

Deadline: 31.03.2017, 13:50

☐ Gr. 1, J. Karder, MSc.☐ Gr. 2, P. Fleck, MSc.Name Thomas Herzog Effort in h ?Points 21 Lecturer _____**1. ... like Bunnies****(2 + 2 Points)**

Since your first programming exercises (in the long gone days of your bachelor studies) you know the famous Fibonacci sequence which is defined for all positive integers n :

$$fib(0) = 0$$

$$fib(1) = 1$$

$$fib(n) = fib(n-1) + fib(n-2)$$

Perhaps you also remember that the Fibonacci sequence is not only a well-known example of recursive functions, but is also of unbelievable practical use: Using the Fibonacci sequence you can compute the size of an (ideal) bunny population at time t .

Due to this immense importance of the Fibonacci sequence, we would like to focus on it in this first task. Anyhow, the computation of the sequence should now not be done at runtime but during compilation.

- Write a C++ template metafunction *Fibonacci* which computes the n -th Fibonacci number at compile time. Demonstrate how the function can be used by computing *Fibonacci*<40>. For comparison, also implement a recursive function *fibonacci* which computes the Fibonacci numbers in the conventional way at runtime.
- Extend your program so that it lists the first 40 Fibonacci numbers. Thereby the numbers should be computed on the one hand at compile time using template metafunctions (e.g., *DO* or *WHILE*) and on the other hand conventionally at runtime.

2. 1, 2, 3, 4, 5, 6, 7, ...**(6 + 6 Points)**

- Implement a class *Counter* which provides the methods *reset*, *increment* and *value* to access and manipulate its counter value. The data type and the initial value of the counter value should be configurable. Use a *traits class* to pass all required configuration information to the *Counter* (data type and initial value).
- Furthermore, two additional optional configurations should be available for counters. On the one hand it should be possible to define an upper limit for the counter value, which must not be exceeded (*BoundedCounter*). On the other hand the step size of the counter should be configurable (*VarIncrementCounter*). Use the concept of *inheritance-based wrapper classes* to realize specializations of *Counter* for these two configurations.

Test different configurations (e.g., *IntCounter*, *DoubleCounter*, *BoundedIntCounter*, *VarIncrementIntCounter*, *BoundedVarIncrementIntCounter*, ...)

3. Counters off the Shelf

(8 Points)

Similarly to the *LIST_GENERATOR* example which has been discussed in the lecture, implement your own generator *COUNTER_GENERATOR* to create suitable configurations for your *Counter* class automatically.

Thoroughly think about, how you can pass all the different values to the generator which are required in the configuration (initial value, upper bound, step size).

Test your generator extensively by creating and using different counter variants.

Übung 3

1 ... like Bunnies

1.1 Lösungsidee

Die Implementierungen der Aufgabe *like Bunnies* wird in drei Dateien aufgeteilt.

- fibonacci.hpp
- statement.hpp
- main.cpp

full specialization

Die Berechnung der Fibonacci-Folge wird über eine rekursive Funktion implementiert, die dazu verwendet wird, die Berechnung zur Laufzeit ~~zu~~ durchzuführen. Es wird ein Template implementiert, welches die Fibonacci-Folge beim Kompilieren berechnet. Es werden zwei partielle Ausprägungen des Templates implementiert und zwar für die Werte 0 und 1. 0 damit die Instanziierung der Templates ein definiertes Ende hat und zusätzlich eine partielle Ausprägung für 1, da die Fibonacci-Folge wie folgt definiert ist $fib(n) = fib(n - 1) + fib(n - 2)$.

Für die Berechnung der Fibonacci-Folge über *DO-IF-Template-Metafunctions*, die beim Kompilieren evaluiert werden, wird ein *Template* für die Berechnung der Fibonacci-Folge implementiert, sowie ein *Template FibonacciCondition* für die Bedingung, die entscheidet, wann die Berechnung fertig ist. Um das aktuelle *n* im *Template* zu speichern, wird folgende Anweisung verwendet `enum { current = n }`. Das gespeicherte *n* als *current* wird von der *FibonacciCondition* benötigt, um zu entscheiden, wann die Berechnung fertig ist.

evtl. ~ fuer non-breaking space in latex verwenden ;)

Die *DO-IF-Statement Templates* werden in der Datei *statement.hpp* implementiert. Für das *IF-Template* wird zusätzlich ein partielles *Template* für den boolschen Wert *false* implementiert. Das *DO-Template* erwartet sich zwei Typ-Parameter *Statement* und *Condition*, wobei *Statement* eine einfach verkettete Liste von *Statements* darstellt, wobei das nächste *Statement* über *Statement:NEXT* verfügbar ist.

Listing 1: fibonacci.hpp

```
1  #ifndef _fibonacci_h
2  #define _fibonacci_h
3
4  // function which calculates fibonacci number
5  int fibonacci(int n) {
6      if (n <= 1) {
7          return n;
8      }
9      else {
10         return fibonacci(n - 1) + fibonacci(n - 2);
11     }
12 }
13
14 // partial fibonacci template
15 template<int n>
16 struct Fibonacci {
17     enum { RET = Fibonacci<n - 1>::RET + Fibonacci<n - 2>::RET };
18 };
19
20 // full fibonacci template for 0
21 template<>
22 struct Fibonacci<0> {
23     enum { RET = 0 };
24 };
```

Übung 3

```

25
26 // full fibonacci template for 1
27 template<>
28 struct Fibonacci<1> {
29     enum { RET = 1 };
30 };
31
32 // partial fibonacci condition template
33 template<int n>
34 struct FibonacciStatement {
35     // here we remember the current set template value, otherwise will be lost
36     enum { current = n };
37     static void exec() {
38         cout << "fibonacci<" << n << ">::RET = " << Fibonacci<n>::RET << endl;
39     }
40     typedef FibonacciStatement<n + 1> Next;
41 };
42
43 // partial fibonacci end condition statement template
44 template<int max>
45 struct FibonacciEndCondition {
46     template<typename Statement>
47     struct Code {
48         enum { RET = Statement::current <= max };
49     };
50 };
51
52 #endif

```

Listing 2: statement.hpp

```

1  #ifndef _statement_
2  #define _statement_
3
4  struct Stop {
5      static void exec() {}
6  };
7
8  // If template
9  template<bool Condition, typename Then, typename Else>
10 struct IF {
11     typedef Then RET;
12 };
13
14 // Then template
15 template<typename Then, typename Else>
16 struct IF<false, Then, Else> {
17     typedef Else RET;
18 };
19
20 // Do statement
21 template<typename Statement, typename Condition>
22 struct DO {
23     typedef typename Statement::Next NextStatement;
24     static void exec() {
25         Statement::exec();
26
27         IF<Condition::Code<NextStatement>::RET,
28             DO<NextStatement, Condition>,
29             Stop
30         >::RET::exec();
31     }

```

Übung 3

```
32 };
33 #endif
```

Listing 3: main.cpp

```
1  #include <iostream>
2  #include <chrono>
3  #include "fibonacci.hpp"
4  #include "statement.hpp"
5
6  using namespace std;
7
8  // templated function for measuring
9  template<typename Func>
10 void measure(Func func) {
11     auto start = chrono::high_resolution_clock::now();
12     func();
13     auto end = chrono::high_resolution_clock::now();
14
15     auto duration = chrono::duration_cast<chrono::duration<double>>(end - start);
16     cout << "duration: " << fixed << duration.count() << endl <<
        ↳ "-----" << endl;
17 }
18
19 int main() {
20     static const int n = 40;
21
22     // measure recursive call at runtime
23     measure([&]() { cout << "Recursive call:" << endl << "fibonacci(" << n << ") = " << fibonacci(n)
        ↳ << endl; });
24     // measure compile time calculation
25     measure([&]() { cout << "Compile time call:" << "fibonacci<" << n << ">::RET = " <<
        ↳ Fibonacci<n>::RET << endl; } });
26
27     // iterative recursive calls
28     measure([&]() {
29         for (int i = 0; i < n; i++) {
30             measure([&]() {cout << "Iterative call: " << endl << "fibonacci(" << i << ") = " <<
                ↳ fibonacci(i) << endl; });
31         }
32     });
33
34     // measure compile time call with Do template
35     cout << "Compile time call with Do - If statement: " << endl;
36     measure([&]() {
37         DO<FibonacciStatement<10>, FibonacciEndCondition<40>>::exec();
38     });
39
40     return 0;
41 }
```

Übung 3

1.2 Tests

```

Recursive call:
fibonacci(40) = 102334155
duration: 8.128902
-----
Compile time call: fibonacci<40>::RET = 102334155
duration: 0.001241
-----
Iterative call:
fibonacci(0) = 0
duration: 0.000714
-----
Iterative call:
fibonacci(1) = 1
duration: 0.000589
-----
Iterative call:
fibonacci(2) = 1
duration: 0.000497
-----
Iterative call:
fibonacci(3) = 2
duration: 0.000497
-----
Iterative call:
fibonacci(4) = 3
duration: 0.000497
-----
Iterative call:
fibonacci(5) = 5
duration: 0.000530
-----
Iterative call:
fibonacci(6) = 8
duration: 0.001458
-----
Iterative call:
fibonacci(7) = 13
duration: 0.001091
-----
Iterative call:
fibonacci(8) = 21
duration: 0.002063
-----
Iterative call:
fibonacci(9) = 34
duration: 0.001568
-----
Iterative call:
fibonacci(10) = 55
duration: 0.001613
-----
Iterative call:
fibonacci(11) = 89
duration: 0.001714
-----
Iterative call:
fibonacci(12) = 144
duration: 0.001108
-----

```

Abbildung 1: Test Teil 1

Übung 3

```

-----
Iterative call:
fibonacci(13) = 233
duration: 0.001514
-----
Iterative call:
fibonacci(14) = 377
duration: 0.002782
-----
Iterative call:
fibonacci(15) = 610
duration: 0.000934
-----
Iterative call:
fibonacci(16) = 987
duration: 0.001254
-----
Iterative call:
fibonacci(17) = 1597
duration: 0.001154
-----
Iterative call:
fibonacci(18) = 2584
duration: 0.001057
-----
Iterative call:
fibonacci(19) = 4181
duration: 0.001435
-----
Iterative call:
fibonacci(20) = 6765
duration: 0.002333
-----
Iterative call:
fibonacci(21) = 10946
duration: 0.001917
-----
Iterative call:
fibonacci(22) = 17711
duration: 0.002126
-----
Iterative call:
fibonacci(23) = 28657
duration: 0.003417
-----
Iterative call:
fibonacci(24) = 46368
duration: 0.005488
-----
Iterative call:
fibonacci(25) = 75025
duration: 0.007235
-----
Iterative call:
fibonacci(26) = 121393
duration: 0.011695
-----
Iterative call:
fibonacci(27) = 196418
duration: 0.017702
-----

```

Abbildung 2: Test Teil 2

Übung 3

```

-----
Iterative call:
fibonacci(27) = 196418
duration: 0.017702
-----
Iterative call:
fibonacci(28) = 317811
duration: 0.026852
-----
Iterative call:
fibonacci(29) = 514229
duration: 0.041560
-----
Iterative call:
fibonacci(30) = 832040
duration: 0.066955
-----
Iterative call:
fibonacci(31) = 1346269
duration: 0.104698
-----
Iterative call:
fibonacci(32) = 2178309
duration: 0.167805
-----
Iterative call:
fibonacci(33) = 3524578
duration: 0.274100
-----
Iterative call:
fibonacci(34) = 5702887
duration: 0.436546
-----
Iterative call:
fibonacci(35) = 9227465
duration: 0.713934
-----
Iterative call:
fibonacci(36) = 14930352
duration: 1.189259
-----
Iterative call:
fibonacci(37) = 24157817
duration: 1.894353
-----
Iterative call:
fibonacci(38) = 39088169
duration: 3.030520
-----
Iterative call:
fibonacci(39) = 63245986
duration: 4.836759
-----
duration: 12.925221
-----

```

Abbildung 3: Test Teil 3

Übung 3

```
Compile time call with Do - If statement:  
fibonacci<10>::RET = 55  
fibonacci<11>::RET = 89  
fibonacci<12>::RET = 144  
fibonacci<13>::RET = 233  
fibonacci<14>::RET = 377  
fibonacci<15>::RET = 610  
fibonacci<16>::RET = 987  
fibonacci<17>::RET = 1597  
fibonacci<18>::RET = 2584  
fibonacci<19>::RET = 4181  
fibonacci<20>::RET = 6765  
fibonacci<21>::RET = 10946  
fibonacci<22>::RET = 17711  
fibonacci<23>::RET = 28657  
fibonacci<24>::RET = 46368  
fibonacci<25>::RET = 75025  
fibonacci<26>::RET = 121393  
fibonacci<27>::RET = 196418  
fibonacci<28>::RET = 317811  
fibonacci<29>::RET = 514229  
fibonacci<30>::RET = 832040  
fibonacci<31>::RET = 1346269  
fibonacci<32>::RET = 2178309  
fibonacci<33>::RET = 3524578  
fibonacci<34>::RET = 5702887  
fibonacci<35>::RET = 9227465  
fibonacci<36>::RET = 14930352  
fibonacci<37>::RET = 24157817  
fibonacci<38>::RET = 39088169  
fibonacci<39>::RET = 63245986  
fibonacci<40>::RET = 102334155  
duration: 0.026492  
-----
```

Abbildung 4: Test Teil 4

2 1, 2, 3, 4, 5, 6, ... / Counters of the Shelf

2.1 Lösungsidee

Diese Aufgabenstellung wurde in den folgenden 4 Dateien implementiert:

- value.hpp
- configuration.hpp
- counter.hpp
- boundedCounter.hpp

Für die Repräsentation der *Counter*-Werte wie *Inc* und *Init* wird ein *Template IntValue* für Integer implementiert. Für die Repräsentation von Double Werten werden für jeden verwendeten Wert *Structs* implementiert, da Float und Double Zahlenwerte nicht als Template Argumente gesetzt werden dürfen, daher ist die Lösung über ein Template nicht möglich. Über eine Konstante wird der aktuelle Double Wert in das *Struct* setzt.

Es werden die 3 Konfigurationen *CounterConfig*, *IncCounterConfig* und *BoundedCounterConfig* als *Templates* implementiert, wobei die Konfiguration *CounterConfig* als Basiskonfiguration fungiert, von der die anderen Konfigurationen ableiten. Die Basiskonfiguration *CounterConfig* wird die Typdefinitionen *ValueType*, *IntValue* und *IncValue* bereitstellen, die über alle Konfigurationen wieder verwendbar sind. Die Konfiguration *IncCounterConfig* setzt als *default*-Wert für das Inkrement *IntValue* $< 1 >$, somit ist der *default Counter* immer ein (*increment by one*) *Counter*.

-2: normaler counter (bzw. dessen config) sollte nichts vom increment wert wissen muessen

Es werden die zwei *Counter*-Klassen *Counter* und *BoundedCounter* implementiert, die über eine Konfiguration konfigurierbar sind. Es wird keine *IntCounter*-Klasse (*increment by one*) implementiert, da dieser *IntCounter* ein *Counter* mit Inkrement 1 ist und dieser *IntCounter* über ein *Default* für das *Template*-Argument (*IncValue*) der Konfiguration *IncCounterConfig* realisiert werden kann. Mit diesem Ansatz kann die *Counter*-Klasse wiederverwendet werden und es wird eine Klasse eingespart.

Es wird eine Konfigurator Klasse *CounterConfigurator* implementiert, die je nach gesetzten *Template*-Argumenten, den entsprechenden *Counter* liefert. Dafür wird ein Aufzählungsdatentyp implementiert, der alle verfügbaren *Counter*-Typen abbildet. *Template*-Argumente, die nicht für alle *Counter*-Typen benötigt werden, werden mit *defaults* gesetzt, damit diese optional werden und nicht ausgeprägt werden müssen. Folgende *Template*-Argumente werden auf der Konfigurator Klasse *CounterConfigurator* definiert:

- *counterType* ist das Argument, über den der *Counter*-Typ definiert wird.
- *Init* ist das Argument, über den der Initialwert des *Counters* definiert wird.
- *Bound* ist das optionale Argument, über den der *Bound* des *bounded Counters* definiert wird. Als *Default* wird *NoBound* gesetzt.
- *Inc* ist das optionale Argument, über den das Inkrement des *Counters* definiert wird. Als *Default* wird *IntValue* $< 1 >$ gesetzt.

Für die Entscheidung welcher *Counter* geliefert werden muss, wird das *IF-Statement* aus der Aufgabenstellung *like Bunnies* verwendet. Es wird eine *ELSE-IF Cascade* realisiert, die je nach gesetzter Aufzählung den entsprechenden *Counter* konfiguriert und liefert.

Sollte das Inkrement beim *bounded Counter* den Schwellwert überschreiten, so wird der *Value* auf den zuletzt gültigen Wert gesetzt.

Übung 3

Listing 4: value.hpp

```

1  #ifndef _value_h
2  #define _value_h
3
4  template<int n>
5  struct IntValue {
6      static const int value = n;
7  };
8  struct DoubleValue_1_0 {
9      static const double value;
10 };
11 struct DoubleValue_0_5 {
12     static const double value;
13 };
14 struct DoubleValue_2_0 {
15     static const double value;
16 };
17
18 const double DoubleValue_0_5::value(0.5);
19 const double DoubleValue_1_0::value(1.0);
20 const double DoubleValue_2_0::value(2.0);
21
22 #endif

```

Listing 5: configuration.hpp

```

1  #pragma once
2  #ifndef _configuration_h_
3  #define _configuration_h_
4
5  #include "counter.hpp"
6  #include "boundedCounter.hpp"
7
8  struct NoVarInc {
9  };
10 struct NoBound {
11 };
12
13 template<typename Type, typename Init, typename Inc>
14 struct CounterConfig {
15     typedef Type ValueType;
16     typedef Init InitVal;
17     typedef Inc IncVal;
18 };
19
20 template<typename Type, typename Init, typename Inc = IntValue<1>>
21 struct IncCounterConfig : public CounterConfig<Type, Init, Inc> {
22     typedef IncCounterConfig Config;
23     typedef Counter<Config> Counter;
24 };
25
26 template<typename Type, typename Init, typename Bound, typename Inc = IntValue<1>>
27 struct BoundedCounterConfig : public CounterConfig<Type, Init, Inc> {
28     typedef Bound UpperBound;
29     typedef BoundedCounterConfig Config;
30     typedef BoundedCounter<Counter<Config>> Counter;
31 };
32
33 template<bool Condition, typename Then, typename Else>

```

Xcounterconfig
boundedXcounterconfig
varincXcounterconfig
boundedvarincXcounterconfig

X = int/double

Übung 3

```

34 struct IF {
35     typedef Then RET;
36 };
37
38 template<typename Then, typename Else>
39 struct IF<false, Then, Else> {
40     typedef Else RET;
41 };
42
43 enum CounterType {
44     intInc, intIncBounded, varIntInc, varIntIncBounded,
45     doubInc, doubIncBounded, varDoubInc, varDoubIncBounded
46 };
47
48 template<CounterType counterType, typename Init, typename Bound = NoBound, typename Inc =
    ↪ NoVarInc>
49 class CounterConfigurationGenerator {
50 private:
51     enum {
52         isIntInc = counterType == intInc,
53         isIntIncBounded = counterType == intIncBounded,
54         isVarIntInc = counterType == varIntInc,
55         isVarIntIncBounded = counterType == varIntIncBounded,
56         isDoubInc = counterType == doubInc,
57         isDoubIncBounded = counterType == doubIncBounded,
58         isVarDoubInc = counterType == varDoubInc,
59         isVarDoubIncBounded = counterType == varDoubIncBounded
60     };
61
62 public:
63     typedef typename IF<isIntInc, Counter<IncCounterConfig<int, Init, IntValue<1>>>,
64         typename IF<isIntIncBounded, BoundedCounter<Counter<BoundedCounterConfig<int, Init, Bound,
        ↪ IntValue<1>>>>,
65         typename IF<isVarIntInc, Counter<IncCounterConfig<int, Init, Inc>>,
66         typename IF<isVarIntIncBounded, BoundedCounter<Counter<BoundedCounterConfig<int, Init, Bound,
        ↪ Inc>>>,
67         typename IF<isDoubInc, Counter<IncCounterConfig<double, Init, DoubleValue_1_0>>,
68         typename IF<isDoubIncBounded, BoundedCounter<Counter<BoundedCounterConfig<double, Init, Bound,
        ↪ DoubleValue_1_0>>>,
69         typename IF<isVarDoubInc, Counter<IncCounterConfig<double, Init, Inc>>,
70         BoundedCounter<Counter<BoundedCounterConfig<double, Init, Bound, Inc>>>>
71         ::RET>::RET>::RET>::RET>::RET>::RET>::RET Counter;
72 };
73 #endif

```

-1: kann aus den template argumenten ermittelt werden
vgl. list_generator aus foliensatz

es reicht:
is_bounded = ...
is_var_inc = ...

damit kann mit IFs der korrekte
counter zurueckgegeben werden

Listing 6: counter.hpp

```

1  #ifndef _counter_h
2  #define _counter_h
3
4  // Counter template
5  template<typename Configuration>
6  class Counter {
7
8  protected:
9      // Custom Type to shorten call path
10     typedef typename Configuration::Config Config;
11     // Custom Type to shorten call path
12     typedef typename Config::ValueType ValueType;
13
14 protected:
15     // The held value reference

```

Übung 3

```

16  ValueType value;
17
18  public:
19      // Initialize value for given ValueType
20      Counter() : value(Config::InitVal::value) {
21      }
22
23      // Get actual value
24      ValueType Value() const {
25          return value;
26      }
27
28      // Increment value by by one
29      void Increment() {
30          value = value + Config::IncVal::value;
31      }
32
33      // Reset value by deleting old and creating new one
34      void Reset() {
35          value = Config::InitVal::value;
36      }
37 };
38
39 #endif

```

Listing 7: boundedCounter.hpp

```

1  #ifndef _boundedCounter_h
2  #define _boundedCounter_h
3
4  // partial template bounded counter
5  template<typename BaseCounter>
6  class BoundedCounter : public BaseCounter {
7
8  protected:
9      typedef typename BaseCounter::Config Config;
10     typedef typename BaseCounter::ValueType ValueType;
11
12  protected:
13     ValueType bound;
14
15  public:
16     BoundedCounter() : bound(Config::UpperBound::value) {
17     }
18
19     void Increment() {
20         ValueType tmp = value;
21         if (value < bound) {
22             BaseCounter::Increment();
23         }
24         if (value > bound) {
25             value = tmp;
26         }
27     }
28 };
29
30 #endif

```

Listing 8: main.hpp

Übung 3

```

1  #include <iostream>
2  #include "value.hpp"
3  #include "configuration.hpp"
4  #include "counter.hpp"
5  #include "boundedCounter.hpp"
6
7  using namespace std;
8
9  int main() {
10     // Tests Integer counter
11     cout << "IncCounterConfig<int, IntValue<0>>:" << endl;
12     typedef typename IncCounterConfig<int, IntValue<0>>::Counter DefIntCounter;
13     DefIntCounter defIntCounter;
14     cout << "Inc: " << defIntCounter.Value() << endl;
15     defIntCounter.Increment();
16     cout << "Inc: " << defIntCounter.Value() << endl;
17     defIntCounter.Increment();
18     cout << "Inc: " << defIntCounter.Value() << endl;
19     defIntCounter.Increment();
20     cout << "Inc: " << defIntCounter.Value() << endl;
21     defIntCounter.Reset();
22     cout << "Reset: " << defIntCounter.Value() << endl;
23     cout << "-----" << endl << endl;
24
25     cout << "IncCounterConfig<int, IntValue<0>, IntValue<2>>:" << endl;
26     typedef typename IncCounterConfig<int, IntValue<0>, IntValue<2>>::Counter IntIncCounter;
27     IntIncCounter intIncCounter;
28     cout << "Inc: " << intIncCounter.Value() << endl;
29     intIncCounter.Increment();
30     cout << "Inc: " << intIncCounter.Value() << endl;
31     intIncCounter.Increment();
32     cout << "Inc: " << intIncCounter.Value() << endl;
33     intIncCounter.Increment();
34     cout << "Inc: " << intIncCounter.Value() << endl;
35     intIncCounter.Reset();
36     cout << "Reset: " << intIncCounter.Value() << endl;
37     cout << "-----" << endl << endl;
38
39     cout << "BoundedCounterConfig<int, IntValue<1>, IntValue<2>, IntValue<1>>:" << endl;
40     typedef typename BoundedCounterConfig<int, IntValue<1>, IntValue<2>, IntValue<1>>::Counter
41     ↪ IntBoundedCounter;
42     IntBoundedCounter intBoundedCounter;
43     cout << "Inc: " << intBoundedCounter.Value() << endl;
44     intBoundedCounter.Increment();
45     cout << "Inc: " << intBoundedCounter.Value() << endl;
46     intBoundedCounter.Increment();
47     cout << "Inc: " << intBoundedCounter.Value() << endl;
48     intBoundedCounter.Increment();
49     cout << "Inc: " << intBoundedCounter.Value() << endl;
50     intBoundedCounter.Reset();
51     cout << "Rest: " << intBoundedCounter.Value() << endl;
52     cout << "-----" << endl << endl;
53
54     // Tests Double counter
55     cout << "IncCounterConfig<double, DoubleValue_0_5>:" << endl;
56     typedef typename IncCounterConfig<double, DoubleValue_0_5>::Counter DefDoubleCounter;
57     DefDoubleCounter defDoubleCounter;
58     cout << "Inc: " << defDoubleCounter.Value() << endl;
59     defDoubleCounter.Increment();
60     cout << "Inc: " << defDoubleCounter.Value() << endl;
61     defDoubleCounter.Increment();
62     cout << "Inc: " << defDoubleCounter.Value() << endl;

```

ist schon ein varinccounter

Übung 3

```

62 defDoubleCounter.Increment();
63 cout << "Inc: " << defDoubleCounter.Value() << endl;
64 defDoubleCounter.Reset();
65 cout << "Reset: " << defDoubleCounter.Value() << endl;
66 cout << "-----" << endl << endl;
67
68 cout << "IncCounterConfig<double, DoubleValue_0_5, DoubleValue_2_0>:" << endl;
69 typedef typename IncCounterConfig<double, DoubleValue_0_5, DoubleValue_2_0>::Counter
   ↳ DoubleIncCounter;
70 DoubleIncCounter doubleIncCounter;
71 cout << "Inc: " << doubleIncCounter.Value() << endl;
72 doubleIncCounter.Increment();
73 cout << "Inc: " << doubleIncCounter.Value() << endl;
74 doubleIncCounter.Increment();
75 cout << "Inc: " << doubleIncCounter.Value() << endl;
76 doubleIncCounter.Increment();
77 cout << "Inc: " << doubleIncCounter.Value() << endl;
78 doubleIncCounter.Reset();
79 cout << "Reset: " << doubleIncCounter.Value() << endl;
80 cout << "-----" << endl << endl;
81
82 cout << "BoundedCounterConfig<double, DoubleValue_0_5, DoubleValue_2_0, DoubleValue_1_0>:" <<
   ↳ endl;
83 typedef typename BoundedCounterConfig<double, DoubleValue_0_5, DoubleValue_2_0,
   ↳ DoubleValue_1_0>::Counter DoubleBoundedCounter;
84 DoubleBoundedCounter doubleBoundedCounter;
85 cout << "Inc: " << doubleBoundedCounter.Value() << endl;
86 doubleBoundedCounter.Increment();
87 cout << "Inc: " << doubleBoundedCounter.Value() << endl;
88 doubleBoundedCounter.Increment();
89 cout << "Inc: " << doubleBoundedCounter.Value() << endl;
90 doubleBoundedCounter.Increment();
91 cout << "Inc: " << doubleBoundedCounter.Value() << endl;
92 doubleBoundedCounter.Reset();
93 cout << "Reset: " << doubleBoundedCounter.Value() << endl;
94 cout << "-----" << endl << endl;
95
96 // Tests Int Configurator
97 cout << "CounterConfigurationGenerator<intInc, IntValue<0>>:" << endl;
98 typedef typename CounterConfigurationGenerator<intInc, IntValue<0>>::Counter GenIntIncCounter;
99 GenIntIncCounter genIntIncCounter;
100 cout << "Inc: " << genIntIncCounter.Value() << endl;
101 genIntIncCounter.Increment();
102 cout << "Inc: " << genIntIncCounter.Value() << endl;
103 genIntIncCounter.Increment();
104 cout << "Inc: " << genIntIncCounter.Value() << endl;
105 genIntIncCounter.Increment();
106 cout << "Inc: " << genIntIncCounter.Value() << endl;
107 genIntIncCounter.Reset();
108 cout << "Reset: " << genIntIncCounter.Value() << endl;
109 cout << "-----" << endl << endl;
110
111 cout << "CounterConfigurationGenerator<intIncBounded, IntValue<0>, IntValue<2>>:" << endl;
112 typedef typename CounterConfigurationGenerator<intIncBounded, IntValue<0>, IntValue<2>>::Counter
   ↳ GenIntIncBoundCounter;
113 GenIntIncBoundCounter genIntIncBoundCounter;
114 cout << "Inc: " << genIntIncBoundCounter.Value() << endl;
115 genIntIncBoundCounter.Increment();
116 cout << "Inc: " << genIntIncBoundCounter.Value() << endl;
117 genIntIncBoundCounter.Increment();
118 cout << "Inc: " << genIntIncBoundCounter.Value() << endl;
119 genIntIncBoundCounter.Increment();
120 cout << "Inc: " << genIntIncBoundCounter.Value() << endl;

```

Übung 3

```

121 genIntIncBoundCounter.Reset();
122 cout << "Reset: " << genIntIncBoundCounter.Value() << endl;
123 cout << "-----" << endl << endl;
124
125 cout << "CounterConfigurationGenerator<varIntInc, IntValue<0>, NoBound, IntValue<2>>:" << endl;
126 typedef typename CounterConfigurationGenerator<varIntInc, IntValue<0>, NoBound,
    ↪ IntValue<2>>::Counter GenIntVarIncCounter;
127 GenIntVarIncCounter genIntVarIncCounter;
128 cout << "Inc: " << genIntVarIncCounter.Value() << endl;
129 genIntVarIncCounter.Increment();
130 cout << "Inc: " << genIntVarIncCounter.Value() << endl;
131 genIntVarIncCounter.Increment();
132 cout << "Inc: " << genIntVarIncCounter.Value() << endl;
133 genIntVarIncCounter.Increment();
134 cout << "Inc: " << genIntVarIncCounter.Value() << endl;
135 genIntVarIncCounter.Reset();
136 cout << "Reset: " << genIntVarIncCounter.Value() << endl;
137 cout << "-----" << endl << endl;
138
139 cout << "CounterConfigurationGenerator<varIntIncBounded, IntValue<0>, IntValue<2>,
    ↪ IntValue<1>>:" << endl;
140 typedef typename CounterConfigurationGenerator<varIntIncBounded, IntValue<0>, IntValue<2>,
    ↪ IntValue<1>>::Counter GenIntVarIncBoundCounter;
141 GenIntVarIncBoundCounter genIntVarIncBoundCounter;
142 cout << "Inc: " << genIntVarIncBoundCounter.Value() << endl;
143 genIntVarIncBoundCounter.Increment();
144 cout << "Inc: " << genIntVarIncBoundCounter.Value() << endl;
145 genIntVarIncBoundCounter.Increment();
146 cout << "Inc: " << genIntVarIncBoundCounter.Value() << endl;
147 genIntVarIncBoundCounter.Increment();
148 cout << "Inc: " << genIntVarIncBoundCounter.Value() << endl;
149 genIntVarIncBoundCounter.Reset();
150 cout << "Reset: " << genIntVarIncBoundCounter.Value() << endl;
151 cout << "-----" << endl << endl;
152
153 // Tests Double Configurator
154 cout << "CounterConfigurationGenerator<doubInc, DoubleValue_0_5>:" << endl;
155 typedef typename CounterConfigurationGenerator<doubInc, DoubleValue_0_5>::Counter
    ↪ GenDoubleIncCounter;
156 GenDoubleIncCounter genDoubleIncCounter;
157 cout << "Inc: " << genDoubleIncCounter.Value() << endl;
158 genDoubleIncCounter.Increment();
159 cout << "Inc: " << genDoubleIncCounter.Value() << endl;
160 genDoubleIncCounter.Increment();
161 cout << "Inc: " << genDoubleIncCounter.Value() << endl;
162 genDoubleIncCounter.Increment();
163 cout << "Inc: " << genDoubleIncCounter.Value() << endl;
164 genDoubleIncCounter.Reset();
165 cout << "Reset: " << genDoubleIncCounter.Value() << endl;
166 cout << "-----" << endl << endl;
167
168 cout << "CounterConfigurationGenerator<doubIncBounded, DoubleValue_0_5, DoubleValue_2_0>:" <<
    ↪ endl;
169 typedef typename CounterConfigurationGenerator<doubIncBounded, DoubleValue_0_5,
    ↪ DoubleValue_2_0>::Counter GenDoubleIncBoundCounter;
170 GenDoubleIncBoundCounter genDoubleIncBoundCounter;
171 cout << "Inc: " << genDoubleIncBoundCounter.Value() << endl;
172 genDoubleIncBoundCounter.Increment();
173 cout << "Inc: " << genDoubleIncBoundCounter.Value() << endl;
174 genDoubleIncBoundCounter.Increment();
175 cout << "Inc: " << genDoubleIncBoundCounter.Value() << endl;
176 genDoubleIncBoundCounter.Increment();
177 cout << "Inc: " << genDoubleIncBoundCounter.Value() << endl;

```


Übung 3

```

178 genDoubleIncBoundCounter.Reset();
179 cout << "Reset: " << genDoubleIncBoundCounter.Value() << endl;
180 cout << "-----" << endl << endl;
181
182 cout << "CounterConfigurationGenerator<varDoubInc, IntValue<0>, NoBound, DoubleValue_0_5>:" <<
    ↪ endl;
183 typedef typename CounterConfigurationGenerator<varDoubInc, DoubleValue_0_5, NoBound,
    ↪ DoubleValue_0_5>::Counter GenDoubleVarIncCounter;
184 GenDoubleVarIncCounter genDoubleVarIncCounter;
185 cout << "Inc: " << genDoubleVarIncCounter.Value() << endl;
186 genDoubleVarIncCounter.Increment();
187 cout << "Inc: " << genDoubleVarIncCounter.Value() << endl;
188 genDoubleVarIncCounter.Increment();
189 cout << "Inc: " << genDoubleVarIncCounter.Value() << endl;
190 genDoubleVarIncCounter.Increment();
191 cout << "Inc: " << genDoubleVarIncCounter.Value() << endl;
192 genDoubleVarIncCounter.Reset();
193 cout << "Reset: " << genDoubleVarIncCounter.Value() << endl;
194 cout << "-----" << endl << endl;
195
196 cout << "CounterConfigurationGenerator<varDoubIncBounded, DoubleValue_0_5, DoubleValue_2_0,
    ↪ DoubleValue_1_0>:" << endl;
197 typedef typename CounterConfigurationGenerator<varDoubIncBounded, DoubleValue_0_5,
    ↪ DoubleValue_2_0, DoubleValue_1_0>::Counter GenDoubleVarIncBoundCounter;
198 GenDoubleVarIncBoundCounter genDoubleVarIncBoundCounter;
199 cout << "Inc: " << genDoubleVarIncBoundCounter.Value() << endl;
200 genDoubleVarIncBoundCounter.Increment();
201 cout << "Inc: " << genDoubleVarIncBoundCounter.Value() << endl;
202 genDoubleVarIncBoundCounter.Increment();
203 cout << "Inc: " << genDoubleVarIncBoundCounter.Value() << endl;
204 genDoubleVarIncBoundCounter.Increment();
205 cout << "Inc: " << genDoubleVarIncBoundCounter.Value() << endl;
206 genDoubleVarIncBoundCounter.Reset();
207 cout << "Reset: " << genDoubleVarIncBoundCounter.Value() << endl;
208 cout << "-----" << endl << endl;
209
210 return 0;
211 }

```

Übung 3

2.2 Tests

```

IncCounterConfig<int, IntValue<0>>>:
Inc: 0
Inc: 1
Inc: 2
Inc: 3
Reset: 0
-----

IncCounterConfig<int, IntValue<0>, IntValue<2>>>:
Inc: 0
Inc: 2
Inc: 4
Inc: 6
Reset: 0
-----

BoundedCounterConfig<int, IntValue<1>, IntValue<2>, IntValue<1>>>:
Inc: 1
Inc: 2
Inc: 2
Inc: 2
Reset: 1
-----

IncCounterConfig<double, DoubleValue_0_5>:
Inc: 0.5
Inc: 1.5
Inc: 2.5
Inc: 3.5
Reset: 0.5
-----

IncCounterConfig<double, DoubleValue_0_5, DoubleValue_2_0>:
Inc: 0.5
Inc: 2.5
Inc: 4.5
Inc: 6.5
Reset: 0.5
-----

BoundedCounterConfig<double, DoubleValue_0_5, DoubleValue_2_0, DoubleValue_1_0>:
Inc: 0.5
Inc: 1.5
Inc: 1.5
Inc: 1.5
Reset: 0.5
-----

```

Abbildung 5: Test Teil 1

Übung 3

```
CounterConfigurationGenerator<intInc, IntValue<0>>:
```

```
Inc: 0
Inc: 1
Inc: 2
Inc: 3
Reset: 0
```

```
CounterConfigurationGenerator<intIncBounded, IntValue<0>, IntValue<2>>:
```

```
Inc: 0
Inc: 1
Inc: 2
Inc: 2
Reset: 0
```

```
CounterConfigurationGenerator<varIntInc, IntValue<0>, NoBound, IntValue<2>>:
```

```
Inc: 0
Inc: 2
Inc: 4
Inc: 6
Reset: 0
```

```
CounterConfigurationGenerator<varIntIncBounded, IntValue<0>, IntValue<2>, IntValue<1>>:
```

```
Inc: 0
Inc: 1
Inc: 2
Inc: 2
Reset: 0
```

```
CounterConfigurationGenerator<doubInc, DoubleValue_0_5>:
```

```
Inc: 0.5
Inc: 1.5
Inc: 2.5
Inc: 3.5
Reset: 0.5
```

```
CounterConfigurationGenerator<doubIncBounded, DoubleValue_0_5, DoubleValue_2_0>:
```

```
Inc: 0.5
Inc: 1.5
Inc: 1.5
Inc: 1.5
Reset: 0.5
```

```
CounterConfigurationGenerator<varDoubInc, IntValue<0>, NoBound, DoubleValue_0_5>:
```

```
Inc: 0.5
Inc: 1
Inc: 1.5
Inc: 2
Reset: 0.5
```

```
CounterConfigurationGenerator<varDoubIncBounded, DoubleValue_0_5, DoubleValue_2_0, DoubleValue_1_0>:
```

```
Inc: 0.5
Inc: 1.5
Inc: 1.5
Inc: 1.5
Reset: 0.5
```

Abbildung 6: Test Teil 2