templates applied to design and implementation

Jose Caicoya

templates

templates

templates

templates

template <typename T>

Template Specialisation

CompileTimeEvaluator

```
1  template <bool Condition>
2  struct CompileTimeEvaluator;
3
4
5  template <>
6  struct CompileTimeEvaluator<true>
7  {};
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

CompileTimeEvaluator

```
1  template <bool Condition>
2  struct CompileTimeEvaluator;
3
4
5  template <>
6  struct CompileTimeEvaluator<true>
7  {};
8
9
10
11 using integer = int;
12 templateExamples::CompileTimeEvaluator<sizeof(integer) == sizeof(int)>();
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

CompileTimeEvaluator

```
1  template <bool Condition>
2  struct CompileTimeEvaluator;
3
4
5  template <>
6  struct CompileTimeEvaluator<true>
7  {};
8
9
10
11 using integer = int;
12 templateExamples::CompileTimeEvaluator<sizeof(integer) == sizeof(int)>();
13 templateExamples::CompileTimeEvaluator<sizeof(integer) == sizeof(char)>();
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

CompileTimeEvaluator

```
1  template <bool Condition>
2  struct CompileTimeEvaluator;
3
4
5  template <>
6  struct CompileTimeEvaluator<true>
7  {};
8
9
10
11 using integer = int;
12 templateExamples::CompileTimeEvaluator<sizeof(integer) == sizeof(int)>();
13 templateExamples::CompileTimeEvaluator<sizeof(integer) == sizeof(char)>();
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

Implicit instantiate of undefined template ‘CompileTimeEvaluator<false>’

foo specialisation

```
1  template <typename DataBaseValueType,
2            typename ParameterType>
3  bool foo(DataBase<DataBaseValueType> &dataBase,
4           ParameterType parameter)
5  {
6      DataBaseValueType dataToInsert(parameter);
7      if(true == dataBase.contains(dataToInsert)) { return true; }
8      std::size_t size = dataBase.size();
9      dataBase.insert(dataToInsert);
10     return dataBase.size() == size + 1;
11
12 }
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

foo specialisation

```
1  template <typename DataBaseValueType,
2            typename ParameterType>
3  bool foo(DataBase<DataBaseValueType> &dataBase,
4           ParameterType parameter)
5  {
6      DataBaseValueType dataToInsert(parameter);
7      if(true == dataBase.contains(dataToInsert)) { return true; }
8      std::size_t size = dataBase.size();
9      dataBase.insert(dataToInsert);
10     return dataBase.size() == size + 1;
11
12 }
13
14 template <typename ParameterType>
15 void foo(DataBase<std::string> &dataBase,
16          ParameterType parameter)
17 {
18     std::stringstream ss;
19     ss << parameter;
20     std::string data = ss.str();
21     dataBase.insert(data);
22 }
23
24
25
26
27
28
29
30
31
32
```

foo specialisation

```
1  template <typename DataBaseValueType,
2            typename ParameterType>
3  bool foo(DataBase<DataBaseValueType> &dataBase,
4           ParameterType parameter)
5  {
6      DataBaseValueType dataToInsert(parameter);
7      if(true == dataBase.contains(dataToInsert)) { return true; }
8      std::size_t size = dataBase.size();
9      dataBase.insert(dataToInsert);
10     return dataBase.size() == size + 1;
11
12 }
13
14 template <typename ParameterType>
15 void foo(DataBase<std::string> &dataBase,
16          ParameterType parameter)
17 {
18     std::stringstream ss;
19     ss << parameter;
20     std::string data = ss.str();
21     dataBase.insert(data);
22 }
23
24 template <>
25 bool foo(DataBase<std::string> &dataBase,
26          std::string data)
27 {
28     dataBase.insert(data);
29 }
30
31
32
```

Variadic Templates

parametersToString

```
1  template <typename ParameterType>
2  std::string parametersToString(ParameterType &&lastParameter)
3  {
4      std::stringstream ss;
5      ss << lastParameter;
6      return ss.str();
7  }
8
9
10 template <typename ParameterType,
11           typename... ArgumentTypes>
12 std::string parametersToString(ParameterType &&currentParameter,
13                                ArgumentTypes&&... arguments)
14 {
15     std::stringstream ss;
16     ss << currentParameter << ' ' <<
17     parametersToString(std::forward<ArgumentTypes>(arguments)...);
18     return ss.str();
19 }
20
21
22
23
24
25
26
27
28
29
30
31
32
```

parametersToStringReverse

```
1  template <typename ParameterType>
2  std::string parametersToStringReverse(ParameterType &&firstParameter)
3  {
4      std::stringstream ss;
5      ss << firstParameter;
6      return ss.str();
7  }
8
9
10 template <typename ParameterType,
11           typename... ArgumentTypes>
12 std::string parametersToStringReverse(ParameterType &&currentParameter,
13                                       ArgumentTypes... arguments)
14 {
15     std::stringstream ss;
16     ss <<  parametersToStringReverse(std::forward<ArgumentTypes>(arguments)…)
17        << ' ' << currentParameter;
18     return ss.str();
19 }
20
21
22
23
24
25
26
27
28
29
30
31
32
```

The Palindrome Problem

The Palindrome Problem

```
template <typename Fisrt,
          typename... ArgumentTypes,
          typename Last>
bool isPalindrome(First first,
                  ArgementTypes... &&arguments,
                  Last last)
{
    // TODO
}
```

The Palindrome Problem

```
template <typename Fisrt,
          typename... ArgumentTypes,
          typename Last>
bool isPalindrome(First first,
                  ArgementTypes... &&arguments,
                  Last last)
{
    // Nobody can't call me!!
}
```

The Palindrome Problem

```
template <typename Fisrt,
          typename... ArgumentTypes,
          typename Last>
bool isPalindrome(First first,
                  ArgementTypes... &&arguments,
                  Last last)
{
    // Nobody can't call me!!
}
```

std::tuple

what we want

```
printIsPalindrome(1, 'A', 15.1, 'A', 1);
printIsPalindrome(1, 'A', 15.1, 'B', 1);
```

what we want

```
printIsPalindrome(1, 'A', 15.1, 'A', 1);
printIsPalindrome(1, 'A', 15.1, 'B', 1);

printIsPalindrome(1);
printIsPalindrome();
```

what we want

```
printIsPalindrome(1, 'A', 15.1, 'A', 1);
printIsPalindrome(1, 'A', 15.1, 'B', 1);

printIsPalindrome(1);
printIsPalindrome();

printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
printIsPalindrome(1, 'A', 15.1, 16.1, 'A', 1);
```

what we want

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8  printIsPalindrome(1, 'A', 15.1, 16.1, 'A', 1);
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     std::tuple<Args...> argsTuple(std::forward<Args>(args)...);
16     std::cout << argsTuple << " is";
17     if(false == isPalindrome(argsTuple))
18     {
19         std::cout << " not";
20     }
21     std::cout << " a palindrome"<< std::endl;
22 }
23
24
25
26
27
28
29
30
31
32
```

what we want

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8  printIsPalindrome(1, 'A', 15.1, 16.1, 'A', 1);
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     //std::tuple<Args...> argsTuple(std::forward<Args>(args)...);
16     auto argsTuple = std::forward_as_tuple(args...);
17     std::cout << argsTuple << " is";
18     if(false == isPalindrome(argsTuple))
19     {
20         std::cout << " not";
21     }
22     std::cout << " a palindrome"<< std::endl;
23 }
24
25
26
27
28
29
30
31
32
```

what we want

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8  printIsPalindrome(1, 'A', 15.1, 16.1, 'A', 1);
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     auto argsTuple = std::forward_as_tuple(args...);
16     std::cout << argsTuple << " is";
17     if(false == isPalindrome(argsTuple))
18     {
19         std::cout << " not";
20     }
21     std::cout << " a palindrome"<< std::endl;
22 }
23
24
25
26
27
28
29
30
31
32
```

tuple printer

```
1  template<typename... Tuple>
2  std::ostream& operator<<(std::ostream &out, const std::tuple<Tuple...> &tuple) {
3      out << "<";
4      Printer<std::tuple<Tuple...>, 0, sizeof...(Tuple) - 1>::print(out, tuple);
5      out << ">";
6      return out;
7  }
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

tuple printer

```
1  template<typename... Tuple>
2  std::ostream& operator<<(std::ostream &out, const std::tuple<Tuple...> &tuple) {
3      out << "<";
4      Printer<std::tuple<Tuple...>, 0, sizeof...(Tuple) - 1>::print(out, tuple);
5      out << ">";
6      return out;
7  }
8
9
10 template<typename Tuple, int N, int Last>
11 struct Printer
12 {
13     static void print(std::ostream& out, const Tuple &tuple) {
14         out << std::get<N>(value) << ", ";
15         Printer<Tuple, N + 1, Last>::print(out, tuple);
16     }
17 };
18
19
20 template<typename Tuple, int N>
21 struct Printer<Tuple, N, N>
22 {
23     static void print(std::ostream& out, const Tuple &tuple) {
24         out << std::get<N>(tuple);
25     }
26 };
27
28
29
30
31
32
```

what we want

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8  printIsPalindrome(1, 'A', 15.1, 16.1, 'A', 1);
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     auto argsTuple = std::forward_as_tuple(args...);
16     std::cout << argsTuple << " is";
17     if(false == isPalindrome(argsTuple))
18     {
19         std::cout << " not";
20     }
21     std::cout << " a palindrome"<< std::endl;
22 }
23
24
25
26
27
28
29
30
31
32
```

isPalindrome

```
1  template<typename... Tuple>
2  bool isPalindrome(const std::tuple<Tuple...> &tuple)
3  {
4      return Wrapper<std::tuple<Tuple...>, 0, sizeof...(Tuple) - 1>::palindrome(tuple);
5  }
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

isPalindrome

```
1  template<typename... Tuple>
2  bool isPalindrome(const std::tuple<Tuple...> &tuple)
3  {
4      return Wrapper<std::tuple<Tuple...>, 0, sizeof...(Tuple) - 1>::palindrome(tuple);
5  }
6
7
8  template<typename Tuple, unsigned Left, unsigned Right>
9  struct Wrapper
10 {
11     static bool palindrome(const Tuple &tuple) {
12         if(std::get<Left>(value) != std::get<Right>(tuple))
13         {
14             return false;
15         }
16         return Wrapper<Tuple, Left+1, Right-1>::palindrome(tuple);
17     }
18 };
19
20
21 template<typename Tuple, unsigned Middle>
22 struct Wrapper<Tuple, Middle, Middle>
23 {
24     static bool palindrome(const Tuple &tuple) {
25         return true;
26     }
27 };
28
29
30
31
32
```

what we want

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8  printIsPalindrome(1, 'A', 15.1, 16.1, 'A', 1);
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     auto argsTuple = std::forward_as_tuple(args...);
16     std::cout << argsTuple << " is";
17     if(false == isPalindrome(argsTuple))
18     {
19         std::cout << " not";
20     }
21     std::cout << " a palindrome"<< std::endl;
22 }
23
24
25
26
27
28
29
30
31
32
```

what we want

<1, A, 15.1, A, 1> is a palindrome

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8  printIsPalindrome(1, 'A', 15.1, 16.1, 'A', 1);
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     auto argsTuple = std::forward_as_tuple(args...);
16     std::cout << argsTuple << " is";
17     if(false == isPalindrome(argsTuple))
18     {
19         std::cout << " not";
20     }
21     std::cout << " a palindrome"<< std::endl;
22 }
23
24
25
26
27
28
29
30
31
32
```

what we want

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8  printIsPalindrome(1, 'A', 15.1, 16.1, 'A', 1);
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     auto argsTuple = std::forward_as_tuple(args...);
16     std::cout << argsTuple << " is";
17     if(false == isPalindrome(argsTuple))
18     {
19         std::cout << " not";
20     }
21     std::cout << " a palindrome"<< std::endl;
22 }
23
24
25
26
27
28
29
30
31
32
```

<1, A, 15.1, A, 1> is a palindrome

<1, A, 15.1, B, 1> is not a palindrome

what we want

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8  printIsPalindrome(1, 'A', 15.1, 16.1, 'A', 1);
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     auto argsTuple = std::forward_as_tuple(args...);
16     std::cout << argsTuple << " is";
17     if(false == isPalindrome(argsTuple))
18     {
19         std::cout << " not";
20     }
21     std::cout << " a palindrome"<< std::endl;
22 }
23
24
25
26
27
28
29
30
31
32
```

<1, A, 15.1, A, 1> is a palindrome

<1, A, 15.1, B, 1> is not a palindrome

<1> is a palindrome

what we want

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     auto argsTuple = std::forward_as_tuple(args...);
16     std::cout << argsTuple << " is";
17     if(false == isPalindrome(argsTuple))
18     {
19         std::cout << " not";
20     }
21     std::cout << " a palindrome"<< std::endl;
22 }
23
24
25
26
27
28
29
30
31
32
```

<1, A, 15.1, A, 1> is a palindrome

<1, A, 15.1, B, 1> is not a palindrome

<1> is a palindrome

Non-type template argument evaluates to 18446744073709551615,
which cannot be narrowed to type ‘unsigned int’

tuple printer

```
1  template<typename... Tuple>
2  std::ostream& operator<<(std::ostream &out, const std::tuple<Tuple...> &tuple) {
3      out << "<";
4      Printer<std::tuple<Tuple...>, 0, sizeof...(Tuple) - 1>::print(out, tuple);
5      out << ">";
6      return out;
7  }
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

Non-type template argument evaluates to 18446744073709551615,
which cannot be narrowed to type ‘unsigned int’

tuple printer

```
1  template<typename... Tuple>
2  std::ostream& operator<<(std::ostream &out, const std::tuple<Tuple...> &tuple) {
3      out << "<";
4      if(sizeof...(Tuple) > 0)
5      {
6          Printer<std::tuple<Tuple...>, 0, sizeof...(Tuple) - 1>::print(out, tuple);
7      }
8      out << ">";
9      return out;
10 }
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

tuple printer

```
1  template<typename... Tuple>
2  std::ostream& operator<<(std::ostream &out, const std::tuple<Tuple...> &tuple) {
3      out << "<";
4      if(sizeof...(Tuple) > 0)
5      {
6          Printer<std::tuple<Tuple...>, 0, sizeof...(Tuple) - 1>::print(out, tuple);
7      }
8      out << ">";
9      r
10 }
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

Non-type template argument evaluates to 18446744073709551615,
which cannot be narrowed to type ‘unsigned int’

printPalindrome empty

```
1  template <bool IsZero>
2  struct Checker
3  {
4      template <typename... Tuple>
5      static bool isPalindrome(const std::tuple<Tuple...> &tuple);
6
7      template <typename... Tuple>
8      static void printTuple(std::ostream &out, const std::tuple<Tuple...> &tuple);
9  };
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

Checker true

```
1  template <bool IsZero>
2  struct Checker
3  {
4      template <typename... Tuple>
5      static bool isPalindrome(const std::tuple<Tuple...> &tuple);
6
7      template <typename... Tuple>
8      static void printTuple(std::ostream &out, const std::tuple<Tuple...> &tuple);
9  };
10
11
12 template <>
13 struct Checker<true>
14 {
15     template <typename... Tuple>
16     static bool isPalindrome(const std::tuple<Tuple...> &tuple)
17     {
18         return true;
19     }
20
21
22     template <typename... Tuple>
23     static void printTuple(std::ostream &out, const std::tuple<Tuple...> &tuple)
24     {
25        //Do nothing
26     }
27 };
28
29
30
31
32
```

Checker false

```
1  template <bool IsZero>
2  struct Checker
3  {
4      template <typename... Tuple>
5      static bool isPalindrome(const std::tuple<Tuple...> &tuple);
6
7      template <typename... Tuple>
8      static void printTuple(std::ostream &out, const std::tuple<Tuple...> &tuple);
9  };
10
11
12 template <>
13 struct Checker<false>
14 {
15     template <typename... Tuple>
16     static bool isPalindrome(const std::tuple<Tuple...> &tuple)
17     {
18         return Wrapper<std::tuple<Tuple...>, 0,
19                        sizeof...(Tuple) - 1>::palindrome(tuple);
20     }
21
22     template <typename... Tuple>
23     static void printTuple(std::ostream &out, const std::tuple<Tuple...> &tuple)
24     {
25         Printer<std::tuple<Tuple...>, 0, sizeof...(Tuple) - 1>::print(out, tuple);
26     }
27 };
28
29
30
31
32
```

calling Checker

```
1  template <bool IsZero>
2  struct Checker
3  {
4      template <typename... Tuple>
5      static bool isPalindrome(const std::tuple<Tuple...> &tuple);
6
7      template <typename... Tuple>
8      static void printTuple(std::ostream &out, const std::tuple<Tuple...> &tuple);
9  };
10
11
12 template<typename... Tuple>
13 std::ostream& operator<<(std::ostream& out, const std::tuple<Tuple...> &tuple) {
14     out << "<";
15     const bool IsZero = (sizeof...(Tuple) == 0);
16     Checker<IsZero>::printTuple(out, tuple);
17     out << ">";
18     return out;
19 }
20
21
22 template<typename... Tuple>
23 bool isPalindrome(const std::tuple<Tuple...> &tuple)
24 {
25     const bool IsZero = (sizeof...(Tuple) == 0);
26     return Checker<IsZero>::isPalindrome(tuple);
27 }
28
29
30
31
32
```

what we want

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     auto argsTuple = std::forward_as_tuple(args...);
16     std::cout << argsTuple << " is";
17     if(false == isPalindrome(argsTuple))
18     {
19         std::cout << " not";
20     }
21     std::cout << " a palindrome"<< std::endl;
22 }
23
24
25
26
27
28
29
30
31
32
```

<1, A, 15.1, A, 1> is a palindrome

<1, A, 15.1, B, 1> is not a palindrome

<1> is a palindrome

Non-type template argument evaluates to 18446744073709551615,
which cannot be narrowed to type ‘unsigned int’

what we want

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8  printIsPalindrome(1, 'A', 15.1, 16.1, 'A', 1);
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     auto argsTuple = std::forward_as_tuple(args...);
16     std::cout << argsTuple << " is";
17     if(false == isPalindrome(argsTuple))
18     {
19         std::cout << " not";
20     }
21     std::cout << " a palindrome"<< std::endl;
22 }
23
24
25
26
27
28
29
30
31
32
```

<1, A, 15.1, A, 1> is a palindrome

<1, A, 15.1, B, 1> is not a palindrome

<1> is a palindrome

<> is a palindrome

what we want

```
1  printIsPalindrome(1, 'A', 15.1, 'A', 1);
2  printIsPalindrome(1, 'A', 15.1, 'B', 1);
3
4  printIsPalindrome(1);
5  printIsPalindrome();
6
7  printIsPalindrome(1, 'A', 15.1, 15.1, 'A', 1);
8  printIsPalindrome(1, 'A', 15.1, 16.1, 'A', 1);
9
10
11
12 template <typename ...Args>
13 void printIsPalindrome(Args&& ...args)
14 {
15     auto argsTuple = std::forward_as_tuple(args...);
16     std::cout << argsTuple << " is";
17     if(false == isPalindrome(argsTuple))
18     {
19         std::cout << " not";
20     }
21     std::cout << " a palindrome"<< std::endl;
22 }
23
24
25
26
27
28
29
30
31
32
```

<1, A, 15.1, A, 1> is a palindrome

<1, A, 15.1, B, 1> is not a palindrome

<1> is a palindrome

<> is a palindrome

Static_assert failed “tuple_element index out of range”

Factory with Register

ActionInterface

```
1  class ActionInterface
2  {
3  public:
4      virtual void execute(DataType data) = 0;
5  };
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

CreateAction

```
1  class ActionInterface
2  {
3  public:
4      virtual void execute(DataType data) = 0;
5  };
6
7
8  class CreateAction : public ActionInterface
9  {
10 public:
11     CreateAction(StringDataBase &dataBase) : _dataBase(dataBase) {}
12
13     void execute(DataType data) { _dataBase.insert(data); }
14
15 private:
16     StringDataBase &_dataBase;
17 };
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

EraseAction

```
1  class ActionInterface
2  {
3  public:
4      virtual void execute(DataType data) = 0;
5  };
6
7
8  class EraseAction : public ActionInterface
9  {
10 public:
11     EraseAction(StringDataBase &dataBase) : _dataBase(dataBase) {}
12
13     void execute(DataType data) { _dataBase.erase(data); }
14
15 private:
16     StringDataBase &_dataBase;
17 };
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

UnknownAction

```
1  class ActionInterface
2  {
3  public:
4      virtual void execute(DataType data) = 0;
5  };
6
7
8  class UnknownAction : public ActionInterface
9  {
10 public:
11     UnknownAction() {}
12
13     void execute(DataType /*data*/) { std::cout << "operation unknown" << std::endl; }
14 };
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

ActionFactory

```
1  enum class ActionType { Creation, Erase, Unknown };
2
3
4  class ActionFactory
5  {
6  public:
7
8      ActionFactory(StringDataBase &dataBase) : _dataBase(dataBase) {}
9
10     std::shared_ptr<ActionInterface> build(ActionType actionType)
11     {
12         std::shared_ptr<ActionInterface> action;
13         switch(actionType)
14         {
15             case ActionType::Creation:
16                 action = std::make_shared<CreateAction>(_dataBase);
17                 break;
18             case ActionType::Erase:
19                 action = std::make_shared<EraseAction>(_dataBase);
20                 break;
21             default:
22                 action = std::make_shared<UnknownAction>();
23         }
24         return action;
25     }
26
27 private:
28     StringDataBase &_dataBase;
29 };
30
31
32
```

FactoryWithRegister

```
1  template <typename KeyType,
2            typename ResultType,
3            typename ...Args>
4  class FactoryWithRegister
5  {
6  public:
7      void registerFoo(KeyType key,
8                       std::shared_ptr<ResultType> (*foo)(Args ...args))
9      {
10         _data[key] = foo;
11     }
12
13     std::shared_ptr<ResultType> createObject(KeyType key,
14                                              Args ...args)
15     {
16         return _data[key](args...);
17     }
18
19 private:
20     std::map<KeyType,
21              std::shared_ptr<ResultType> (*)(Args ...)> _data;
22 };
23
24
25
26
27
28
29
30
31
32
```

FactoryWithRegister

```
1  using Factory = FactoryWithRegister<ActionType,
2                                      ActionInterface,
3                                      StringDataBase &>;
4
5  std::shared_ptr<ActionInterface> fooCreateAction(StringDataBase &dataBase)
6  { return std::shared_ptr<ActionInterface>(std::make_shared<CreateAction>(dataBase)); }
7
8  std::shared_ptr<ActionInterface> fooEraseAction(StringDataBase &dataBase)
9  { return std::shared_ptr<ActionInterface>(std::make_shared<EraseAction>(dataBase)); }
10
11 std::shared_ptr<ActionInterface> fooUnknownAction(StringDataBase &/*dataBase*/)
12 { return std::shared_ptr<ActionInterface>(std::make_shared<UnknownAction>()); }
13
14
15 Factory actionFactory;
16
17 actionFactory.registerFoo(ActionType::Creation,
18                           fooCreateAction);
19
20 actionFactory.registerFoo(ActionType::Erase,
21                           fooEraseAction);
22
23 actionFactory.registerFoo(ActionType::Unknown,
24                           fooUnknownAction);
25
26
27
28
29
30
31
32
```

FactoryWithRegister

```
1
2  Factory actionFactory;
3
4  actionFactory.registerFoo(ActionType::Creation,
5                            fooCreateAction);
6
7  actionFactory.registerFoo(ActionType::Erase,
8                            fooEraseAction);
9
10 actionFactory.registerFoo(ActionType::Unknown,
11                           fooUnknownAction);
12
13 ------------------------------------------------------------------------
14
15
16 auto inputData = optionaInputData.get();
17 auto actionType = std::get<1>(inputData);
18 auto data = std::get<2>(inputData);
19
20
21 auto actionPtr = actionFactory.createObject(actionType,
22                                             dataBase);
23 actionPtr->execute(data);
24
25
26
27
28
29
30
31
32
```

Muchas gracias

https://github.com/jcaicoya/UsingStd16

jcaicoya@hotelbeds.com

caicoya@gmail.com