

# Programming in C/C++

## Exercises set eight: function templates

Christiaan Steenkist  
Jaime Betancor Valado  
Remco Bos

January 20, 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

```
1 #include "main.h"
2
3 using namespace std;
```

Listing 2: main.h

```
1 #ifndef MAIN_H
2 #define MAIN_H
3
4 template <typename T>
5 T as(auto castVar)
6 {
7     return static_cast<T>(castVar);
8 }
9
10 #endif
```

Listing 3: main.cc

```
1 #include "main.ih"
2 #include <iostream>
3
4 int main(int argc, char **argv)
5 {
6     int chVal = 'X';
7
8     cout << as<char>(chVal) << '\n';
9 }
```

### 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 function saving code space and preventing code bloat? I SURE would.

### Output

```
1 0x804873e
2 0x804873e
```

### Code listings

Listing 4: main.h

```
1 #include <iostream>
2
3 using namespace std;
4
5 void add1();
6 void add2();
```

Listing 5: pointerunion.h

```
1 #ifndef POINTERUNION_H
2 #define POITNERUNION_H
3
4 union PointerUnion
5 {
6     int (*fp)(int const &, int const &);
7     void *vp;
```

```

8  };
9
10 #endif

```

Listing 6: addtwo.h

```

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

```

Listing 7: add1.cc

```

1  #include <iostream>
2  #include "addtwo.h"
3  #include "pointerunion.h"
4
5  void add1()
6  {
7      PointerUnion pu = { addTwo };
8      std::cout << "Address addTwo1: " << pu.vp << '\n';
9  }

```

Listing 8: add2.cc

```

1  #include <iostream>
2  #include "addtwo.h"
3  #include "pointerunion.h"
4
5  void add2()
6  {
7      PointerUnion pu = { addTwo };
8      std::cout << "Address addTwo2: " << pu.vp << '\n';
9  }

```

Listing 9: main.cc

```
1 #include "main.h"
2
3 int main()
4 {
5     add1();
6     add2();
7 }
```

## 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

```
1 #ifndef INCLUDED_EXCEPTION_
2 #define INCLUDED_EXCEPTION_
3
4 #include <string>
5 #include <exception>
6
7 class Exception: public std::exception
8 {
9     std::string d_what;
10
11     public:
12         Exception() = default;
13
14         std::string &str();
15         char const *what() const
16             noexcept(true) override;
17 };
18
19 std::string &Exception::str()
20 {
21     return d_what;
22 }
```

```

23
24 template <typename T>
25 inline Exception &&operator<<(Exception &&in, T rp)
26 {
27     in.str() += rp;
28     return std::move(in);
29 }
30 #endif

```

### Exercise 60, GUARDS!

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

Listing 11: main.ih

```

1 #include "main.h"
2
3 using namespace std;

```

Listing 12: main.h

```

1 #ifndef MAIN_H
2 #define MAIN_H
3
4 #include "guard.h"
5 #include <iostream>
6
7 void test()
8 {
9     std::cout << "Hello!\n";
10 }
11
12 #endif

```

Listing 13: main.cc

```

1 #include "main.ih"
2
3 int main(int argc, char **argv)
4 {
5     mutex someMutex;
6     Semaphore someSemaphore;
7

```

```

8   Guard guard1(someMutex);
9   Guard guard2(someSemaphore);
10
11   guard1(test);
12   guard2(test);
13 }

```

## Semaphore

Listing 14: main.ih

```

1  #include "semaphore.h"
2  #include <iostream>
3
4  using namespace std;

```

Listing 15: main.h

```

1  #ifndef SEMAPHORE_H
2  #define SEMAPHORE_H
3
4  #include <condition_variable>
5  #include <mutex>
6  #include <cstdint>
7
8  class Semaphore
9  {
10     protected:
11         std::mutex mutable d_mutex;
12         std::condition_variable d_condition;
13         std::size_t d_nAvailable = 0;
14
15         virtual bool done();
16
17     public:
18         Semaphore() = default;
19         Semaphore(std::size_t nAvailable);
20
21         std::size_t size() const;
22
23         void notify();

```

```

24     void notify_all();
25     void wait();
26
27     std::mutex &mutexRef();
28
29     bool empty();
30 };
31
32 #endif

```

Listing 16: mutexref.cc

```

1  #include "semaphore.ih"
2
3  mutex &Semaphore::mutexRef()
4  {
5      return d_mutex;
6  }

```

### Guard

Listing 17: guard.ih

```

1  #include "guard.h"
2
3  using namespace std;

```

Listing 18: guard.h

```

1  #ifndef GUARD_H
2  #define GUARD_H
3
4  #include "semaphore.h"
5
6  class Guard
7  {
8      std::mutex &d_mutex;
9
10     public:
11         Guard(std::mutex &someMutex);
12         Guard(Semaphore &semaphore);
13

```

```

14     void operator() (void (*func) ());
15 };
16
17 #endif

```

Listing 19: constructor1.cc

```

1 #include "guard.ih"
2
3 Guard::Guard(mutex &someMutex)
4 :
5     d_mutex(someMutex)
6 {
7 }

```

Listing 20: constructor2.cc

```

1 #include "guard.ih"
2
3 Guard::Guard(Semaphore &semaphore)
4 :
5     d_mutex(semaphore.mutexRef())
6 {
7 }

```

Listing 21: paroperator.cc

```

1 #include "guard.ih"
2
3 void Guard::operator() (void (*func) ())
4 {
5     d_mutex.lock();
6     func();
7     d_mutex.unlock();
8 }

```

## 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: Because max is defined in the global namespace.



## Question 2

Q: When compiling this function the compiler complains with a message like:

```
1 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

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

to the source. Explain why this solves the problem.

A: The compiler does not have to check if both values are the same type (i.e. `double` in the previous question), so it can convert the `int` into a `double`.

## Question 4

Q: Assume we would then call `::max('a', 12)`. Which `max()` function is then used and why? A: It will use the first one, the template. Because `'a'` is a `char` and not a `double`.

## 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 22: main.cc

```
1 #include <iostream>
2
3 using namespace std;
4
5 template <typename Type>
6 inline Type const &max(Type const &left,
7     Type const &right)
```

```
8 {  
9     return left > right ? left : right;  
10 }  
11  
12 int main()  
13 {  
14     cout << ::max<double>(3.5, 4) << endl;  
15 }
```

### 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 only used with the specified type (in this case double). If some variable can be converted to the specified type, then it will always work. A limitation is when the compiler cannot convert a type to another type (e.g. a `str::string` to `int`).