C++ Exercises
Set 5

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

14:07

March 14, 2023

33

In this exercise, we create a concept BasicMath which verifies that two types passed to it can use the basic math operations upon themselves.

Listing 1: BasicMath/BasicMath.h

```
#ifndef SET4_BASICMATH_H
#define SET4_BASICMATH_H

                        // Concept that ensures that two types have basic
                        // binary math operators established in one direction
                        // i.e: not rhs op lhs.
template <typename Lhs, typename Rhs>
concept BinaryMath =
  requires(Lhs lhs, Rhs rhs){
    lhs + rhs;
    lhs - rhs;
    lhs * rhs;
    lhs / rhs;
  };

                        // Concept that basic math operators exist between two
                        // types, with either type being used as the Lhs
template <typename Type1, typename Type2>
concept BasicMath =
                        // Check binary operations work in either dir.
  BinaryMath<Type1, Type2> and BinaryMath<Type2, Type1> and
  requires(Type1 type1, Type2 type2)
  {
                        // Check unary ops.
    -type1;
    -type2;
  };

#endif //SET4_BASICMATH_H
```

Listing 2: basicMathTest.h

```
#ifndef SET4_BASICMATHTEST_H
#define SET4_BASICMATHTEST_H

#include "BasicMath/BasicMath.h"

                                    // A demo template function to showcase
                                    // the BasicMath concept.
template <typename Lhs, typename Rhs>
  requires(BasicMath<Lhs, Rhs>)
auto basicMathTest(Lhs lhs, Rhs rhs)
{
  return -(lhs / (rhs - lhs)) + (lhs * rhs);
}

#endif //SET4_BASICMATHTEST_H
```

Listing 3: main.ih

```
#include "basicMathTest.h"
#include <iostream>

using namespace std;
```

Listing 4: main.cc

```
#include "main.ih"

int main()
{
  basicMathTest(1, 2); // Compiles
//  basicMathTest(1, "BasicMathTest"s);  // WC: int, string doesn't satisfy
                                         // BasicMath
}
```

34

In this exercise, we define a concept that checks that a template type can be indexed.

Listing 5: Indexable/Indexable.h

```
#ifndef SET5_INDEXABLE_H
#define SET5_INDEXABLE_H

#include <concepts>

                                  // This concept ensures that a template type
                                  // is indexable and returns a reference
                                  // to the data it stores
template <template <typename> class Container, typename Data>
concept Indexable =
  requires(Container<Data> container)
  {
    { container[0] } -> std::same_as<Data &>;
  };

#endif //SET5_INDEXABLE_H
```

35

In this exercise, we explore how template can use concepts.

Listing 6: Dereferenceable/Dereferenceable.h

```
#ifndef SET5_DEREFERENCEABLE_H
#define SET5_DEREFERENCEABLE_H

                                  // A concept which ensures that some type
                                  // supports the dereference operator.
template <typename Type>
concept Dereferenceable =
  requires(Type obj)
  {
    *obj;
  };

#endif //SET5_DEREFERENCEABLE_H
```

Listing 7: Deref1/Deref1.h

```
#ifndef SET5_DEREF1_H
#define SET5_DEREF1_H
```

```
#include "../Dereferenceable/Dereferenceable.h"

                                // Class demoing using a concept explicitly, in
                                // a requires clause
template <typename Type>
  requires Dereferenceable<Type>
class Deref1
{
    Type d_deref;

  public:
    Deref1(Type deref);
};

template<typename Type>
  requires Dereferenceable<Type>
Deref1<Type>::Deref1(Type deref)
  : d_deref(deref)
{}

#endif //SET5_DEREF1_H
```

Listing 8: Deref2/Deref2.h

```
#ifndef SET5_DEREF2_H
#define SET5_DEREF2_H

#include "../Dereferenceable/Dereferenceable.h"

                                // Class demoing using a concept explicitly, in
                                // a requires clause
template <Dereferenceable Type>
class Deref2
{
    Type d_deref;

  public:
    Deref2(Type deref);
};

template<Dereferenceable Type>
Deref2<Type>::Deref2(Type deref)
  : d_deref(deref)
{}

#endif //SET5_DEREF2_H
```

Listing 9: dereference1.h

```
#ifndef SET5_DEREFERENCE1_H
#define SET5_DEREFERENCE1_H

#include "Dereferenceable/Dereferenceable.h"

                                // A function template showing the use of a
                                // concept directly when declaring the template
                                // type parameter.
template <Dereferenceable Deref>
auto dereference1(Deref toDeref)
{
return *toDeref;
}

#endif //SET5_DEREFERENCE1_H
```

Listing 10: dereference2.h

```
#ifndef SET5_DEREFERENCE2_H
#define SET5_DEREFERENCE2_H
```

```
#include "Dereferenceable/Dereferenceable.h"

                                // An implicit function template that shows the
                                // use a concept in the parameter list.
auto dereference2(Dereferenceable auto toDeref)
{
  return *toDeref;
}

#endif //SET5_DEREFERENCE2_H
```

Listing 11: main.ih

```
#include "Deref1/Deref1.h"
#include "Deref2/Deref2.h"

#include "dereference1.h"
#include "dereference2.h"

#include <iostream>

using namespace std;
```

Listing 12: main.cc

```
#include "main.ih"
                                // Demo Deref, dereference1, dereference2
int main()
{
  int var = 10;
  int *ip = &var;

  dereference1(ip);
  dereference2(ip);

  Deref1{ip};
  Deref2{ip};
}
```

36

This is OK. Use an existing requirement that directly matches the exercise

In this exercise, we explore defining requirements without concepts, for template functions / classes.

Listing 13: ComparingDemo.h

```
#ifndef SET5_COMPARINGDEMO_H
#define SET5_COMPARINGDEMO_H

                                // template function declaration which uses Lhs /
                                // Rhs such that they should be comparable.

template <typename Lhs, typename Rhs>
  requires
    requires(Lhs lhs, Rhs rhs)
    {
                                // We have to compare lhs / rhs to ensure
                                // commutativity of application (though not of
                                // result)
      lhs == rhs;
      rhs == lhs;

      lhs != rhs;
      rhs != lhs;

      lhs < rhs;
      rhs < lhs;

      lhs > rhs;
```

```
      rhs > lhs;

      lhs <= rhs;
      rhs <= lhs;

      lhs >= rhs;
      rhs >= lhs;
    }
auto compare();

                                    // template class delcaration with a template
                                    // type parameter which should be comparable
template <typename Type>
  requires
    requires(Type lhs, Type rhs)
    {
      lhs == rhs;
      lhs != rhs;

      lhs < rhs;
      lhs <= rhs;

      lhs > rhs;
      lhs <= rhs;
    }
class Compare;

#endif //SET5_COMPARINGDEMO_H
```

37

In this exercise, we explore the use of nested concepts.

Listing 14: Concepts.h

```
#ifndef SET5_CONCEPTS_H
#define SET5_CONCEPTS_H

                            // Concept that checks that Type works with
                            // addition / subtraction.
template <typename Type>
concept Add =
requires(Type lhs, Type rhs)
{
  lhs + rhs;
  lhs - rhs;
};

                            // Concept that checks that Type works with
                            // multiplication / division.
template <typename Type>
concept Mul =
requires(Type lhs, Type rhs)
{
  rhs * lhs;
  rhs / lhs;
};

                            // A concept that checks whether Type support
                            // Addition / Multiplication / both.
template <typename Type>
concept AddMul = Add<Type> or Mul<Type>;

                            // A concept that checks whether Type supports
                            // Addition / Multiplication (and only one).
template <typename Type>
concept AddOrMul =
(Add<Type> and not Mul<Type>) or
(Mul<Type> and not Add<Type>);

#endif //SET5_CONCEPTS_H
```

In this exercise, we apply concepts to variadic template directly.

Listing 15: Add/Add.h

```
#ifndef SET5_ADD_H
#define SET5_ADD_H

#include "../BasicMath/BasicMath.h"

#include <iostream>

                                    // Struct that defines an operation static
                                    // function that operates on two
                                    // operands that must support BasicMath.
                                    // The operation uses addition.
template <typename Lhs, typename Rhs>
  requires BasicMath<Lhs, Rhs>
struct Add
{
  static void operation(Lhs lhs, Rhs rhs);
};

                                // Output the result of addition of lhs, rhs to
                                // cout.
template <typename Lhs, typename Rhs>
  requires BasicMath<Lhs, Rhs>
void Add<Lhs, Rhs>::operation(Lhs lhs, Rhs rhs)
{
  std::cout << (lhs + rhs) << ' ';
}

#endif //SET5_ADD_H
```

Listing 16: math.h

```
#ifndef SET5_MATH_H
#define SET5_MATH_H

                                    // math applies Operation between the first
                                    // argument and each of the subsequent
                                    // arguments. So for arguments:
                                    // 1) arg1, arg2
                                    // 2) arg1, arg3
                                    //    ...
                                    // N) arg1, argN
template <template <typename, typename> class Operation, typename ...Args>
void math(Args &&...args);

                                    // Base case: If we only have 1 argument, do
                                    // nothing
template <template <typename, typename> class Operation, typename Last>
void math(Last last)
{};

                                    // General case: Apply operation to First and
                                    // Second, then call math on First and Rest.
template <
  template <typename, typename> class Operation,
  typename First, typename Second, typename ...Rest
>
void math(First first, Second second, Rest &&...rest)
{
  Operation<First, Second>::operation(first, second);
  math<Operation>(first, rest...);
}

#endif //SET5_MATH_H
```

7

In this exercise, we define a robust Concept to check that an iterator is a random access iterator.

39?

Listing 17: IteratorConcepts/BasicIterators.h

```
#ifndef SET5_BASICITERATORS_H
#define SET5_BASICITERATORS_H

#include <concepts>

                                        // Concept to check if increment op.
                                        // exists on Type
template <typename Type>
concept Incrementable =
requires(Type type)
{
  {++type} -> std::same_as<Type &>;
  {type++} -> std::same_as<Type>;
};

                                  // Concept to check if dereference op.
                                  // exists on Type
template <typename Type>
concept Dereferenceable =
requires(Type type)
{
  {*type} -> std::same_as<typename Type::value_type &>;
};

                                  // Concept to check if dereference op.
                                  // exists on Type, returning a const ref.
template <typename Type>
concept ConstDereferenceable =
requires(Type type)
{
  {*type} -> std::convertible_to<typename Type::value_type const &>;
};

                                    // Concept to check whether the Type has the
                                    // appropriate using declarations for an
                                    // iterator
template <typename Type>
concept HasIteratorUsings =
requires(Type type)
{
  typename Type::difference_type;
  typename Type::value_type;
  typename Type::pointer;
  typename Type::reference;
  typename Type::iterator_category;
};

                                  // Concept to check if the Type can work as a
                                  // input iterator
template <typename Type>
concept InIterator =
  std::equality_comparable<Type> and Incrementable<Type> and
  ConstDereferenceable<Type> and HasIteratorUsings<Type>;

                                  // Concept to check if the Type can work as a
                                  // output iterator
template <typename Type>
concept OutIterator =
  std::equality_comparable<Type> and Incrementable<Type> and
  Dereferenceable<Type> and HasIteratorUsings<Type>;

#endif //SET5_BASICITERATORS_H
```

No, Type returned by iter++ cannot be a reference

this is not sufficient you need more comparisons.

Listing 18: IteratorConcepts/FwdIterator.h

```
#ifndef SET5_FWDITERATOR_H
#define SET5_FWDITERATOR_H

#include "BasicIterators.h"

                                // Concept to check if the Type can work as a
                                // forward iterator
template <typename Type>
concept FwdIterator =
  InIterator<Type> and OutIterator<Type>;

#endif //SET5_FWDITERATOR_H
```

Listing 19: IteratorConcepts/BiIterator.h

```
#ifndef SET5_BIITERATOR_H
#define SET5_BIITERATOR_H

#include "FwdIterator.h"

                              // Concept to check if the Type can be decremented
template <typename Type>
concept Decrementable =
requires(Type type)
{
  {--type} -> std::same_as<Type &>;
  {type--} -> std::same_as<Type>;
};

                                // Concept to check if the Type can work as a
                                // Bidirectional iterator
template <typename Type>
concept BiIterator =
  FwdIterator<Type> and Decrementable<Type>;

#endif //SET5_BIITERATOR_H
```

Listing 20: IteratorConcepts/RndIterator.h

```
#ifndef SET5_RNDITERATOR_H
#define SET5_RNDITERATOR_H

#include "BiIterator.h"

                                // Concept to check if the Type supports addition
template <typename Type>
concept Addable =
requires(Type type)
{
  {type += 0} -> std::same_as<Type &>;
  {type + 0} -> std::same_as<Type>;
};

                                // Concept to check if the Type supports
                                // subtraction
template <typename Type>
concept Subtractable =
requires(Type type)
{
  {type -= 0} -> std::same_as<Type &>;
  {type - 0} -> std::same_as<Type>;
};


                                // Concept to check if the Type can work as a
                                // random access iterator
template <typename Type>
concept RndIterator =
  BiIterator<Type> and Subtractable<Type> and Addable<Type>
```

```
  and
  requires(Type lhs, Type rhs)
  {
      {lhs - rhs} -> std::convertible_to<int>;
  };

#endif //SET5_RNDITERATOR_H
```

Listing 21: sortWithConcepts.h

```
#ifndef SET5_SORTWITHCONCEPTS_H
#define SET5_SORTWITHCONCEPTS_H

#include "IteratorConcepts/RndIterator.h"

#include <algorithm>

                        // Sort, but the inputs are verified via concepts to
                        // be Random Access Iterators
template <RndIterator RAI>
void sortWithConcepts(RAI first, RAI last)
{
std::sort(first, last);
}

#endif //SET5_SORTWITHCONCEPTS_H
```

Listing 22: main.ih

```
#include "sortWithConcepts.h"

#include <vector>

using namespace std;
```

Listing 23: main.cc

```
#include "main.ih"

                    // Demo to show the use of the RndIterator concept
int main()
{
  vector<int> vi{1,2,3,4,3,2,1};

  sortWithConcepts(vi.begin(), vi.end());
}
```

40

In this exercise, we implement free operators for a nested iterator class, and ensure that those free operators refer to just the normal iterator (and not the reverse iterator) by using concepts.

Listing 24: Storage/NormalIterator.h

```
#ifndef SET5_NORMALITERATOR_H
#define SET5_NORMALITERATOR_H

#include <concepts>

                    // Forward declare Storage to compare against it's
                    // iterator type
template <typename Data>
class Storage;

                    // Used for free operators of Storage<Data>::iterator
                    // Prevents binding to any other iterator (such as the
                    // reverse iterator)
```

```
template <typename Iterator>
concept NormalIter =
  std::same_as<
    Iterator,
                          // Iterator must be the same iterator as in Storage
    typename Storage<typename Iterator::value_type>::iterator
  >;

#endif //SET5_NORMALITERATOR_H
```

Listing 25: Storage/Storage.h

```
#ifndef SET4_STORAGE_H
#define SET4_STORAGE_H

#include "NormalIterator.h"

#include <vector>

                                        // Free operator declarations for Storage
                                        // to reference as friends.
int operator-(NormalIter auto const &lhs, NormalIter auto const &rhs);
bool operator==(NormalIter auto const &lhs, NormalIter auto const &rhs);
auto operator<=>(NormalIter auto const &lhs, NormalIter auto const &rhs);
auto operator+(NormalIter auto const &lhs, int step);

template <typename Data>
class Storage
{
  std::vector<Data *> d_storage;

  public:
                                                // Iterators for d_storage
    struct iterator;
    using reverse_iterator = std::reverse_iterator<iterator>;

                                                // Add to d_storage
    void push_back(Data *data);
                                                // Get iterators for d_storage
    iterator begin();
    iterator end();
    reverse_iterator rbegin();
    reverse_iterator rend();
};

#include "Iterator.f"
#include "StorageImpl.f"
#include "IteratorImpl.f"
#include "IteratorFreeOpImpl.f"

#endif //SET4_STORAGE_H
```

Listing 26: Storage/Iterator.f

```
                                    // Definition of iterator nested class in
                                    // in Storage<Data>
template <typename Data>
struct Storage<Data>::iterator
{
                                    // Using decl. for random access iterator
    using iterator_category = std::random_access_iterator_tag;
    using difference_type   = std::ptrdiff_t;
    using value_type        = Data;
    using pointer           = value_type const *;
    using reference         = value_type const &;

  private:
    typename std::vector<Data *>::iterator d_current;

  public:
                                    // Constructor needed for sort
```

```
    iterator(typename std::vector<Data *>::iterator const &current);

                                    // Free operators
    friend auto operator<=><>(iterator const &lhxs, iterator const &rhs);
    friend bool operator==<>(iterator const &lhs, iterator const &rhs);
    friend int operator-<>(iterator const &lhs, iterator const &rhs);
    friend auto operator+<>(iterator const &lhs, int step);

                                    // Operators for moving through
                                    // the vector via iterators
    iterator operator-(int step) const;
    iterator &operator--();
    iterator operator--(int);
    iterator &operator++();
    iterator operator++(int);

    iterator &operator+=(int step);
    iterator &operator-=(int step);

                                    // Access operators
    Data &operator*();
    Data *operator->();

};
```

Listing 27: Storage/StorageImpl.f

```
template <typename Data>
inline Storage<Data>::iterator::iterator(
  typename std::vector<Data *>::iterator const &current)
  :
  d_current(current)
{}

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 d_storage.begin();
}

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

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

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

Listing 28: Storage/IteratorImpl.f

```
template <typename Data>
inline typename Storage<Data>::iterator &Storage<Data>::iterator::operator++()
{
  ++d_current;
  return *this;
```

```
}

template <typename Data>
inline typename Storage<Data>::iterator Storage<Data>::iterator::operator++(int)
{
  return iterator(d_current++);
}

template <typename Data>
inline typename Storage<Data>::iterator &Storage<Data>::iterator::operator--()
{
  --d_current;
  return *this;
}

template <typename Data>
inline typename Storage<Data>::iterator Storage<Data>::iterator::operator--(int)
{
  return iterator(d_current--);
}

template <typename Data>
inline typename Storage<Data>::iterator Storage<Data>::iterator::operator-(
  int step) const
{
  Storage<Data>::iterator ret{ *this };
  ret.d_current -= step;                    // avoids ambiguity
  return ret;
}

template <typename Data>
inline Data &Storage<Data>::iterator::operator*()
{
  return **d_current;
}

template <typename Data>
inline Data *Storage<Data>::iterator::operator->()
{
  return &**d_current;
}

template <typename Data>
inline typename Storage<Data>::iterator &Storage<Data>::iterator::operator+=(int
    step)
{
  d_current += step;
  return *this;
}

template <typename Data>
inline typename Storage<Data>::iterator &Storage<Data>::iterator::operator-=(int
    step)
{
  d_current -= step;
  return *this;
}
```

Listing 29: Storage/IteratorFreeOpImpl.f

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

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

int operator-(NormalIter auto const &lhs, NormalIter auto const &rhs)
{
```

```
  return lhs.d_current - rhs.d_current;
}

auto operator+(NormalIter auto const &lhs, int step)
{
  NormalIter auto ret{lhs };
  ret.d_current += step;
  return ret;
}
```

The program in the exercise sorts the command line arguments (incl. the program name). We slightly modify to also output the (space separated) results of the sort, right after each sort.

We separate the results of each sort with a newline.

With the arguments:
these are not 1 but several (>2) command line arguments, and a program named a.out, we have the following results:

(>2) a.out 1 are arguments but command line not several these
these several not line command but arguments are 1 a.out (>2)