,
    SENTBYTES,
    RECVDPACKETS,
    RECVDBYTES,
    nFields,
};

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

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

    // Generic index operator
    template <typename TIndex>
    size_t operator[](TIndex idx) const;

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

#include "storage.i"

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

#endif
```

Listing 11: storage/storage.ih

```
#include "storage.hh"
#include <iostream>
#include <initializer_list>
```

Listing 12: storage/storage.i

```
template <typename TIndex>
inline size_t Storage::operator[](TIndex idx) const
{
    // Use at() for bounds checking
    return d_data.at(static_cast<size_t>(idx));
}

template <typename TIndex>
inline size_t &Storage::operator[](TIndex idx)
{
    return d_data.at(static_cast<size_t>(idx));
}
```

6

Designing a member function template for an overloaded operator.

Listing 13: exception.h

```
#ifndef INCLUDED_EXCEPTION_
#define INCLUDED_EXCEPTION_

#include <sstream>
#include <exception>

class Exception: public std::exception
{
    std::string d_what;
```

```
    template <typename Type>
    friend Exception &&operator<<(Exception &&in, Type arg);

    public:
        Exception() = default;

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

#include "exception.i"  // operator<< definition

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

#endif
```

Listing 14: exception.ih

```
#include "exception.h"
#include <sstream>

using namespace std;
```

Listing 15: exception.i

```
#include "exception.ih"

template <typename Type>
Exception &&operator<<(Exception &&in, Type const &arg)
{
    ostringstream oss;
    oss << arg;
    in.d_what += oss.str();
    return move(in);
}
```

7

Writing a function template that can be applied recursively on sets.

Listing 16: insertion.hh

```
#ifndef INSERTION_HH_
#define INSERTION_HH_

template<typename Type>
std::ostream &operator<<(std::ostream &out, std::set<Type> const &nSet)
{
    out << '{';
    bool first = true;
    for (auto const &item : nSet)
    {
        if (!first)
            out << ", ";
        first = false;
        out << item;
    }
    out << '}';
    return out;
}

#endif
```

The 2 reasons for not using a double ampersand (&&) are:

1. When using && instead of const &, the function can only take non-const arguments, thus not allowing to insert const sets into an ostream. Thus const & is the most adecuate type for passing both rvalues and lvalues independently from their constness.

2. Using && would make sense if it was necessary to use perfect forwarding in our implementation, which is not the case.