Programming in C/C++
Exercises set two: advanced class templates

Christiaan Steenkist
Jaime Betancor Valado
Remco Bos

February 2, 2017

Exercise 9, Needle fishing

We made a function that returns the place of the first template class in a haystack of classes.

Code listings

Listing 1: main.cc

```
#include "type.h"

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';
}
```

Listing 2: type.h

```
#ifndef TYPE_H
#define TYPE_H
```

```

#include <iostream>

using namespace std;

// TYPEIDX LOOPER
// Generic case: no Needle found
template <typename Needle, size_t index,
    typename Other, typename ...Stack>
class TypeIdx : public TypeIdx<Needle,
    index + 1, Stack...>
{};

// Exit case: Needle found
template <typename Needle, size_t index,
    typename ...Stack>
class TypeIdx<Needle, index, Needle, Stack...>
{
  public:
    enum { located = index + 1 };
};

// Exit case: Needle found
template <typename Needle, size_t index>
class TypeIdx<Needle, index, Needle>
{
  public:
    enum { located = index + 1};
};

// Exit case: No more hay
template <typename Needle, size_t index,
    typename Other>
class TypeIdx<Needle, index, Other>
{
  public:
    enum { located = 0 };
};

// TYPE STARTER
```

```cpp
// Starter for TypeIdx and looping
template <typename Needle, typename ...Stack>
class Type : public TypeIdx<Needle, 0, Stack...>
{};

// Starter for empty haystack
template <typename Needle>
class Type<Needle>
{
  public:
    enum { located = 0};
};

#endif
```

Exercise 10, Needle fishing with nested class

We changed exercise 9, such that it now uses a nested helper class

Code listings

Listing 3: main.cc

```cpp
#include "type.h"

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';
}
```

Listing 4: type.h

```cpp
#ifndef TYPE_H
#define TYPE_H
```

```

#include <iostream>

using namespace std;

// Starter for the needlehunt
template <typename Needle, typename ...Stack>
class Type
{
  // Generic case: no Needle found
  template <size_t index, typename Other,
      typename ...Stacker>
  struct TypeIdx
  {
    public:
      enum {located = TypeIdx<index + 1,
            Stacker...>::located};
  };

  // Exit case: Needle found
  template <size_t index, typename ...Stacker>
  struct TypeIdx<index, Needle, Stacker...>
  {
    public:
      enum { located = index + 1 };
  };

  // Exit case: Needle found
  template <size_t index>
  struct TypeIdx<index, Needle>
  {
    public:
      enum { located = index + 1};
  };

  // Exit case: No more hay
  template <size_t index, typename Other>
  struct TypeIdx<index, Other>
  {
    public:
      enum { located = 0 };
  };

  public:
    enum { located = TypeIdx<0, Stack...>::located};
};

// Starter for empty haystack
template <typename Needle>
class Type<Needle>
{
  public:
    enum { located = 0};
};

#endif
```

Exercise 13, Binary operators

We made a class that overloads binary operators

Code listings

Listing 5: main.cc

```
#include "main.ih"

int main(int argc, char **argv)
{
  Arithmetic<int> jart;
  Arithmetic<double> dart;
  //Arithmetic<size_t> illegal;

  jart = 10;
  dart = 10;
  //illegal = 10;

  Arithmetic<double> dart2(dart);
  dart = 13;

  cout << jart.value() << '\n';
  cout << dart.value() << '\n';
  cout << dart2.value() << '\n';
}
```

Listing 6: main.ih

```
#include "arithmetic.h"
#include "adder.h"
#include <iostream>

using namespace std;
```

Listing 7: adder.add.cc

```
#include "adder.ih"

void Adder::add(Adder const &rhs)
{
  d_value += rhs.value();
}
```

Listing 8: adder.h

```
#ifndef ADDER_H
#define ADDER_H

#include <string>

#include "binopsbase.h"

class Adder: public BinopsBase<Adder, '+'>
{
  friend BinopsBase<Adder, '+'>;

  std::string d_value;

  public:
    Adder &operator=(std::string const &rhs);
    std::string const &value() const;

  private:
    void add(Adder const &rhs);
};

#endif
```

Listing 9: adder.ih

```
#include "adder.h"
```

Listing 10: adder.value.cc

```
#include "adder.ih"

std::string const &Adder::value() const
{
  return d_value;
}
```

Listing 11: arithmetic.h

```
#ifndef ARITHMETIC_H
#define ARITHMETIC_H

#include <cstring>
#include <string>

#include "binopsbase.h"

template <typename T>
class Arithmetic: public BinopsBase<Arithmetic<T>>
{};

template <>
class Arithmetic<int>:
    public BinopsBase<Arithmetic<int>>
{
  friend BinopsBase<Arithmetic<int>>;

  int d_value = 0;

  public:
    Arithmetic<int>() = default;
    Arithmetic<int>(Arithmetic<int> const &rhs)
    :
      d_value(rhs.d_value)
    {}
    Arithmetic<int>(Arithmetic<int> &&rhs)
    {
      swap(rhs);
    }

    Arithmetic<int> &operator=(int const &rhs)
    {
      d_value = rhs;
      return *this;
    }

    int const &value() const
    {
      return d_value;
    }

    void swap(Arithmetic<int> &other)
    {
      char step[sizeof(Arithmetic<int>)];
      std::memcpy(step, this,
          sizeof(Arithmetic<int>));
      std::memcpy(&other, step,
          sizeof(Arithmetic<int>));
      std::memcpy(this, &other,
          sizeof(Arithmetic<int>));
    }

  private:
    void add(Arithmetic<int> const &rhs)
    {
      d_value += rhs.value();
    }
};

template <>
class Arithmetic<double>:
    public BinopsBase<Arithmetic<double>>
```

```
{
  friend BinopsBase<Arithmetic<double>>;

  double d_value = 0;

  public:
    Arithmetic<double>() = default;
    Arithmetic<double>(Arithmetic<double> const &rhs)
    :
      d_value(rhs.d_value)
    {}
    Arithmetic<double>(Arithmetic<double> &&rhs)
    {
      swap(rhs);
    }

    Arithmetic<double> &operator=(double const &rhs)
    {
      d_value = rhs;
      return *this;
    }

    double const &value() const
    {
      return d_value;
    }

    void swap(Arithmetic<double> &other)
    {
      char step[sizeof(Arithmetic<double>)];
      std::memcpy(step, this,
          sizeof(Arithmetic<double>));
      std::memcpy(&other, step,
          sizeof(Arithmetic<double>));
      std::memcpy(this, &other,
          sizeof(Arithmetic<double>));
    }

  private:
    void add(Arithmetic<double> const &rhs)
    {
      d_value += rhs.value();
    }
};

#endif
```

Listing 12: arithmetic.ih

```
#include "arithmetic.h"
```

Listing 13: binopsbase.h

```
#ifndef BINOPSBASE_H
#define BINOPSBASE_H

template <typename Derived, int ...operators>
class BinopsBase
{
  friend Derived &operator+(Derived &lhs,
      Derived const &rhs);
  friend Derived &operator*(Derived &lhs,
      Derived const &rhs);

  void addWrap(Derived const &rhs)
  {
    Derived::add(rhs);
  }

  void mulWrap(Derived const &rhs)
  {
    Derived::mul(rhs);
  }
};

#endif
```

Listing 14: binopsbase.ih

```
#include "binopsbase.h"
```

Exercise 14, Generic variadic template

We changed the class BinopsBase to a variadic template class using a set of int argument.

Code listings

Listing 15: main.cc

```
#include "main.ih"

int main(int argc, char **argv)
{
  Arithmetic<int> jart;
  Arithmetic<double> dart;
  //Arithmetic<size_t> illegal;

  jart = 10;
  dart = 10;
  //illegal = 10;

  Arithmetic<double> dart2(dart);
  dart = 13;

  cout << jart.value() << '\n';
  cout << dart.value() << '\n';
  cout << dart2.value() << '\n';
}
```

Listing 16: main.ih

```
#include "arithmetic.h"
#include "adder.h"
#include <iostream>

using namespace std;
```

Listing 17: operations.h

```
#ifndef OPERATIONS_H
#define OPERATIONS_H

```

```
template <typename Base, typename Derived>
class Add
{};

template <typename Base, typename Derived>
class Mul
{};

#endif
```

Listing 18: operations.ih

```
#include "operations.h"
```

Listing 19: binopsbase.h

```
#ifndef BINOPSBASE_H
#define BINOPSBASE_H

#include "operations.h"

template <typename Binops,
    typename Derived, int ...operators>
class BinopsBase0
{};

template <typename Binops, typename Derived>
class BinopsBase0<Binops, Derived, 0>
{};

template <typename Derived, int ...operators>
class BinopsBase : public BinopsBase0<BinopsBase<
    Derived, operators...>, Derived, operators...>
{
  friend Derived &operator+(Derived &lhs,
      Derived const &rhs);
  friend Derived &operator*(Derived &lhs,
      Derived const &rhs);

  friend Add<BinopsBase<Derived, operators...>,
      Derived>;
  friend Mul<BinopsBase<Derived, operators...>,
      Derived>;

  void addWrap(Derived const &rhs)
  {
    Derived::add(rhs);
  }

  void mulWrap(Derived const &rhs)
  {
    Derived::mul(rhs);
  }
};

#endif
```

Listing 20: binopsbase.ih

```
#include "binopsbase.h"
```