we're in the advanced part of the course. please read the comment and use it in subsequent exercises

C++ Exercises
Set 3

FB

Author(s): Pau Lopez, Peter Versluis

12:19

March 1, 2025

17 8
18 8
19 8
20 8
21 8
23 8

17

A trait class that computes different attributes of an integer is implemented.

Listing 1: NrTrait.hh

8

```
#ifndef NR_TRAIT_HH
#define NR_TRAIT_HH

template <int Nr>
struct NrTrait
{
    enum
    {
        value = Nr,
        absolute = Nr < 0 ? -Nr : Nr,
        even = Nr % 2 == 0,
        by3 = Nr % 3 == 0,
        sum_of_digits = absolute < 10 ? absolute
                      : absolute % 10 + NrTrait<absolute / 10>::sum_of_digits,
        width = absolute < 10 ? 1 : 1 + NrTrait<absolute / 10>::width
    };
};

// specialization for case Nr = 0, needed for the recursion
template <>
struct NrTrait<0>
{
    enum
    {
        value = 0,
        absolute = 0,
        even = 1,
        by3 = 1,
        sum_of_digits = 0,
        width = 1
    };
};

#endif
```

Listing 2: main.cc

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

int main()
{
    // showing in a single cout statement what the value, evenness,
    // divisibility by 3 and width is of the value -1971962
    std::cout << "value: " << NrTrait<-1971962>::value << '\n'
              << "absolute value: " << NrTrait<-1971962>::absolute << '\n'
              << "even: " << NrTrait<-1971962>::even << '\n'
              << "divisible by 3: " << NrTrait<-1971962>::by3 << '\n'
```

```
                << "sum of digits: " << NrTrait<-1971962>::sum_of_digits << '\n'
                << "width: " << NrTrait<-1971962>::width << '\n';
}
```

18

8

A small meta program that shows the binary representation of a number.

Listing 3: Bin.hh

```
#ifndef BIN_HH
#define BIN_HH

#include <iosfwd>

template <size_t Nr>
struct Bin
{
    enum { value = (Nr & 1) + 10 * Bin<(Nr >> 1)>::value };
};

template <>
struct Bin<0>
{
    enum { value = 0 };
};

#endif
```

19

8

The insertion operator is overloaded for a variadic class template.

Listing 4: Chars.hh

PRAR!

```
#ifndef CHARS_HH
#define CHARS_HH

#include <tuple>
#include <ostream>

template <char ...Args>
class Chars
{
    // allow insertions into ostreams
    template <char ...Params>
    friend std::ostream& operator<<(std::ostream& out,
                                    Chars<Params...> const &chars);
};

template <char ...Args>
inline std::ostream &operator<<(std::ostream &out, Chars<Args...> const &chars)
{
    // Use unary left fold to insert each character into the stream
    return (out << ... << Args);
}


#endif
```

20

A class that merges 2 class templates.

Listing 5: chars/chars.hh

```
#ifndef CHARS_HH
#define CHARS_HH
```

PRAK

```
#include <tuple>
#include <ostream>

template <char ...Args>
class Chars
{
    // allow insertions into ostreams
    template <char ...Params>
    friend std::ostream& operator<<(std::ostream& out,
                                    Chars<Params...> const &chars);
};

template <char ...Args>
inline std::ostream &operator<<(std::ostream &out, Chars<Args...> const &chars)
{
    // Use unary left fold to insert each character into the stream
    return (out << ... << Args);
}


#endif
```

Listing 6: onechar/onechar.hh

```
#ifndef ONE_CHAR_HH
#define ONE_CHAR_HH

template <char Ch>
class OneChar
{};

#endif
```

Listing 7: merge.hh

```
#ifndef MERGE_HH
#define MERGE_HH

#include"chars/chars.hh"
#include "onechar/onechar.hh"

// Forward declaration for the general case
template <typename LHS, typename RHS>
class Merge;

// Specialization for merging two Chars templates
template <char ...Lhs, char ...Rhs>
class Merge<Chars<Lhs...>, Chars<Rhs...>>
{
    public:
        using CP = Chars<Lhs..., Rhs...>;
};

// Specialization for merging a Chars template with a OneChar template
template <char ...Lhs, char Rhs>
class Merge<Chars<Lhs...>, OneChar<Rhs>>
{
    public:
        using CP = Chars<Lhs..., Rhs>;
};

#endif
```

21

A class template that compares types of template parameters is developed.

Listing 8: type.hh

4

```
#ifndef TYPE_HH
#define TYPE_HH

// definition of Helper class
#include "helper.i"

// general case: more than 1 template argument
template <typename Needle, typename ...Types>
struct Type
{
    enum { located = TypeIdx<Needle, Types...>::value };
};

// base case: only 1 template argument
template <typename Needle>
struct Type<Needle>
{
    enum { located = 0 };
};


#endif
```

~CC. Use comment to explain (brief!) what's happening

Listing 9: helper.i

```
#include <type_traits>

template <typename Needle, typename First, typename ...Types>
struct TypeIdx
{
    enum
    {
        value = std::is_same<Needle, First>::value ?
                                    1 : 1 + TypeIdx<Needle, Types...>::value
    };
};

template <typename Needle, typename First>
struct TypeIdx<Needle, First>
{
    enum { value = std::is_same<Needle, First>::value };
};
```

NSC

23

The class template in ex21 is modified so the TypeIdx class is a private nested class of Type.

Listing 10: type.hh

```
#ifndef TYPE_HH
#define TYPE_HH


// general case: more than 1 template argument
template <typename Needle, typename ...Types>
class Type
{
    template <typename NeedleIdx, typename FirstIdx, typename ...TypesIdx>
    struct TypeIdx;

    template <typename NeedleIdx, typename FirstIdx>
    struct TypeIdx<NeedleIdx, FirstIdx>;

    public:
        enum { located = TypeIdx<Needle, Types...>::value };
};

// definition of Helper class
#include "helper.i"
```

```
// base case: only 1 template argument
template <typename Needle>
struct Type<Needle>
{
    enum { located = 0 };
};


#endif
```

Listing 11: helper.i

```
#include <type_traits>

template <typename Needle, typename ...Types>
template <typename NeedleIdx, typename FirstIdx, typename ...TypesIdx>
struct Type<Needle, Types...>::TypeIdx
{
    enum
    {
        value = std::is_same<NeedleIdx, FirstIdx>::value ?
                                1 : 1 + TypeIdx<NeedleIdx, TypesIdx...>::value
    };
};

template <typename Needle, typename ...Types>
template <typename NeedleIdx, typename FirstIdx>
struct Type<Needle, Types...>::TypeIdx<NeedleIdx, FirstIdx>
{
    enum { value = std::is_same<NeedleIdx, FirstIdx>::value };
};
```

0 0 1 2 2 2