C++ Exercises
Set 3

Author(s): Channa Dias Perera, Alex Swaters, Ivan Yanakiev
Previously rated by Frank
17:55

March 7, 2023

17

In this exercise we design a simple trait class for size_t values. It documents the value passed to it, whether its divisible by 2 or 3, and the number of digits in its decimal representation.

Listing 1: NrTrait/NrTrait.h

```
#ifndef SET3_NRTRAIT_H
#define SET3_NRTRAIT_H

#include <cstddef>

template <size_t val>
struct NrTrait
{
  enum
  {
    value = val,
    even = val % 2 == 0,
    by3 = val % 3 == 0,
    width = val < 10 ? 1 : 1 + NrTrait<val / 10>::width
  };
};

template<>
struct NrTrait<0>
{
  enum
  {
    value = 0,
    width = 1,
    even = true,
    by3 = false,
  };
};

#endif //SET3_NRTRAIT_H
```

Listing 2: main.ih

```
#include "NrTrait/NrTrait.h"
#include <iostream>

using namespace std;
```

Listing 3: main.cc

```
#include "main.ih"

int main()
{
```

```
                    // Showing how we use NrTrait / how we show the traits of
                    // 1871862 to stdout via a single cout statement.
  cout << NrTrait<1971962>::value << '\n'
    << NrTrait<1971962>::even << '\n'
    << NrTrait<1971962>::by3 << '\n'
    << NrTrait<1971962>::width << '\n';

}
```

18

In this exercise, we design a template class that contains the binary representation of a size_t value passed to it via its template parameter.

Listing 4: Bin/Bin.h

-8 18

```
#ifndef SET3_BIN_H
#define SET3_BIN_H

#include <cstddef>

                                        // Bin is a template class that converts
                                        // an decimal value into "binary" value,
                                        // held in the enum: binary.

                                        // value is an integer but it visually
                                        // looks like the corresponding binary num.
template <size_t val>
struct Bin
{
  enum
  {
                                        // We compute the rightmost bit of val via
                                        // val % 2. The rest of the binary would
                                        // then be in val >> 1. We then compute the
                                        // rest of the binary via
                                        // Bin<(val >> 1)>::value, which we multiply
                                        // by 10 to put leftmost rest in the right
                                        // decimal place.
    value = Bin<(val >> 1)>::value * 10 + val % 2
  };
};


                                        // Base case: val = 0, "binary" is also 0
template <>
struct Bin<0>
{
  enum
  {
    value = 0
  };
};

#endif //SET3_BIN_H
```

19

AFAICS there's no NTBS constructed

In this exercise, we create a template class that promotes a NTBS. It takes in a sequence of characters and when an object of it is created and converted to an NTBS, it returns the NTBS of the characters passed to it.

Listing 5: Chars.h

```
#ifndef SET3_CHARS_H
#define SET3_CHARS_H

#include <ostream>

                                        // Type promotion for a series of characters
```

```
template<char ...chars>
struct Chars
{};

template <char ...chars>
std::ostream &operator<<(std::ostream &out, Chars<chars...> charsObj)
{
                        // We unwrap chars, with each char having it's own
                        // insertion expression. Basically (though this is not
                        // valid syntax:
                        // out << chars[0], out << chars[1], ...
  ((out << chars), ...);
  return out;
}

#endif //SET3_CHARS_H
```

20

In this exercise, we create a Merge template class that can merge two Chars template classes (or a Chars template class and a OneChar template class) into one Chars template class (stored as nested type CP) containing all the chars in both Chars, preserving left-to-right order.

Listing 6: OneChar.h

```
#ifndef SET3_ONECHAR_H
#define SET3_ONECHAR_H

                        // Simple type promotion class from char.
template <char ch>
struct OneChar
{
  enum {value = ch };
};

#endif //SET3_ONECHAR_H
```

Listing 7: Chars.h

```
#ifndef SET3_CHARS_H
#define SET3_CHARS_H

#include <ostream>

                        // Type promotion for a series of characters
template<char ...chars>
struct Chars
{};

template <char ...chars>
std::ostream &operator<<(std::ostream &out, Chars<chars...> charsObj)
{
                        // We unwrap chars, with each char having it's own
                        // insertion expression. Basically (though this is not
                        // valid syntax:
                        // out << chars[0], out << chars[1], ...
  ((out << chars), ...);
  return out;
}

#endif //SET3_CHARS_H
```

Listing 8: Merge.h

```
#ifndef MERGE_H
#define MERGE_H

#include "Chars.h"
```

```
#include "OneChar.h"

                                // Merge is a template class that takes either:
                                // * Two Chars template classes
                                // * A Chars and OneChar template class
                                // It _merges_ these template classes by holding a
                                // type CP.
                                // CP is a Chars class containing the
                                // character template arguments in the first Chars
                                // class followed by the character template
                                // arguments in the second Chars / OneChar class.
template <typename First, typename Second>
struct Merge;

                                // Case 1: We get two Chars classes
template <char... FirstChars, char... SecondChars>
struct Merge<Chars<FirstChars...>, Chars<SecondChars...>>
{
                                  // expand the parameter packs in order
    using CP = Chars<FirstChars..., SecondChars...>;
};

                               // Case 2: We get a Chars class and then a OneChar
                               // class
template <char... FirstChars, char SecondChars>
struct Merge<Chars<FirstChars...>, OneChar<SecondChars>>
{
                                    // append OneChar character parameter to
                                    // expanded Chars variadic argument.
    using CP = Chars<FirstChars..., SecondChars>;
};

#endif
```

22

In this exercise, we convert a decimal number to a number in radix 2 <= n <= 36 via template meta programming.

We tested the program against
values: 100, 57005, 0, SIZE_MAX
radixs: 2, 8, 16, 32, 6

Listing 9: Chars/Tail.h

```
#ifndef SET3_TAIL_H
#define SET3_TAIL_H

#include "Chars.h"

                                // Template class to get the tail of a CharsT
                                // template.
                                // e.g: Tail<Char<'a','b','c'>>::tail
                                // -> Char<'b','c'>
template <typename CharsT>
struct Tail
{};

template<char first, char ...rest>
struct Tail<Chars<first, rest...>>
{
  using tail = Chars<rest...>;
};

#endif //SET3_TAIL_H
```

Listing 10: IfElse/IfElse.h

```
#ifndef SET3_IFELSE_H
#define SET3_IFELSE_H
```

```
template <bool cond, typename T1, typename T2>
struct IfElse
{};

template <typename T1, typename T2>
struct IfElse<true, T1, T2>
{
  using type = T1;
};

template <typename T1, typename T2>
struct IfElse<false, T1, T2>
{
  using type = T2;
};

#endif //SET3_IFELSE_H
```

Listing 11: Rep/Rep.h

```
#ifndef SET3_REP_H
#define SET3_REP_H

#include <cstddef>

                                // Basically checks if the remainder needs to
                                // written as a letter or whether a "normal"
                                // number would suffix.

                                // Lowest radix: 2, since this is binary
                                // Largest radix: 36, digits: 0..9, a..z
template <size_t rem, size_t radix>
struct Rep
{
  enum
  {
                                          // This is a compiler check to ensure that
                                          // the radix is within the specified
                                          // bounds. If not, we have 1 / false
                                          // which is 1 / 0 which is a compiler
                                          // error, and thus we restrict Rep's usage
    validRadix = 1 / (radix >= 2 && radix <= 36),
                                          // whether we use should alphabet char
                                          // for representation
    isAlpha = rem > 9,
                                          // rem - 10 because because 0 .. 9 will
                                          // take up 10 digits, so to represent
                                          // n, we need the n - 10th char after
                                          // 'a'.
    rep = isAlpha ? 'a' + rem - 10 : '0' + rem
  };
};

#endif //SET3_REP_H
```

Listing 12: Convert/GenerateRepRec.h

```
#ifndef SET3_GENERATEREPREC_H
#define SET3_GENERATEREPREC_H

#include "../Chars/Chars.h"
#include "../Rep/Rep.h"

#include <cstddef>

                                          // We store the representation in the rep
                                          // parameter pack
template <size_t num, size_t radix, char ...rep>
struct GenerateRepRec
{
```

```
                                        // We essentially compute the
                                        // representation of the right most digit
                                        // and store in rep, and compute the
                                        // rest via a recursive call.
  using CP = typename GenerateRepRec<
    num / radix,
    radix,
    Rep<num % radix, radix>::rep, rep...
  >::CP;
};

                                        // In the base case where there is no more
                                        // numbers to compute (which we signal
                                        // with num = 0, we store the
                                        // representation in Chars
template <size_t radix, char ...rep>
struct GenerateRepRec<0, radix, rep...>
{
  using CP = Chars<rep...>;
};

#endif //SET3_GENERATEREPREC_H
```

Listing 13: Convert/GenerateRep.h

```
#ifndef SET3_GENERATEREP_H
#define SET3_GENERATEREP_H

#include "GenerateRepRec.h"
#include "../IfElse/IfElse.h"

                                    // GenerateRep is simply a wrapper to use
                                    // GenerateRepRec
template <size_t num, size_t radix, char ...rep>
struct GenerateRep
{
                                    // GenerateRepRec cannot handle num == 0, so
                                    // we handle it directly
  using CP = typename IfElse<num == 0,
    Chars<'0'>, typename GenerateRepRec<num, radix, rep...>::CP
  >::type;
};

#endif //SET3_GENERATEREP_H
```

Listing 14: Convert/Convert.h

```
#ifndef SET3_CONVERT_H
#define SET3_CONVERT_H

#include "GenerateRep.h"

                        // We store the converted number's representation in
                        // the rep parameter pack
template <size_t num, size_t radix>
struct Convert
{
                        // We use GenerateRep as the auxiliary class to generate
                        // the representation.
  using CP = typename GenerateRep<num, radix>::CP;
};

#endif //SET3_CONVERT_H
```

23

We did not use a support class in 21, but did something similar by specializing Type. In this exercise, we nested one of our implemenations of 21 within Type privately, hiding it from the rest

of the implemenation. Then, we make a public enum with located defined by the located of the nested class.

Listing 15: Type/Type.h

```
#ifndef TYPE_INCLUDED_H
#define TYPE_INCLUDED_H

                                // We will specialize Type. It will contain
                                // TypeIdx which will contain the location to
                                // find if a type is in variadic template argument
template <typename ...Types>
class Type
{};
                                // To encode our logic in TypeIdx, we specialize
                                // Type for the cases we encounter, and then
                                // define our logic in a definition of TypeIdx
                                // (based on the specialized outer class)
                                // subsequently.
#include "generalCase.f"
#include "notFoundCase.f"
#include "foundCase.f"

#endif
```

Listing 16: Type/generalCase.f

```
                                // General case, our needle is not found at the
                                // top of the haystack
template <typename Needle, typename First, typename ...HayStack>
class Type<Needle, First, HayStack...>
{
  struct TypeIdx;

  public:
    enum
    {
      located = TypeIdx::location
    };
};

template <typename Needle, typename First, typename ...HayStack>
struct Type<Needle, First, HayStack...>::TypeIdx
{
  enum
  {
    index = Type<Needle, HayStack...>::located,
    location = index ? index + 1 : 0
  };
};
```

Listing 17: Type/foundCase.f

```
                                // Base case 2: We have found our needle
template <typename Needle, typename ...HayStack>
class Type<Needle, Needle, HayStack...>
{
  struct TypeIdx;
  public:
    enum
    {
      located = TypeIdx::location
    };
};

template <typename Needle, typename ...HayStack>
struct Type<Needle, Needle, HayStack...>::TypeIdx
{
  enum
```

```
  {
    location = 1
  };
};
```

Listing 18: Type/notFoundCase.f

```
                                    // Base case 1: We could not find our
                                    // needle
template <typename Needle>
class Type<Needle>
{
  struct TypeIdx;
  public:
    enum
    {
      located = TypeIdx::location
    };
};

template <typename Needle>
struct Type<Needle>::TypeIdx
{
  enum
  {
    location = 0
  };
};
```

Listing 19: main.cc

```
#include "Type/Type.h"
#include <iostream>

int main()
{
    std::cout
        << Type<int>::located << ' '
        << Type<int, double>::located << ' '
        << Type<int, int>::located << ' '
        << Type<int, double, int>::located << ' '
        << Type<int, double, int>::located << ' '
        << Type<int, double, int, int, int>::located
        << '\n';
}
```

24

In order to represent arbitrary numbers as character
we can just take the characters one by one and use them
in a variadic template. After all characters are taken
we could simply store all the charcaters in a static array,
which could then be printed to std::cout or passed to a string
and then print the std::string value.

Listing 20: I2C.h

```
#ifndef I2C_H
#define I2C_H

#include <cstddef>

                                    // I2CAux is our workhorse class. It keeps
                                    // the digits we will eventually convert to
                                    // chars, in the digits parameter pack.
template <size_t val, size_t ...digits>
struct I2CAux
{
                                    // We add the left most digit to the parameter
```

```
                                    // pack and continue processing
  static constexpr char const *s_ntbs =
    I2CAux<(val / 10), (val % 10), digits...>::s_ntbs;
};


                    // If we have digits and we get val == 0, we just hard code
                    // the result
template <>
struct I2CAux<0>
{
  static constexpr char const s_ntbs[] = {'0', '\0'};
};

                                    // Otherwise, if we have digits
template <size_t ...digits>
struct I2CAux<0, digits...>
{
                                      // we use parameter pack expansion to
                                      // generalize the method that is in the slides
  static constexpr char const s_ntbs[] = {('0' + digits)..., '\0'};
};


                                          // I2C is a template class which converts
                                          // a size_t to a C-style string rep.
template <size_t val>
struct I2C
{
  static constexpr char const *s_ntbs = I2CAux<val>::s_ntbs;
};

#endif
```

this is a pointer, not the ntbs string itself