

## Week 4: II

### Exercise 25

../25/insertable/insertable.h

```
1  #ifndef INCLUDED_INSERTABLE_
2  #define INCLUDED_INSERTABLE_
3
4  #define HDR_   template <typename Data, \
5                template <typename, typename> class Container, \
6                template <typename> class AllocationPolicy>
7
8  template <
9      typename Data,
10     template <typename, typename> class Container = std::vector,
11     template <typename> class AllocationPolicy = std::allocator
12 >
13 class Insertable: public Container<Data, AllocationPolicy<Data>>
14 {
15     using Cont = Container<Data, AllocationPolicy<Data>>;
16
17     public:
18         Insertable();
19         Insertable(const Cont &rhs);
20         Insertable(Cont &&rhs);
21         Insertable(const Insertable &rhs);
22         Insertable(Insertable &&rhs);
23         Insertable(Data &&rhs);
24 };
25
26 #include "insertion.h" // Free function, but connected to this class
27
28 #undef HDR_
29 #endif
```

../25/insertable/insertion.h

```
1  #ifndef INCLUDED_INSERTIONT_
2  #define INCLUDED_INSERTIONT_
3
4  template <typename Out, class Insertable>
5  std::ostream &operator<<(Out &out, const Insertable &ins)
6  // Just the declaration here too? Wasn't sure; see ex. 26 for the code.
7
8  #endif
```

## Exercise 26

../26/insertable/insertable.h

```
1  #ifndef INCLUDED_INSERTABLE_
2  #define INCLUDED_INSERTABLE_
3
4  #define HDR_   template <typename Data, \
5                template <typename, typename> class Container, \
6                template <typename> class AllocationPolicy>
7
8  template <
9      typename Data,
10     template <typename, typename> class Container = std::vector,
11     template <typename> class AllocationPolicy = std::allocator
12 >
13 class Insertable: public Container<Data, AllocationPolicy<Data>>
14 {
15     using Cont = Container<Data, AllocationPolicy<Data>>;
16
17     public:
18         Insertable();
19         Insertable(const Cont &rhs);
20         Insertable(Cont &&rhs);
21         Insertable(const Insertable &rhs);
22         Insertable(Insertable &&rhs);
23         Insertable(Data &&rhs);
24 };
25
26 // Constructors just call constructor of underlying type
27 HDR_
28 Insertable<Data, Container, AllocationPolicy>::Insertable()
29     : Cont()
30 {};
31 HDR_
32 Insertable<Data, Container, AllocationPolicy>::Insertable(const Cont &rhs)
33     : Cont(rhs)
34 {};
35 HDR_
36 Insertable<Data, Container, AllocationPolicy>::Insertable(Cont &&rhs)
37     : Cont(rhs)
38 {};
39 HDR_
40 Insertable<Data, Container, AllocationPolicy>::Insertable(const Insertable &rhs)
41     : Cont(rhs)
42 {};
43 HDR_
44 Insertable<Data, Container, AllocationPolicy>::Insertable(Insertable &&rhs)
45     : Cont(rhs)
46 {};
47 HDR_
48 Insertable<Data, Container, AllocationPolicy>::Insertable(Data &&rhs)
49     : Cont(rhs)
50 {};
51
52 #include "insertion.h" // Free function, but connected to this class
53
54 #undef HDR_
55 #endif
```

../26/insertable/insertion.h

```
1  #ifndef INCLUDED_INSERTIONT_
2  #define INCLUDED_INSERTIONT_
3
```

```
4  template <typename Out, class Insertable>
5  Out &operator<<(Out &out, const Insertable &ins)
6  {
7      for (auto el: ins)
8          out << el << '\n';
9      return out;
10 };
```

```
11
12 #endif
13 // Note: ostream as template parameter instead; otherwise it interferes with
14 // the one in the std. library that already has a insertion operator with
15 // one template parameter (ambiguous)
```

../26/main.ih

```
1  #define ERR(msg) printf("%s : %d", (msg), __LINE__)
2
3  using namespace std;
4
5  #include <iostream>
6  #include <vector>
7
8  #include "insertable/insertable.h"
```

../26/main.cc

```
1  #include "main.ih"
2
3  int main(int argc, char const **argv)
4  {
5      typedef Insertable<int, std::vector> InsertableVector;
6      std::vector<int> vi {1, 2, 3, 4, 5};
7
8      InsertableVector iv;
9      InsertableVector iv2(vi);
10     InsertableVector iv3(4);
11     InsertableVector iv4(iv2);
12
13     cout << iv2 << '\n' <<
14           iv3 << '\n' <<
15           iv4 << '\n';
16
17     iv3.push_back(123);
18     cout << iv3 << '\n';
19 }
```

## Exercise 28

../28/basictraits/basictraits.h

```
1  #ifndef INCLUDED_BASIC_
2  #define INCLUDED_BASIC_
3
4  template<typename T>
5  class BasicTraits
6  {
7      template<typename T2>
8      struct Basic
9      {
10         typedef T2 Type;
11         enum
12         {
13             isPlain = 1,
14             isPointer = 0,
15             isRef = 0,
16             isRRef = 0
17         };
18     };
19
20     template<typename T2>
21     struct Basic<T2 *> //pointer to a type
22     {
23         typedef T2 Type;
24         enum
25         {
26             isPlain = 0,
27             isPointer = 2,
28             isRef = 0,
29             isRRef = 0
30         };
31     };
32
33     template<typename T2>
34     struct Basic<T2 &> //reference to a type
35     {
36         typedef T2 Type;
37         enum
38         {
39             isPlain = 0,
40             isPointer = 0,
41             isRef = 3,
42             isRRef = 0
43         };
44     };
45
46     template<typename T2>
47     struct Basic<T2 &&> //rvalue ref to a type
48     {
49         typedef T2 Type;
50         enum
51         {
52             isPlain = 0,
53             isPointer = 0,
54             isRef = 0,
55             isRRef = 4
56         };
57     };
58
59 public:
60
61     BasicTraits(BasicTraits const &other) = delete;
62
```

```
63     typedef typename Basic<T>::Type ValueType;
64     typedef ValueType *PtrType;
65     typedef ValueType &RefType;
66     typedef ValueType &&RvalueRefType;
67
68     //template<typename T2>
69     BasicTraits(ValueType input);
70     BasicTraits(ValueType *input);
71
72
73
74     ValueType value()
75     {
76         return d_value;
77     }
78
79     enum
80     {
81         isPlainType = Basic<T>::isPlain,
82         isPointerType = Basic<T>::isPointer,
83         isRefType = Basic<T>::isRef,
84         isRRefType = Basic<T>::isRRef
85     };
86
87     private:
88
89         ValueType d_value;
90 };
91
92 template<typename T>
93 BasicTraits<T>::BasicTraits(ValueType input)
94 {
95     d_value = input;
96 }
97 template<typename T>
98 BasicTraits<T>::BasicTraits(ValueType *input)
99 {
100     d_value = *input;
101 }
102
103
104
105
106
107 #endif
```

../28/main.ih

```
1 #define ERR(msg) printf("%s : %d", (msg), __LINE__)
2
3 #include "basictraits/basictraits.h"
4
5 #include <iostream>
6
7 using namespace std;
```

../28/main.cc

```
1 #include "main.ih"
2
3 int main(int argc, char const **argv)
4 {
5     int plain = 3;
6     auto ptr = &plain;
7 }
```

```
8  cout << "plain :" << BasicTraits<const int>::isPlainType << '\n'
9      << "pointer :" << BasicTraits<int*>::isPointerType << '\n'
10     << "reference :" << BasicTraits<int&>::isRefType << '\n'
11     << "rvalue reference :" << BasicTraits<int&&>::isRRefType << '\n';
12
13     BasicTraits<int>    BObject1(ptr);
14     BasicTraits<int*>   BObject2(ptr);
15     BasicTraits<int&>   BObject3(ptr);
16     BasicTraits<int&&>  BObject4(ptr);
17
18     cout << BObject1.value() << '\n'
19          << BObject2.value() << '\n'
20          << BObject3.value() << '\n'
21          << BObject4.value() << '\n'; //prints 3 regardless of template arg
22     //pain is niet const.....
23 }
```

### Exercise 31

In line 98, the `+` operator is defined to return a `BinExpr` of which the third template argument is `ADD`. In the struct which is then returned the `operator[]` (line 45) is defined to return a function `cp` which is a member of a struct with a template parameter `operation`, which in this case is `ADD`, as this was the third template argument provided to the struct `BinExpr` in which the struct `Operation` is nested.

Changing the `ADD` in line 77 to `MUL` generates errors since it will now attempt to call a function from a struct `Operation` with a third template argument `SUM` in a struct encompassing it with third template argument `SUM`. Which has not been defined. Since the only options for the third template argument are now `MUL` and `SUB`.

So the association is made in line 45.

## Exercise 32

With an expression template to add VI objects the resulting expression is still a temp object whose elements are the sums of the elements of the four VI objects.

Let  $N$  be the number of elements in each VI object, of which we have 4.

Without using an expression template the result would be obtained as follows: First  $v1 + v2$  would be computed and stored as a tmp (lets call it  $v12$ ) Then  $v12 + v3$  would be computed as stored as a tmp ( $v123$ ) Then finally  $v123 + v4$  would be computed and stored as the resulting tmp. In this case we would have  $(4-1) * 2 * N = 6N$  index operations.

However, by using an expression template, the result is obtained, without creating tmps between each step, instead it simply passes on the addresses of the indexes, nothing is actually computed until the expression  $v1 + v2 + v3 + v4$  is converted to a new vector: result.  $result = (((v1 + v2) + v3) + v4)$ , where the sum of the of each index of the separate vectors is assigned to the corresponding index of result. So here we need to look up the values in the VI objects and assign the results in the result vector, which gives us a total number of  $5N$  index computations.

Clearly  $5N < 6N$ , so by using an expression template we can reduce the number of index computations.