Programming in C/C++
Exercises set eight: function templates

Christiaan Steenkist
Jaime Betancor Valado
Remco Bos

January 25, 2017

Exercise 56, cast as

Let's be honest, static casts are a pain. We tried to alleviate this pain by shortening it to a simple as.

Code listings

Listing 1: main.ih

```
#include "main.h"

using namespace std;
```

Listing 2: main.h

```
#ifndef MAIN_H
#define MAIN_H

template <typename T>
T as(auto const castVar)
{
  return static_cast<T>(castVar);
}

template <typename T>
T as(auto const &&castVar)
{
```

```
  return static_cast<T>(castVar);
}

#endif
```

Listing 3: main.cc

```
#include "main.ih"
#include <iostream>

int main(int argc, char **argv)
{
  int chVal = 'X';

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

Exercise 57, code bloat away

Gee who would have thunked that the compiler is actually smart and combines two identical instances of a template function together into a single functino saving code space and preventing code bloat? I SURE would.

Output

```
0x804873e
0x804873e
```

Code listings

Listing 4: main.h

```
#include <iostream>

using namespace std;

void add1();
void add2();
```

Listing 5: pointerunion.h

```
#ifndef POINTERUNION_H
#define POITNERUNION_H
```

```

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

#endif
```

Listing 6: addtwo.h

```
#ifndef ADDTWO_H
#define ADDTWO_H

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

#endif
```

Listing 7: add1.cc

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

void add1()
{
    PointerUnion pu = { addTwo };
    std::cout << "Address addTwo1: "<< pu.vp << '\n';
}
```

Listing 8: add2.cc

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

void add2()
{
```

```
    PointerUnion pu = { addTwo };
    std::cout << "Address addTwo2: "<< pu.vp << '\n';
}
```

Listing 9: main.cc

```
#include "main.h"

int main()
{
    add1();
    add2();
}
```

Exercise 58, function templates

We were tasked to modify the definition of operator overloading in exercise 36 using function templates.

Code listings

Listing 10: exception.h

```
#ifndef INCLUDED_EXCEPTION_
#define INCLUDED_EXCEPTION_

#include <string>
#include <exception>

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

    public:
        Exception() = default;

        std::string &str();
        char const *what() const
            noexcept(true) override;
};

```

```
std::string &Exception::str()
{
    return d_what;
}

template <typename T>
inline Exception &&operator<<(Exception &&in, T rp)
{
    in.str() += rp;
    return std::move(in);
}
#endif
```

Exercuse 60, GUARDS!

Whether you are using a mutex or a smaphore, this guard's got your back.

Listing 11: main.ih

```
#include "main.h"

using namespace std;
```

Listing 12: main.h

```
#ifndef MAIN_H
#define MAIN_H

#include "guard.h"
#include <iostream>

void test()
{
  std::cout << "Hello!\n";
}

#endif
```

Listing 13: main.cc

```
#include "main.ih"

int main(int argc, char **argv)
```

```
{
  mutex someMutex;
  Semaphore someSemaphore;

  Guard guard1(someMutex);
  Guard guard2(someSemaphore);

  guard1(test);
  guard2(test);
}
```

Semaphore

Listing 14: main.ih

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

using namespace std;
```

Listing 15: main.h

```
#ifndef SEMAPHORE_H
#define SEMAPHORE_H

#include <condition_variable>
#include <mutex>
#include <cstddef>

class Semaphore
{
  protected:
    std::mutex mutable d_mutex;
    std::condition_variable d_condition;
    std::size_t d_nAvailable = 0;

    virtual bool done();

  public:
    Semaphore() = default;
    Semaphore(std::size_t nAvailable);
```

```

    std::size_t size() const;

    void notify();
    void notify_all();
    void wait();

    std::mutex &mutexRef();

    bool empty();
};

#endif
```

Listing 16: mutexref.cc

```
#include "semaphore.ih"

mutex &Semaphore::mutexRef()
{
  return d_mutex;
}
```

Guard

Listing 17: guard.ih

```
#include "guard.h"

using namespace std;
```

Listing 18: guard.h

```
#ifndef GUARD_H
#define GUARD_H

#include "semaphore.h"

class Guard
{
  std::mutex &d_mutex;

```

```
  public:
    Guard(std::mutex &someMutex);
    Guard(Semaphore &semaphore);

    template <typename T>
    void operator()(T func)
    {
      d_mutex.lock();
      func();
      d_mutex.unlock();
    }
};

#endif
```

Listing 19: constructor1.cc

```
#include "guard.ih"

Guard::Guard(mutex &someMutex)
:
  d_mutex(someMutex)
{
}
```

Listing 20: constructor2.cc

```
#include "guard.ih"

Guard::Guard(Semaphore &semaphore)
:
  d_mutex(semaphore.mutexRef())
{
}
```

Exercise 62, function selection

Here some questions and some answers, thats it.

Question 1

Q: Why is the scope resolution operator required when calling max()? A: Otherwise the compiler thinks that the default max function template must be used, instead of the self-defined max function template.

Question 2

Q: 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?

A: The compiler first sees a double and decides that the typename Type must be double. The compiler will not promote the int to a double by itself, because it checks if the int is a double, which it isn't.

Question 3

Q: Assume we add a function

```
double max(double const &left, double const &right)
```

to the source. Explain why this solves the problem.

A: To use the template version, the compiler needs to see that the arguments provided to the max function have the same type and then use that type. Providing a function where the compiler already knows which types to use, it only needs to know if types provided are convertible to a double, like all primitive types.

Question 4

Q: Assume we would then call `::max('a', 12)`. Which `max()` function is then used and why? A: It will use the second one, because a char and an int are not the same type. However a char and an int are primitive types, and can be converted to a double, so the second function can be used.

Question 5

Q: 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? A: Use this line: `cout << ::max<double>(3.5, 4) << endl;`

Listing 21: main.cc

```
#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<double>(3.5, 4) << endl;
}
```

Question 6

Q: Specify a general characteristic of the answer to the previous question (i.e., can the approach always be used or are there certain limitations?). A: The characteristic is that the template is used when any primitive types is used. The compiler already knows it should use a double, so it can convert primitive types. However the solution of the previous question is limited to primitive types.