Week 4

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  #define CONT_ Container<Data, AllocationPolicy<Data>>
8  #define INS_  Insertable<Data, Container, AllocationPolicy>
9
10 #include <vector>
11 #include <memory>
12 #include <iterator>
13
14 template <typename Data,
15 template <typename, typename> class Container = std::vector,
16 template <typename> class AllocationPolicy = std::allocator>
17 class Insertable: public Container<Data, AllocationPolicy<Data>>
18 {
19   public:
20     Insertable();
21     Insertable(const CONT_ &RHS);
22     Insertable(const Insertable &RHS);
23     Insertable(Data RHS);
24 };
25
26 #undef HDR_
27 #undef CONT_
28 #undef INS_
29 #endif
```

Exercise 26

../26/insertable/insertable.h

```
#ifndef INCLUDED_INSERTABLE_
#define INCLUDED_INSERTABLE_

#define HDR_  template <typename Data, \
              template <typename, typename> class Container, \
              template <typename> class AllocationPolicy>
#define CONT_ Container<Data, AllocationPolicy<Data>>
#define INS_  Insertable<Data, Container, AllocationPolicy>

#include <vector>
#include <memory>
#include <iterator>

template <typename Data,
template <typename, typename> class Container = std::vector,
template <typename> class AllocationPolicy = std::allocator>
class Insertable: public Container<Data, AllocationPolicy<Data>>
{
  public:
    Insertable();
    Insertable(const CONT_ &RHS);
    Insertable(const Insertable &RHS);
    Insertable(Data RHS);
};

// Constructors just call constructor of underlying type
HDR_
INS_::Insertable()
: CONT_()
{};
HDR_
INS_::Insertable(const CONT_ &RHS)
  : CONT_(RHS)
{};
HDR_
INS_::Insertable(const Insertable &RHS)
  : CONT_(RHS)
{};
HDR_
INS_::Insertable(Data RHS)
  : CONT_(RHS)
{};

#undef HDR_
#undef CONT_
#undef INS_
#endif
```

../26/insertion.h

```
#ifndef INCLUDED_INSERTIONT_
#define INCLUDED_INSERTIONT_

template <class Insertable>
std::ostream &operator<<(std::ostream &out, Insertable &ins)
{
  for (auto el: ins)
    out << el << '\n';
  return out;
};

#endif
```

../26/main.ih

```
#define ERR(msg) printf("%s : %d", (msg), __LINE__)

using namespace std;

#include <iostream>

#include "insertable/insertable.h"
#include "insertion.h"
```

../26/main.cc

```
#include "main.ih"

int main(int argc, char const **argv)
{
  typedef Insertable<int, std::vector> InsertableVector;
  std::vector<int> vi {1, 2, 3, 4, 5};

  InsertableVector iv;
  InsertableVector iv2(vi);
  InsertableVector iv3(4);
  InsertableVector iv4(iv2);

  cout << iv2 << '\n' <<
          iv3 << '\n' <<
          iv4 << '\n';

  iv3.push_back(123);
  cout << iv3 << '\n';
}
```

Exercise 29

Note: since exercise 30 was completed first, I used it as a base for exercise 29, just with its own plus struct. An easier example could be constructed for this particular exercise.

../29/expr.h

```
1  #ifndef INCLUDED_EXPRT_
2  #define INCLUDED_EXPRT_
3
4  #define EXPR_ template<typename LHS, \
5                         typename RHS, \
6                         template<typename> class Operation>
7
8  #include <cstddef>
9  #include <functional>
10
11 EXPR_
12 struct Expr;
13
14 // Trait class
15 template<typename RHS>
16 struct BasicType
17 {
18   typedef RHS ObjType;
19 };
20
21 EXPR_
22 struct BasicType<Expr<LHS, RHS, Operation>>
23 {
24   typedef typename Expr<LHS, RHS, Operation>::ObjType ObjType;
25 };
26
27 EXPR_
28 struct Expr
29 {
30   typedef typename BasicType<RHS>::ObjType ObjType;
31   typedef typename ObjType::value_type value_type;
32
33   LHS const &d_lhs;
34   RHS const &d_rhs;
35
36   Expr(LHS const &lhs, RHS const &rhs);
37
38   value_type operator[](size_t ix) const;
39   operator ObjType() const;
40 };
41
42 EXPR_
43 Expr<LHS, RHS, Operation>::Expr(LHS const &lhs, RHS const &rhs)
44 :
45   d_lhs(lhs),
46   d_rhs(rhs)
47 {};
48
49 EXPR_
50 typename Expr<LHS, RHS, Operation>::value_type Expr<LHS, RHS, Operation>::operator[](
       size_t ix) const
51 {
52   static Operation<value_type> operation;
53   return operation(d_lhs[ix], d_rhs[ix]);
54 }
55
56 EXPR_
57 Expr<LHS, RHS, Operation>::operator Expr<LHS, RHS, Operation>::ObjType() const
58 {
59   ObjType retVal;
```

```
60    for (size_t ix = 0; ix != d_lhs.size(); ++ix)
61      retVal.push_back((*this)[ix]);
62    return retVal;
63  }
64
65  #include "plusdeluxe.h"
66  template<typename LHS, typename RHS>
67  Expr<LHS, RHS, plusdeluxe> operator+(LHS const &lhs, RHS const &rhs)
68  {
69    return Expr<LHS, RHS, plusdeluxe>(lhs, rhs);
70  }
71  // Works in this case, but depends on the continued existence of
72  // its constituents. In other cases, RBV may be better.
73
74  #undef EXPR_
75  #endif
```

../29/plusdeluxe.h

```
1   #ifndef INCLUDED_PLUSDELUXET_
2   #define INCLUDED_PLUSDELUXET_
3
4   template<typename RetType>
5   struct plusdeluxe
6   {
7     RetType operator()(const RetType &lhs, const RetType &rhs) const
8     {
9       return lhs + rhs;
10    }
11  };
12
13  #endif
```

Exercise 30

../30/expr.h

```
1  #ifndef INCLUDED_EXPRT_
2  #define INCLUDED_EXPRT_
3
4  #define EXPR_ template<typename LHS, \
5                         typename RHS, \
6                         template<typename> class Operation>
7
8  #include <cstddef>
9  #include <functional>
10
11 EXPR_
12 struct Expr;
13
14 // Trait class
15 template<typename RHS>
16 struct BasicType
17 {
18   typedef RHS ObjType;
19 };
20
21 EXPR_
22 struct BasicType<Expr<LHS, RHS, Operation>>
23 {
24   typedef typename Expr<LHS, RHS, Operation>::ObjType ObjType;
25 };
26
27 EXPR_
28 struct Expr
29 {
30   typedef typename BasicType<RHS>::ObjType ObjType;
31   typedef typename ObjType::value_type value_type;
32
33   LHS const &d_lhs;
34   RHS const &d_rhs;
35
36   Expr(LHS const &lhs, RHS const &rhs);
37   size_t size() const;
38
39   value_type operator[](size_t ix) const;
40   operator ObjType() const;
41 };
42
43 EXPR_
44 Expr<LHS, RHS, Operation>::Expr(LHS const &lhs, RHS const &rhs)
45 :
46   d_lhs(lhs),
47   d_rhs(rhs)
48 {};
49
50 EXPR_
51 size_t Expr<LHS, RHS, Operation>::size() const
52 {
53   return d_lhs.size();
54 };
55
56 EXPR_
57 typename Expr<LHS, RHS, Operation>::value_type Expr<LHS, RHS, Operation>::operator[](
       size_t ix) const
58 {
59   static Operation<value_type> operation;
60   return operation(d_lhs[ix], d_rhs[ix]);
61 }
```

```
62
63  EXPR_
64  Expr<LHS, RHS, Operation>::operator Expr<LHS, RHS, Operation>::ObjType() const
65  {
66    ObjType retVal;
67    for (size_t ix = 0; ix != d_lhs.size(); ++ix)
68      retVal.push_back((*this)[ix]);
69    return retVal;
70  }
71
72  template<typename LHS, typename RHS>
73  Expr<LHS, RHS, std::multiplies> operator*(LHS const &lhs, RHS const &rhs)
74  {
75    return Expr<LHS, RHS, std::multiplies>(lhs, rhs);
76  }
77
78  template<typename LHS, typename RHS>
79  Expr<LHS, RHS, std::plus> operator+(LHS const &lhs, RHS const &rhs)
80  {
81    return Expr<LHS, RHS, std::plus>(lhs, rhs);
82  }
83
84  template<typename LHS, typename RHS>
85  Expr<LHS, RHS, std::divides> operator/(LHS const &lhs, RHS const &rhs)
86  {
87    return Expr<LHS, RHS, std::divides>(lhs, rhs);
88  }
89  // As in 29, these work in this situation, but must be wary of scope issues
90  // as these depend on references
91
92  #undef EXPR_
93  #endif
```

../30/main.ih

```
1  #define ERR(msg) printf("%s : %d", (msg), __LINE__)
2
3  #include "expr.h"
4  #include "printvector.h"
5
6  #include <vector>
7
8  template <typename T>
9  void print(T inputVector);
10 // This one is just for testing
11
12 using namespace std;
```

../30/main.cc

```
1  #include "main.ih"
2
3  #include <vector>
4
5  int main()
6  {
7    using IVect = vector<int>;
8    IVect iv1(10, 4);         // IVect: vector<int>
9    IVect iv2(10, 3);
10   IVect iv3(10, 2);
11   IVect iv4(10, 1);
12
13   IVect iResult { iv1 * (iv2 + iv3) / iv4 };
14
15   using DVect = vector<double>;
```

```
16    DVect dv1(10, 4.1);      // DVect: vector<double
17    DVect dv2(10, 3.1);
18    DVect dv3(10, 2.1);
19    DVect dv4(10, 1.1);
20
21    DVect dResult { dv1 * (dv2 + dv3) / dv4 };
22
23    print(dv1);        // Just for testing
24    print(dResult);
25  }
```

This one is just for testing the outcomes:

../30/printvector.h

```
1  #ifndef INCLUDED_PRINTVECTORT_
2  #define INCLUDED_PRINTVECTORT_
3
4  #include <iostream>
5
6  template <typename T>
7  void print(T inputVector)
8  {
9    for (auto &el: inputVector)
10     std::cout << el << ' ';
11   std::cout << '\n';
12 }
13
14 #endif
```