Week 4: II

Exercise 25

../25/insertable/insertable.h

```cpp
#ifndef INCLUDED_INSERTABLE_
#define INCLUDED_INSERTABLE_

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

template <
          typename Data,
          template <typename, typename> class Container = std::vector,
          template <typename> class AllocationPolicy = std::allocator
         >
class Insertable: public Container<Data, AllocationPolicy<Data>>
{
  using Cont = Container<Data, AllocationPolicy<Data>>;

  public:
    Insertable();
    Insertable(const Cont &rhs);
    Insertable(Cont &&rhs);
    Insertable(const Insertable &rhs);
    Insertable(Insertable &&rhs);
    Insertable(Data &&rhs);
};

#include "insertion.h"  // Free function, but connected to this class

#undef HDR_
#endif
```

../25/insertable/insertion.h

```cpp
#ifndef INCLUDED_INSERTIONT_
#define INCLUDED_INSERTIONT_

template <typename Out, class Insertable>
std::ostream &operator<<(Out &out, const Insertable &ins)
// Just the declaration here too? Wasn't sure; see ex. 26 for the code.

#endif
```

Exercise 26

../26/insertable/insertable.h

```cpp
#ifndef INCLUDED_INSERTABLE_
#define INCLUDED_INSERTABLE_

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

template <
          typename Data,
          template <typename, typename> class Container = std::vector,
          template <typename> class AllocationPolicy = std::allocator
         >
class Insertable: public Container<Data, AllocationPolicy<Data>>
{
  using Cont = Container<Data, AllocationPolicy<Data>>;

  public:
    Insertable();
    Insertable(const Cont &rhs);
    Insertable(Cont &&rhs);
    Insertable(const Insertable &rhs);
    Insertable(Insertable &&rhs);
    Insertable(Data &&rhs);
};

// Constructors just call constructor of underlying type
HDR_
Insertable<Data, Container, AllocationPolicy>::Insertable()
  : Cont()
{};
HDR_
Insertable<Data, Container, AllocationPolicy>::Insertable(const Cont &rhs)
  : Cont(rhs)
{};
HDR_
Insertable<Data, Container, AllocationPolicy>::Insertable(Cont &&rhs)
  : Cont(rhs)
{};
HDR_
Insertable<Data, Container, AllocationPolicy>::Insertable(const Insertable &rhs)
  : Cont(rhs)
{};
HDR_
Insertable<Data, Container, AllocationPolicy>::Insertable(Insertable &&rhs)
  : Cont(rhs)
{};
HDR_
Insertable<Data, Container, AllocationPolicy>::Insertable(Data &&rhs)
  : Cont(rhs)
{};

#include "insertion.h"  // Free function, but connected to this class

#undef HDR_
#endif
```

../26/insertable/insertion.h

```cpp
#ifndef INCLUDED_INSERTIONT_
#define INCLUDED_INSERTIONT_

```

```
template <typename Out, class Insertable>
Out &operator<<(Out &out, const Insertable &ins)
{
  for (auto el: ins)
    out << el << '\n';
  return out;
};

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

../26/main.ih

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

using namespace std;

#include <iostream>
#include <vector>

#include "insertable/insertable.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 28

../28/basictraits/basictraits.h

```
#ifndef INCLUDED_BASIC_
#define INCLUDED_BASIC_

template<typename T>
class BasicTraits
{
  template<typename T2>
  struct Basic
  {
    typedef T2 Type;
    enum
    {
      isPlain = 1,
      isPointer = 0,
      isRef = 0,
      isRRef = 0
    };
  };

  template<typename T2>
  struct Basic<T2 *> //pointer to a type
  {
    typedef T2 Type;
    enum
    {
      isPlain = 0,
      isPointer = 2,
      isRef = 0,
      isRRef = 0
    };
  };

  template<typename T2>
  struct Basic<T2 &> //reference to a type
  {
    typedef T2 Type;
    enum
    {
      isPlain = 0,
      isPointer = 0,
      isRef = 3,
      isRRef = 0
    };
  };

  template<typename T2>
  struct Basic<T2 &&> //rvalue ref to a type
  {
    typedef T2 Type;
    enum
    {
      isPlain = 0,
      isPointer = 0,
      isRef = 0,
      isRRef = 4
    };
   };

 public:

   BasicTraits(BasicTraits const &other) = delete;

```

```
    typedef typename Basic<T>::Type ValueType;
    typedef ValueType *PtrType;
    typedef ValueType &RefType;
    typedef ValueType &&RvalueRefType;

    //template<typename T2>
    BasicTraits(ValueType input);
    BasicTraits(ValueType *input);



    ValueType value()
    {
      return d_value;
    }

    enum
    {
      isPlainType = Basic<T>::isPlain,
      isPointerType = Basic<T>::isPointer,
      isRefType = Basic<T>::isRef,
      isRRefType = Basic<T>::isRRef
    };

  private:

     ValueType d_value;
};

template<typename T>
BasicTraits<T>::BasicTraits(ValueType input)
{
     d_value = input;
}
template<typename T>
BasicTraits<T>::BasicTraits(ValueType *input)
{
     d_value = *input;
}





#endif
```

../28/main.ih

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

#include "basictraits/basictraits.h"

#include <iostream>

using namespace std;
```

../28/main.cc

```
#include "main.ih"

int main(int argc, char const **argv)
{
  int plain = 3;
  auto ptr = &plain;

  cout << "plain :" << BasicTraits<int>::isPlainType << '\n'
       << "pointer :" << BasicTraits<int*>::isPointerType << '\n'
       << "reference :" << BasicTraits<int&>::isRefType << '\n'
       << "rvalue reference :" << BasicTraits<int&&>::isRRefType << '\n';

   BasicTraits<int>   BTobject1(ptr);
   BasicTraits<int*>  BTobject2(ptr);
   BasicTraits<int&>  BTobject3(ptr);
   BasicTraits<int&&> BTobject4(ptr);

   cout << BTobject1.value() << '\n'
        << BTobject2.value() << '\n'
        << BTobject3.value() << '\n'
        << BTobject4.value() << '\n';
}
```

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.