C++ Exercises
Set 3

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

16:47

February 28, 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
  {
    width = 1
  };
};

#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

```
#ifndef SET3_BIN_H
#define SET3_BIN_H

#include <cstddef>

                                // The idea is to store the binary representation
                                // in the bits variadic template parameter.

                                // cont keeps track of whether we have to keep
                                // adding to bits, or if we write the actual
                                // expression out.
template <size_t val, bool cont = true, size_t ...bits>
struct Bin
{
                                    // Every step down the template instantiation
                                    // we determine the right-most bit and add it
                                    // to the front of the existing bits (since bits
                                    // would contain all bits after the current
                                    // bit)
  enum
  {
    afterShift = val >> 1
  };
  static constexpr char const *value =
    Bin<afterShift, afterShift!= 0, (val & 1ull ? 1ull : 0ull), bits...>::value;
};

template<size_t val, size_t ...bits>
struct Bin<val, false, bits...>
{
                                    // In the base case where we have evaluated
                                    // all the bits, we actually write the
                                    // value into the static constexpr.
  static constexpr char const value[] = {('0' + bits)..., '\0'};
};

#endif //SET3_BIN_H
```

19

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

template<char ...chars>
struct Chars
{
                            // contexpr is important, if we want to access the
                            // characters pass to this template at compile-time.

                            // We create s_str by unpacking chars and appending
```

```
                        // the null byte to it. The compiler is smart enough
                        // to determine the length, so we can use [] here.
  static constexpr char const s_str[] = {chars..., '\0'};

  Chars() = default;

  constexpr operator char const *() const;
};

template <char ...chars>
inline constexpr Chars<chars...>::operator char const *() const
{
  return s_str;
}

#endif //SET3_CHARS_H
```

3C

? Why

There's no need to construct the chars in s_str. It requires storage which is not needed. Make sure that the char-template arguments are inserted into an ostream.

20

0 20

Listing 6: OneChar.h

```
#ifndef SET3_ONECHAR_H
#define SET3_ONECHAR_H

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

#endif //SET3_ONECHAR_H
```

SLV!

There's no SC explaining what your template class Merge is doing. SC, especially with templates is very important to inform the reader of what & why things are implemented. Provide SC and the exercise is probably rated OK.

Listing 7: Merge.h

```
#ifndef MERGE_H
#define MERGE_H

#include "Chars.h"
#include "OneChar.h"

template <typename First, typename Second>
struct Merge;

template <char... FirstChars, char... SecondChars>
struct Merge<Chars<FirstChars...>, Chars<SecondChars...>>
{
    using CP = Chars<FirstChars..., SecondChars...>;
};

template <char... FirstChars, char SecondChars>
struct Merge<Chars<FirstChars...>, OneChar<SecondChars>>
{
    using CP = Chars<FirstChars..., SecondChars>;
};

#endif
```

21

In this exercise, we create a template class Type which contains a member 'located', which tells the 1-index'd location in the parameter argument list (starting from argument 2) of the first type in the parameter argument list.

Listing 8: Type/Type.h

```
#ifndef SET3_TYPE_H
#define SET3_TYPE_H

template <typename ...Types>
struct Type
{};

                        // General case: Needle =/= Head, keep on searching
template<typename Needle, typename Head, typename ...Rest>
struct Type<Needle, Head, Rest...>
{
  enum
  {
    index = Type<Needle, Rest...>::located,
    located = index == 0 ? 0 : index + 1
  };
};

                          // Base case 1: Cannot find needle
template <typename Needle>
struct Type<Needle>
{
  enum
  {
    located = 0
  };
};
                    // Base case 2: Found needle
template<typename Needle, typename ...Rest>
struct Type<Needle, Needle, Rest...>
{
  enum
  {
    located = 1
  };
};

#endif //SET3_TYPE_H
```

Listing 9: main.ih

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

using namespace std;
```

Listing 10: main.cc

```
#include "main.ih"

int main()
{
  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';
}
```

22

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

Listing 11: Chars/convertToNTBS.f

Listing 12: 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 13: 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 14: 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
template <size_t rem, size_t radix>
struct Rep
{
  enum
  {
    alpha = rem > 9,
    rep = alpha ? 'a' + rem - 10 : '0' + rem
  };
};

#endif //SET3_REP_H
```

how do you verify that the radix is valid? (cf. the exercise's requirement)

Listing 15: 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 16: 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 17: 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, char ...rep>
struct Convert
```

→ according to the exercise this parameter is not part of Convert

```
{
                        // We use GenerateRep as the auxiliary class to generate
                        // the representation.
  using CP = typename GenerateRep<num, radix, rep...>::CP;
};

#endif //SET3_CONVERT_H
```

Does it work?

7C

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 18: Type.h

```
#ifndef TYPE_INCLUDED_H
#define TYPE_INCLUDED_H

template <typename Needle, typename... Haystack>
class Type
{
private:
    template <typename... Types>
    struct TypeIdx
    {
    };

    template <typename Needle2>
    struct TypeIdx<Needle2>
    {
        enum
        {
            located = 0
        };
    };

    template <typename Needle2, typename... Haystack2>
    struct TypeIdx<Needle2, Needle2, Haystack2...>
    {
        enum
        {
            located = 1
        };
    };

    template <typename Needle2, typename First, typename... Haystack2>
    struct TypeIdx<Needle2, First, Haystack2...>
    {
        enum
        {
            located = Type<Needle2, Haystack2...>::located
                      ? Type<Needle2, Haystack2...>::located + 1
                      : 0
        };
    };

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

#endif
```

SF

no definition, just a declaration

This is NAF: the interface also contains the definitions of the TypeIdx elements

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 19: I2C.h

```
#ifndef I2C_H
#define I2C_H

#include <cstddef>

template <size_t N, char... Chars>
struct I2C
{
    static constexpr char const *s_ntbs =
      I2C<(N / 10), (N % 10), Chars...>::s_ntbs;
};

template <char... Chars>
struct I2C<0, Chars...>
{
    static constexpr char const s_ntbs[] = {('0' + Chars)..., '\0'};
};

template <>
struct I2C<0>
{
    static constexpr char const s_ntbs[] = {'0'};
};

#endif
```