C++ Exercises
Set 4, resend

Author(s): Pau Lopez, Peter Versluis

14:29

March 13, 2025

33

A concept that makes sure that basic math expressions are available for 2 types.

Listing 1: basicmath.hh

```
#ifndef BASIC_MATH_HH
#define BASIC_MATH_HH

// definition of BasicMath concept
template <typename T1, typename T2>
concept BasicMath =
    requires(T1 arg1, T2 arg2)
    {
        arg1 + arg2;
        arg1 - arg2;
        arg1 * arg2;
        arg1 / arg2;
        -arg1;
        -arg2;
    };

// just an empty function template parameter that requires BasicMath
template <typename T1, typename T2> requires BasicMath<T1, T2>
void fun(T1 arg1, T2 arg2) {};

#endif
```

Listing 2: main.cc

```
#include "basicmath.hh"
#include <string>

int main()
{
    int x = 0, y = 1;
    fun(x, y);

    // Example to demonstrate that the compiler complains when the operations
    // listed in BasicMath can't be performed by the 2 types
    // int num = 1;
    // std::string str = "abcd";
    // fun(num, str);
}
```

34

A concept that checks whether a class has the index operator and if it returns by reference

Listing 3: index.hh

```
#ifndef RETURN_REFERENCE_HH
#define RETURN_REFERENCE_HH
#include <concepts>

template <typename Type>
concept ReturnReference =
    requires(Type cont)
    {
        { cont[0] } -> std::same_as<typename Type::value_type &>;
    };

#endif
```


Creating a concept that checks whether a type is dereferenceable and showing how to use it in two different ways in template functions and classes.

Listing 4: dereferenceable.hh

```
#ifndef DEREFERENCEABLE_HH
#define DEREFERENCEABLE_HH

// a concept that checks if a Type supports the dereference operator
template <typename Type>
concept Dereferenceable =
    requires(Type type)
    {
        *type;
    };

#endif
```

Listing 5: main.ih

```
#include "dereferenceable.hh"
#include <iostream>

// an empty function that uses Dereferenceable in its template parameter list
template <Dereferenceable Type>
void fun1(Type tp) {};

// an empty function that uses Dereferenceable in the requirements
template <typename Type> requires Dereferenceable<Type>
void fun2(Type tp) {};

// a class that uses Dereferenceable in its template parameter list
template <Dereferenceable Type>
struct Class1
{
    void fun() const {};
};

// a class function that uses Dereferenceable in the requirements
template <typename Type> requires Dereferenceable<Type>
struct Class2
{
    void fun() const {};
};
```

Listing 6: main.cc

```
#include "main.ih"

int main()
{
    // calling fun1 and fun2 with a dereferenceable type
    int *dereferenceable = 0;
    fun1(dereferenceable);
```

```
    fun2(dereferenceable);

    // creating class1 and class2 with dereferenceable types
    Class1<int *> c1{};
    Class2<int *> c2{};

    // using the classes so the compiler doesn't complain
    c1.fun();
    c2.fun();

    // The following doesn't compile:
    //int not_dereferenceable = 0;
    //fun1(not_dereferenceable);
    //fun2(not_dereferenceable);
    //Class1<int> c1{};
    //Class2<int> c2{};
}
```

36

A concept that checks if two types are comparable and a function and class that take comparable types as template parameters.

Listing 7: comp.hh

```
// definition of the comparable concept
template <typename Lhs, typename Rhs>
concept Comparable =
    requires(Lhs lhs, Rhs rhs)
    {
        lhs <=> rhs;
    };

// a function template that takes 2 comparables template type arguments
// could return -1 if lhs < rhs, 0 if lhs == rhs and 1 otherwise.
template <typename Lhs, typename Rhs> requires Comparable<Lhs, Rhs>
int compare(Lhs lhs, Rhs rhs);

// a class template that takes 1 comparable template type argument
template <typename Type> requires Comparable<Type, Type>
class Compare;
```

37

A concept that checks if two operands satisfy certain conditions on arithmetical operations .

Listing 8: nest.hh

```
// In this implementation, we are not assuming that the types of
// the operands are equal

#ifndef NEST_HH
#define NEST_HH

template <typename Lhs, typename Rhs>
concept Addable =
    requires(Lhs lhs, Rhs rhs)
    {
        lhs + rhs;
        lhs - rhs;
    };

template <typename Lhs, typename Rhs>
concept Multipliable =
    requires(Lhs lhs, Rhs rhs)
    {
        lhs * rhs;
        lhs / rhs;
```

```
    };

template <typename Lhs, typename Rhs>
concept AddMul = Addable<Lhs, Rhs> or Multipliable<Lhs, Rhs>;

template <typename Lhs, typename Rhs>
concept AddOrMul = (Addable<Lhs, Rhs> and not Multipliable<Lhs, Rhs>) or
                   (Multipliable<Lhs, Rhs> and not Addable<Lhs, Rhs>);

#endif
```

39

A concept that checks whether a type has random iterator capabilities or not.

Listing 9: main.ih

```
#include "random.hh"
#include <vector>
#include <algorithm>
#include <utility>
using namespace std;

// A function that calls std::sort on iterators satisfying RndIterator concept
template <RndIterator It>
void rndSort(It &&begin, It &&end)
{
    sort(forward<It>(begin), forward<It>(end));
}
```

Listing 10: random.hh

```
#ifndef RND_IT_HH
#define RND_IT_HH

#include "bidirectional.i"
#include <concepts>

template <typename Type>
concept RndIterator = BiIterator<Type> and
    requires(Type lhs, Type rhs)
    {
        { lhs += 0 } -> std::same_as<Type &>;
        { lhs -= 0 } -> std::same_as<Type &>;
        { lhs + 0 } -> std::same_as<Type>;
        { lhs - 0 } -> std::same_as<Type>;
        { lhs - rhs } -> std::same_as<long int>;
    };

#endif
```

Listing 11: bidirectional.i

```
// should be a .hh file
#ifndef BI_IT_HH
#define BI_IT_HH

#include "forward.i"
#include <concepts>

template <typename Type>
concept BiIterator = FwdIterator<Type> and
    requires(Type type)
    {
        { --type } -> std::same_as<Type &>;
        { type-- } -> std::same_as<Type>;
    };

#endif
```

Listing 12: forward.i

5d

```
// should be a .hh file
#ifndef FWD_IT_HH
#define FWD_IT_HH

#include "input.i"
#include "output.i"

template <typename Type>
concept FwdIterator = InIterator<Type> or OutIterator<Type>;

#endif
```

Listing 13: input.i

```
// should be a .hh file
#ifndef IN_IT_HH
#define IN_IT_HH

#include "common.i"

template <typename Type>
concept InIterator = Comparable<Type> and
                     Incrementable<Type> and
                     ConstDereferenceable<Type>;

#endif
```

Listing 14: output.i

```
// should be a .hh file
#ifndef OUT_IT_HH
#define OUT_IT_HH

#include "common.i"

template <typename Type>
concept OutIterator = Comparable<Type> and
                      Incrementable<Type> and
                      Dereferenceable<Type>;

#endif
```

Listing 15: common.i

```
#ifndef COMMON_HH
#define COMMON_HH
#include <concepts>

template <typename Type>
concept Comparable =
    requires (Type lhs, Type rhs)
    {
        { lhs == rhs } -> std::same_as<bool>;
        { lhs != rhs } -> std::same_as<bool>;
    };

template <typename Type>
concept Dereferenceable =
    requires(Type type)
    {
        { *type } -> std::same_as<typename Type::value_type &>;
    };

template <typename Type>
concept ConstDereferenceable =
    requires(Type type)
    {
        // as suggested in the annotations
```

```
        { *type } ->  std::convertible_to<typename Type::value_type const &>;
    };

template <typename Type>
concept Incrementable =
    requires (Type type)
    {
        { ++type } -> std::same_as<Type &>;
        { type++ } -> std::same_as<Type>;
    };

#endif
```

Listing 16: main.cc

```
#include "main.ih"

int main()
{
    vector<int> vec = {1, 7, 3, 9, 2};

    rndSort(vec.begin(), vec.end());
}
```