C++ Exercises
Set 4

Author(s): Channa Dias Perera, Alex Swaters, Ivan Yanakiev

16:23

March 8, 2023

25

In this exercise, we write an interface for a template class that is used for Storage of elements that can be inserted into an ostream. This template class is configurable by a policy that determines its storage strategy.

Listing 1: Insertable/Insertable.h

```
#ifndef SET3_INSERTABLE_H
#define SET3_INSERTABLE_H

#include <ostream>
                                    // Container is meant to be a template class
                                    // like vector, deque, list, etc.

                                    // Data is the data that would be contained
                                    // inside container. Assumes that Data is
                                    // insertable into a stream.

                                    // Insertable is designed such that if inserted
                                    // into an ostream, a space seaparated list
                                    // of the data elements are put into the ostream
template <template <typename> class Container, typename Data>
struct Insertable : Container<Data>
{
  Insertable();
  Insertable(Insertable const &other);
  Insertable(Insertable &&tmp);

                                      // Construct insertable based of
                                      // existing containers
  Insertable(Container<Data> const &cont);
  Insertable(Container<Data> &&tmpCont);

                                    // Create a container with a single element
  Insertable(Data const &elem);
  Insertable(Data &&tmpElem);
};

                                    // Output the elements held in insertable
                                    // as a space separated list to out.
template <template <typename> class Container, typename Data>
std::ostream &operator<<(
  std::ostream &out, Insertable<Container, Data> insertable
);

#endif //SET3_INSERTABLE_H
```

26

In this exercise, we write an implementation of the interface defined in exercise 25.

Listing 2: Insertable/Insertable.h

```
#ifndef SET3_INSERTABLE_H
#define SET3_INSERTABLE_H

#include <ostream>
#include <iterator>

                                // Container is meant to be a template class
                                // like vector, deque, list, etc.

                                // Data is the data that would be contained
                                // inside container. Assumes that Data is
                                // insertable into a stream.

                                // Insertable is designed such that if inserted
                                // into an ostream, a space seaparated list
                                // of the data elements are put into the ostream
template <template <typename> class Container, typename Data>
struct Insertable : Container<Data>
{
                                  // Defined in ctors.f
  Insertable();
  Insertable(Insertable<Container, Data> const &other);
  Insertable(Insertable<Container, Data> &&tmp);
                                  // Construct insertable based of
                                  // existing containers
  Insertable(Container<Data> const &cont);
  Insertable(Container<Data> &&tmpCont);

                                  // Create a container with a single element
  Insertable(Data const &elem);
  Insertable(Data &&tmpElem);
};

                                // Output the elements held in insertable
                                // as a space separated list to out.
template <template <typename> class Container, typename Data>
std::ostream &operator<<(
  std::ostream &out, Insertable<Container, Data> insertable
);

#include "ctors.f"
#include "insertion.f"

#endif //SET3_INSERTABLE_H
```

Listing 3: Insertable/ctors.f

```
template <template <typename> class Container, typename Data>
Insertable<Container, Data>::Insertable()
{}

template <template <typename> class Container, typename Data>
Insertable<Container, Data>::Insertable(
  Insertable<Container, Data> const &other
)
  : Container<Data>(other)
{}

template <template <typename> class Container, typename Data>
Insertable<Container, Data>::Insertable(Insertable<Container, Data> &&tmp)
  : Container<Data>(tmp)
{}

template <template <typename> class Container, typename Data>
Insertable<Container, Data>::Insertable(Container<Data> const &cont)
  : Container<Data>(cont)
{}

template <template <typename> class Container, typename Data>
Insertable<Container, Data>::Insertable(Container<Data> &&tmpCont)
```

```
  : Container<Data>(tmpCont)
{}

template <template <typename> class Container, typename Data>
Insertable<Container, Data>::Insertable(Data const &elem)
                                // Cannot do Container<Data>(elem) since if
                                // is_same<Data, size_type> then we construct
                                // container of elem Data{} elements.
  : Container<Data>(1, elem)
{}

template <template <typename> class Container, typename Data>
Insertable<Container, Data>::Insertable(Data &&tmpElem)
  : Container<Data>(1, tmpElem)
{}
```

Listing 4: Insertable/insertion.f

```
template <template <typename> class Container, typename Data>
std::ostream &operator<<(
  std::ostream &out, Insertable<Container, Data> insertable
)
{
  copy(
    insertable.begin(), insertable.end(),
    std::ostream_iterator<Data>(out, " ")
  );
  return out;
}
```

27

In this exercise, we write a member template for a class that stores a vector of string representations of numbers. This member template takes in a type (as a template argument) and an index and returns the number in the vector, converted to that type.

Listing 5: IsIntegral/IsIntegral.h

```
#ifndef SET4_ISINTEGRAL_H
#define SET4_ISINTEGRAL_H

                                // Trait class that checks if the Type supplied is
                                // an integer
template <typename Type>
struct IsIntegral
{
                                // To check if an integral type is supplied, we
                                // see if the type does not support non integer
                                // numbers.

                                // 0.5 will be rounded to 0 if the type is
                                // integral, and thus, integral = 1 + 0 == 1 ->
                                // true
  enum
  {
    integral = 1 + static_cast<Type>(0.5) == 1
  };
};

#endif //SET4_ISINTEGRAL_H
```

Listing 6: Data/Data.h

```
#ifndef SET4_DATA_H
#define SET4_DATA_H

#include "../IsIntegral/IsIntegral.h"
```

```
#include <string>
#include <vector>
#include <stdexcept>

                            // A plain class that contains a vector of strings
                            // These strings are character representations of
                            // numbers.

                            // Data contains a member template get returns
                            // vector elements after a conversion.
class Data
{
  std::vector<std::string> d_data;

                           // Support struct to determine the return type of
                           // get
  template <typename Type>
  struct GetReturnType;

  public:
                          // get takes in an index (idx) of the vector to return
                          // after conversion to Type.
    template <typename Type>
    typename Data::GetReturnType<Type>::type get(size_t idx);
};

                          // Requests to conversions other than string return
                          // a value to the Type requested.
template <typename Type>
struct Data::GetReturnType
{
  using type = Type;
};
                          // string is used to signal that a reference to the
                          // value as the specified index is needed instead.
template <>
struct Data::GetReturnType<std::string>
{
  using type = std::string const &;
};

                         // We specialize get for strings, to return the data
                         // directly.
template<>
typename  Data::GetReturnType<std::string>::type
  Data::get<std::string>(size_t idx)
{
  return d_data[idx];
}

                         // Otherwise, we perform the conversion
template <typename Type>
typename  Data::GetReturnType<Type>::type Data::get(size_t idx)
try
{
                        // We perform the correct conversion based on checking for
                        // the types
  if (IsIntegral<Type>::integral)
                        // We always convert to unsigned long long, as it captures
                        // the largest numbers possible. If the type request is
                        // signed and d_data[idx] is signed, the conversion results
                        // in the correct signed result, due to overflow.
      return std::stoull(d_data[idx]);
  else
                        // Handle floating point types.
    return std::stold(d_data[idx]);
}
catch (std::invalid_argument &excep)
{
                  // If conversion cannot be performed, we return Type{}
  return Type{};
}
```

```
#endif //SET4_DATA_H
```

28

In this exercise, we design a trait class that determines whether the supplied type is a plain type, a pointer type, a lvalue reference or a rvalue reference.

Listing 7: makeConstPtr.h

```
#ifndef SET4_MAKECONSTPTR_H
#define SET4_MAKECONSTPTR_H

#include "Trait/Trait.h"

template<typename Type>
typename Trait<Type>::BasicType const *makeConstPtr(Type arg) {
  if constexpr (Trait<Type>::value == 2)
    return arg;
  else if constexpr (Trait<Type>::value == 3)
    return &arg;
  else
    return new typename Trait<Type>::BasicType{arg};
}

#endif //SET4_MAKECONSTPTR_H
```

Listing 8: Trait/Trait.h

```
#ifndef SET4_TRAIT_H
#define SET4_TRAIT_H

                              // This is a trait class used to determine whether
                              // a Type is a plain type, a pointer type, a
                              // lvalue reference or a rvalue reference.

                              // Puts 1,2,3,4 in value, respectively

template <typename Type>
struct Trait
{
  using BasicType = Type;
  enum
  {
    value = 1
  };
};

template <typename Type>
struct Trait<Type *>
{
  using BasicType = Type;
  enum
  {
    value = 2
  };
};

template <typename Type>
struct Trait<Type &>
{
  using BasicType = Type;
  enum
  {
    value = 3
  };
};
template <typename Type>
struct Trait<Type &&>
{
```

```
  using BasicType = Type;
  enum
  {
    value = 4
  };
};

#endif //SET4_TRAIT_H
```

Listing 9: main.ih

```
#include "Trait/Trait.h"
#include "makeConstPtr.h"

#include <iostream>

using namespace std;
```

Listing 10: main.cc

```
#include "main.ih"

int main()
{
  int var = 1;
  int *ip = &var;
  int &ilr = var;
  int &&irr = 2;

  cout << Trait<decltype(var)>::value << '\n'
    << Trait<decltype(ip)>::value << '\n'
    << Trait<decltype(ilr)>::value << '\n'
    << Trait<decltype(irr)>::value << '\n';

  auto ptr = makeConstPtr(var);

  int plain = 3;
  ptr = &plain;
  cout << *ptr;
}
```

29

In this exercise, we implement an operator+ in the case where std::plus does not exist.

Listing 11: plus/plus.h

```
#ifndef SET4_PLUS_H
#define SET4_PLUS_H

                                    // Plus is a template functor that
                                    // performs addition between two
                                    // types (assuming such an operation is
                                    // available).
template <typename Type>
struct plus
{
  Type operator()(Type const &lhs, Type const &rhs) const;
};

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

#endif //SET4_PLUS_H
```

INCONSISTENT! Prefer the 1st.

Listing 12: Expr/Expr.h

```
#ifndef SET4_EXPR_H
#define SET4_EXPR_H

#include "../plus/plus.h"

                                    // Forward declare Expr
template <typename LHS, typename RHS, template <typename> class op>
struct Expr;

                                    // Instead of using std::plus, we use our own
                                    // implementation
template <typename LHS, typename RHS>
auto operator+(LHS const &lhs, RHS const &rhs)
{
  return Expr<LHS, RHS, plus> { lhs, rhs };
}
#endif //SET4_EXPR_H
```

30

In this exercise, we design a generic iterator class.

Listing 13: Iterator/Iterator.h

```
#ifndef SET4_ITERATOR_H
#define SET4_ITERATOR_H

#define ItTemplate template <typename IData, \
    template<typename> class Container>
#define FreeFuncTemplate template <typename IData, \
    template<typename> class Container>
#define ItType Iterator<IData, Container>

#include <iterator>

                                        // Forward declare template for free
                                        // operators to reference

ItTemplate
struct Iterator;

                                        // Forward declare the free operators to
                                        // bind in Iterator template
FreeFuncTemplate
bool operator==(Iterator<IData, Container> const &lhs,
        Iterator<IData, Container> const &rhs);

FreeFuncTemplate
auto operator<=>(Iterator<IData, Container> const &lhs,
        Iterator<IData, Container> const &rhs);

FreeFuncTemplate
std::ptrdiff_t operator-(Iterator<IData, Container> const &lhs,
        Iterator<IData, Container> const &rhs);

ItTemplate
struct Iterator
{
  private:
    typename Container<IData *>::iterator d_current;
  public:
                                        // Using declarations needed for
                                        // random access iterators
    using difference_type = std::ptrdiff_t;
    using value_type = IData;
    using pointer = IData *;
    using reference = IData &;
    using iterator_category = std::random_access_iterator_tag;
```

```
    Iterator() = default;
    Iterator(typename Container<IData *>::iterator const &current);


                                        // Specialize the operators with
                                        // this template's type args to
                                        // bind them.
    friend bool operator==<IData, Container>(
      Iterator<IData, Container> const &lhs,
      Iterator<IData, Container> const &rhs
    );

    friend auto operator<=><IData, Container>(
      Iterator<IData, Container> const &lhs,
      Iterator<IData, Container> const &rhs
    );

    friend difference_type operator-<IData, Container>(
      Iterator<IData, Container> const &lhs,
      Iterator<IData, Container> const &rhs
    );

                                    // Standard operations needed for Rand.
                                    // access iterators
    Iterator &operator++();
    Iterator operator++(int);
    Iterator &operator--();
    Iterator operator--(int);

    Iterator operator+(difference_type diff) const;
    Iterator operator-(difference_type diff) const;

    Iterator &operator+=(difference_type diff);
    Iterator &operator-=(difference_type diff);

    reference operator*() const;
    pointer operator->() const;
};

#include "iteratorImpl.f"

#undef ItTemplate
#undef ItType
#endif
```

Listing 14: Iterator/iteratorImpl.f

```
ItTemplate
inline typename ItType::reference ItType::operator*() const
{
  return **d_current;
}

ItTemplate
inline typename ItType::pointer ItType::operator->() const
{
  return *d_current;
}

ItTemplate
ItType &ItType::operator-=(Iterator::difference_type diff)
{
  d_current -= diff;
  return *this;
}

ItTemplate
ItType &Iterator<IData, Container>::operator+=(Iterator::difference_type diff)
{
  d_current += diff;
  return *this;
}
```

```
ItTemplate
inline ItType ItType::operator-(Iterator::difference_type diff) const
{
  return Iterator(d_current - diff);
}

ItTemplate
inline ItType ItType::operator+(Iterator::difference_type diff) const
{
  return Iterator(d_current + diff);
}

ItTemplate
  ItType ItType::operator--(int)
{
  Iterator temp(*this);
  --d_current;
  return temp;
}

ItTemplate
  ItType &ItType::operator--()
{
  --d_current;
  return *this;
}

ItTemplate
  ItType ItType::operator++(int)
{
  Iterator temp(*this);
  ++d_current;
  return temp;
}

ItTemplate
  ItType &ItType::operator++()
{
  ++d_current;
  return *this;
}

ItTemplate
inline std::ptrdiff_t operator-(ItType const &lhs, ItType const &rhs)
{
return lhs.d_current - rhs.d_current;
}

ItTemplate
auto operator<=>(ItType const &lhs, ItType const &rhs)
{
  return lhs.d_current <=> rhs.d_current;
}

ItTemplate
bool operator==(ItType const &lhs, ItType const &rhs)
{
  return lhs.d_current == rhs.d_current;
}

ItTemplate
ItType::Iterator(typename Container<IData *>::iterator const &current)
: d_current(current)
  {}
```

Listing 15: Storage/Storage.h

```
#ifndef SET4_STORAGE_H
#define SET4_STORAGE_H

#include "../Iterator/Iterator.h"
```

```
#include <vector>

template <typename Data>
class Storage
{
  std::vector<Data *> d_storage;  // store the pointers to the data
  public:
    using iterator = Iterator<Data, std::vector>;
    using reverse_iterator = std::reverse_iterator<iterator>;

    void push_back(Data *data);

    iterator begin();
    iterator end();
    reverse_iterator rbegin();
    reverse_iterator rend();
};

template<typename Data>
inline void Storage<Data>::push_back(Data *data)
{
  d_storage.push_back(data);
}

template <typename Data>
inline typename Storage<Data>::iterator Storage<Data>::begin()
{
  return iterator(d_storage.begin());
}

template <typename Data>
inline typename Storage<Data>::iterator Storage<Data>::end()
{
  return iterator(d_storage.end());
}

template <typename Data>
inline typename Storage<Data>::reverse_iterator Storage<Data>::rbegin()
{
  return reverse_iterator(d_storage.rbegin());
}

template <typename Data>
inline typename Storage<Data>::reverse_iterator Storage<Data>::rend()
{
  return reverse_iterator(d_storage.rend());
}

#endif //SET4_STORAGE_H
```

Running the program asked for in the exericse, we have the following output:

```
5 a.out
a.out april february january june march may
may march june january february april a.out
```

31

In this exercise, we explore the use of expression templates.

It will be a temporary VI (say tmp) objects, who has N elements with tmp[i] == v1[i] + v2[i] + v3[i] + v4[i].

There are 4 * (4 + 1) = 20 index operations.

32

In this exercise, we define an operator that can concatenate two tuples.

Listing 16: TupleMod/TupleMod.h

```
#ifndef SET4_TUPLEMOD_H
#define SET4_TUPLEMOD_H

#include <tuple>
#include <cstddef>
#include <utility>

                            // Operator to create new tuple, with the elements
                            // of tuple rhs appended to the elements of tuple
                            // lhs.
template <typename ...Lhs, typename ...Rhs>
auto operator+(std::tuple<Lhs ...> lhs, std::tuple<Rhs ...> rhs);

                              // TupleMod is a class that allows tuples to be
                              // "modified" to contain extra items.

                              // Tuple is the base tuple, and via add, you can
                              // add other items to it.
template <typename Tuple>
class TupleMod
{
  Tuple const &d_tuple;

  public:
    TupleMod(Tuple const &tuple);

                            // Add takes in a series of arguments and returns a
                            // tuple that is d_tuple with the supplied arguments
                            // appended to it.
    template <typename ...AddParams>
    auto add(AddParams &&...addParams);
  private:
                            // Workhorse function which merges two tuples.
    template <typename ...AddParams, size_t ...thisIdxs, size_t ...addedIdxs>
    auto mergeTuple(
      std::index_sequence<thisIdxs...>,
      std::tuple<AddParams...> tToAdd, std::index_sequence<addedIdxs...>
    );
};

template <typename Tuple>
TupleMod<Tuple>::TupleMod(Tuple const &tuple)
  : d_tuple(tuple)
{}

#include "add.f"
#include "operatorAdd.f"

#endif //SET4_TUPLEMOD_H
```

Listing 17: TupleMod/add.f

```
                            // To unpack d_tuple, we need a parameter
                            // pack of indices. However, we cannot simply
                            // pass in the params received by add, since
                            // this function cannot know where the indices
                            // end and the parameters start.

                            // To fix this, we pass a tuple containing the
                            // the params. However, we need the indices of
                            // this tuple as well then.

                            // We place the tuple parameter in between the
                            // indices parameters, to disambiguate index
                            // parameters.
template <typename Tuple>
template <typename ...AddParams, size_t ...thisIdxs, size_t ...addedIdxs>
auto TupleMod<Tuple>::mergeTuple(
  std::index_sequence<thisIdxs...>,
  std::tuple<AddParams...> tToAdd, std::index_sequence<addedIdxs...>
```

```
)
{
                                    // We ensure that we get the value types and not
                                    // references to types, in this manner, so when
                                    // std::get is called, they automatically copy
                                    // the value they reference.
  return std::tuple<
    typename std::tuple_element<thisIdxs, Tuple>::type ...,
    AddParams...
  >
  {
    std::get<thisIdxs>(d_tuple)..., std::get<addedIdxs>(tToAdd)...
  };
}

template<typename Tuple>
template<typename ...AddParams>
auto TupleMod<Tuple>::add(AddParams &&...addParams)
{
                                    // Make index sequences for d_tuple and
                                    // the addedParams tuple.
  return mergeTuple(
    std::make_index_sequence<std::tuple_size<Tuple>::value>{},
    std::tuple<AddParams...>(addParams...),
    std::make_index_sequence <sizeof... (AddParams)>{}
  );
}
```

Listing 18: TupleMod/operatorAdd.f

```
                                // Convenience function to get a copy of a
                                // value from an lvalue reference
template <typename Type>
Type getValue(Type const &arg)
{
  return arg;
}

                                // Workhorse function for tuple addition,
                                // utilizing TupleMod
template <typename ...Lhs, typename ...Rhs, size_t ...idxs>
auto addToTuple(
  std::tuple<Lhs ...> lhs, std::tuple<Rhs ...> rhs,
  std::index_sequence<idxs...>
)
{
                                // Create the "base" tuple to modify
  TupleMod<std::tuple<Lhs...>> tModL{lhs};

                                // Unpack the elements in rhs and get copies of
                                // these elements, to pass to add.
  return tModL.add(getValue(std::get<idxs>(rhs)) ...);
}


template <typename ...Lhs, typename ...Rhs>
auto operator+(std::tuple<Lhs ...> lhs, std::tuple<Rhs ...> rhs)
{
                               // We call the auxillary function addToTuple with
                               // so it has the indexes for rhs as a param. pack
  return addToTuple(lhs, rhs, std::index_sequence_for<Rhs ...>{});
}
```