Metaprogrammierung in C++

Die Programmiersprache C++ ist sehr komplex. Viele Programmierer benutzen C++, aber vermutlich beherrschen nur relativ wenige die Sprache vollständig. Selbst der maßgebliche Entwickler von C++, Bjarne Stroustrup, wurde von einigen Eigenschaften seiner Sprache überrascht.

1994 fand Erwin Unruh (siehe `www.erwin-unruh.de`) innerhalb der Programmiersprache C++ eine weitere Programmiersprache, die wir hier mit M bezeichnen. Die Sprache M besteht im Kern und vor allem aus den C++-Befehlen zum Vereinbaren und Benutzen von *Klassenschablonen* (class templates). Diese Befehle werden allerdings auf eine spezielle Weise benutzt, die von der ursprünglich intendierten Benutzung deutlich abweicht. *Typen* spielen dabei die Rolle von *Daten* und *Schablonen* die Rolle von *Funktionen*, die man auf diese Daten anwendet.

Die Sprache M hat folgende charakteristischen Merkmale:

1. M-Befehle werden ausgeführt, wenn die umgebende C++-Datei *kompiliert* wird (nicht erst zur Laufzeit, wenn das compilierte C++-Programm ausgeführt wird).

2. Die Sprache M ist *turingmächtig*. Das bedeutet: Im Prinzip kann man jedes Problem, welches überhaupt mit einem Programm lösbar ist, auch mit einem M-Programm lösen.

3. M ist eine *funktionale* Sprache. Das bedeutet: Es gibt in M zwar Konstanten und Funktionen, aber keine Variablen, deren Werte man (z. B. mit einem Zuweisungsbefehl) beliebig oft verändern könnte.

In einem C++-Programm kann man mit M-Befehlen z. B. prüfen, ob die generischen Parameter einer Schablone bestimmte Bedingungen erfüllen (ob z. B. ein als aktueller Parameter angegebener Typ ein Ganzzahltyp ist). Andere Anwendungen bestehen darin, dass man in Abhängigkeit von den näheren Umständen, unter denen ein C++-Programm kompiliert wird, eines von mehreren Unterprogrammen auswählt, welches unter diesen Umständen besonders günstig (z. B. schnell) ist.

Das Schreiben und Benutzen von Befehlen der Sprache M wird häufig als *Metaprogrammierung* bezeichnet.

1 Klassenschablonen in C++

Den Kern der Sprache M bilden C++-Klassenschablonen mit Typparametern (Schablonen mit anderen, nicht-Typ-Parametern, spielen dagegen kaum eine Rolle). Es folgt ein Beispiel für eine Schablone mit drei Typparametern:

Beispiel-1-01: Eine Klassenschablone `tripel` mit drei Typparametern

```
1   template <class T1, class T2, class T3>
2   struct tripel {
3      T1 k1;
4      T2 k2;
5      T3 k3;
6      tripel(T1 k1, T2 k2, T3 k3) : k1(k1), k2(k2), k3(k3) {}
7
8      static const int nr = 3;
9   };
```

In Zeile 1 werden die drei generischen formalen Parameter `T1`, `T2` und `T3` durch das Schlüsselwort `class` als *Typparameter* gekennzeichnet. Das Schlüsselwort `class` ist hier mild irreführend: Anstelle von `T1` darf man beim Instanziieren der Schablone einen *beliebigen Typ* (z. B. `int[]` oder `int` oder `string` oder `vector<int>`) angeben, nicht nur Klassentypen. Statt `class` darf man

an dieser Stelle auch `typename` schreiben, aber dieses Schlüsselwort hat auch noch eine ganz andere Bedeutung und wird hier deshalb vermieden.

In C++ kann man eine Klasse (oder Klassenschablone) wahlweise mit `struct` (wie in Zeile 2) oder mit `class` beginnen. Verwendet man `struct`, sind alle nicht anders gekennzeichneten Elemente der Klasse `public`, verwendet man `class` sind die Elemente dagegen standardmäßig `private`. Das ist der einzige Unterschied zwischen `struct` und `class` an dieser Stelle.

In Zeile 6 wird ein Konstruktor vereinbart, der mit seinen drei Parametern `k1`, `k2`, `k3` die drei Objektattribute `k1`, `k2`, `k3` initialisiert.

Für eine Schablone wie `tripel` kann man beliebig viele (null oder mehr) *Spezialisierungen* vereinbaren. Eine Spezialisierung einer Schablone ist ebenfalls eine Schablone, in der einige der generischen Parameter (im Beispiel: `T1`, `T2` und `T3`) durch geeignete aktuelle Parameter ersetzt wurden.

Beispiel-1-02: Drei Spezialisierungen für die Schablone `tripel`

```
10  // Eine partielle Spezialisierung von tripel mit zwei Typparametern:
11  template <class T2, class T3>
12  struct tripel<int, T2, T3> {
13     int k1;
14     T2  k2;
15     T3  k3;
16     tripel(T2 k2, T3 k3) : k1(17), k2(k2), k3(k3) {};
17
18     static const int nr = 2;
19  };
20  //-------------------------------------------------------------------------
21  // Eine partielle Spezialisierung von tripel mit einem Typparameter:
22  template <class T2>
23  struct tripel<int, T2, int> {
24     int  k1;
25     T2   k2;
26     int  k3;
27     tripel(T2 k2) : k1(17), k2(k2), k3(17) {}
28
29     static const int nr = 1;
30  };
31  //-------------------------------------------------------------------------
32  // Eine vollstaendige Spezialisierung von tripel (natuerlich mit 0 Typpar.):
33  template <>
34  struct tripel<int, int, int> {
35     int  k1;
36     int  k2;
37     int  k3;
38     tripel() : k1(17), k2(17), k3(17) {}
39
40     static const int nr = 0;
41  };
```

Schablonen (und Spezialisierungen) kann man instanziieren, indem man für jeden generischen formalen Parameter einen geeigneten generischen aktuellen Parameter angibt. Jede Instanz einer Klassenschablone (bzw. eine Funktionsschablone) ist eine Klasse (bzw. eine Funktion).

Beispiel-1-03: Vier Instanziierungen verschiedener Schablonen

```
42  tripel<double, double, double> t3 (1.3, 2.3, 3.3);
43  tripel<int,    double, double> t2 (2.2, 2.3);
44  tripel<int,    double, int   > t1 (2.1);
45  tripel<int,    int,    int   > t0;
```

Instanzen der Schablone `tripel` bezeichnen wir als `tripel`-Klassen. In diesen Beispielen enthält jede `tripel`-Klasse ein Klassenattribut namens `nr` vom Typ `int`.

Aufgabe-1-01: Welche Werte haben die folgenden Ausdrücke?

```
t3.k1,  t3.k2,  t3.k3,
t2.k1,  t2.k2,  t2.k3,
t1.k1,  t1.k2,  t1.k3,
t0.k1,  t0.k2,  t0.k3,
tripel<double, double, double>::nr,
tripel<int,    double, double>::nr,
tripel<int,    double, int   >::nr,
tripel<int,    int,    int   >::nr,
tripel<string, long,   float >::nr.
```

2 Schablone oder Funktion? Ja!

Eine *Klassenschablone* ist eine Funktion (im mathematischen Sinne), die ihre aktuellen generischen *Parameter* auf *Klassen* abbildet.

Beispiel-2-01: Eine Klassenschablone namens `klaus`

```
1   template<int n>
2   struct klaus {
3      ...
4   };
```

Die Klassenschablone `klaus` hat einen generischen Parameter vom Typ `int`. Somit bildet sie `int`-Werte auf Klassen ab, z. B. den `int`-Wert `17` auf die Klasse `klaus<17>` und den `int`-Wert `-3` auf die Klasse `klaus<-3>` etc.

Falls in Zeile 3 irgendwelche Klassenelemente vereinbart werden, enthält jede Instanz `klaus<17>`, `klaus<-3>` etc. der Schablone `klaus` entsprechende Elemente. Man kann dann auch sagen: Die Klassenschablone `klaus` bildet `int`-Werte auf diese Klassenelemente ab.

Beispiel-2-02: Die Klassenschablone `klara` mit einem Klassenattribut namens `wert`

```
1   template<int n>
2   struct klara {
3      static const int wert = 2 * n;
4   };
```

Die Klassenschablone `klara` bildet den `int`-Wert `17` auf die Klasse `klara<17>` und somit auch auf das Klassenattribut `klarar<17>::wert` ab. Das Klassenattribut `klarar<17>::wert` hat den Wert `34`. Den `int`-Wert `-3` bildet `klara` auf die Klasse `klara<-3>` und somit auch auf das Klassenattribut `klara<-3>::wert` (mit dem Wert `-6`) ab.

In einem bestimmten Sinne ist `klara` also eine Funktion, die `int`-Werte auf `int`-Werte abbildet (jeden Wert `n` auf den Wert `2*n`). Damit besteht die Möglichkeit, die Funktion `klara` geschachtelt aufzurufen, etwa so:

Beispiel-2-03: Die Funktion `klara` geschachtelt aufrufen

```
5   typedef klara<klara<3>::wert> klaraD;
6   cout << klaraD::wert << endl;  // Ausgabe: 12
```

In Zeile 5 wird eine *Typkonstante* namens `klaraD` vereinbart. Diese Konstante bezeichnet die Klasse `klara<klara<3>::wert>`. Das ist dieselbe Klasse wie `klara<6>`. Das Klassenattribut `wert` dieser Klasse hat den Wert 12.

Die Funktion (bzw. Klassenschablone) `binaer` im folgenden Beispiel ähnelt der Funktion `klara`, ruft sich aber *rekursiv* auf. Eine Spezialisierung der Schablone sorgt dafür, dass die Rekursion nach endlich vielen Schritten aufhört und nicht endlos weitergeht:

Beispiel-2-04: Die M-Funktion (Klassenschablone) `binaer`

```
7   template<int n>
8   struct binaer {
9      static int const value = binaer<n/10>::value * 2 + n%10;
10  };
11
12  template<>
13  struct binaer<0> {
14     static int const value = 0;
15  };
```

Aufgabe-2-01: Welche Werte haben die folgenden Ausdrücke?

`binaer<0>::value`, `binaer<1>::value`, `binaer<10>::value`, `binaer<11>::value`, `binaer<101010>::value` ?

Allgemein gilt: Wendet man die Funktion `binaer` auf einen `int`-Wert `w` an, dessen dezimale Darstellung nur aus den (Binär-) Ziffern `0` und `1` besteht, dann erhält man eine Klasse `binaer<w>`, deren Klassenattribut `binaer<w>::value` die entsprechende Binärzahl als Wert hat. Z. B. hat das Klassenattribut `binaer<10>::value` den Wert 2, `binaer<11>::value` den Wert 3 und `binaer<101010>::value` den Wert 42.

Wendet man die Funktion `binaer` auf einen `int`-Wert `w` an, dessen dezimale Darstellung nicht nur Binärziffern, sondern andere Dezimalziffern enthält, bekommt das Klassenattribut `binaer<w>::value` einen „merkwürdigen Wert“.

Aufgabe-2-02: Welche Werte haben die folgenden Ausdrücke?

`binaer<5>::value`, `binaer<21>::value`?

Stark vereinfacht kann man sagen: Die Funktion `binaer` bildet bestimmte Dezimalzahlen wie 10_{10}, 11_{10} und 101010_{10} auf entsprechende Binärzahlen 10_2, 11_2 und 101010_2 ab. Diese Binärzahlen entsprechen den Dezimalzahlen 2_{10}, 3_{10} und 42_{10}.

In der Beispieldatei `Beispiel03.cpp` findet man eine Variante der Klassenschablone `binaer` namens `binary`. Instanziiert man die Schablone `binary` mit einem `int`-Wert, dessen dezimale Darstellung nicht nur Binärziffern enthält (z. B. `binary<31>`) wird vom Compiler eine Fehlermeldung ausgegeben.

Die Klassenschablone `binaer` kann man also als eine Funktion verstehen, die („nach einem merkwürdigen Schema“) `int`-Werte auf `int`-Werte abbildet. Das besondere an dieser Funktion: Ihre Aufrufe werden schon *zur Compilezeit* ausgeführt, nicht erst zur Laufzeit. Ein Ausdruck wie z. B. `binaer<11>::value` ist somit *statisch*, wie das folgende Beispiel deutlich machen soll:

Beispiel-2-05: Ein Ausdruck wie `binaer<11>` ist statisch!

```
16   int len = 5;
17   int ir1[len];               // Verboten, len         ist nicht statisch
18   int ir2[5];                 // Erlaubt, der Ausdruck 5     ist statisch
19   int ir3[binaer<101>::value]; // Erlaubt,  binaer<101>::value ist statisch
```

Terminologie: Eine Schauspielerin, die eigentlich Anna Müller heisst und die Tochter der Lehrerin Müller ist, kann auf der Bühne die Elektra, Tochter des Feldherrn Agamemnon spielen. Ganz ähnlich ist in den vorangehenden Beispielen `binaer` eigentlich eine *Klassenschablone*, spielt aber die Rolle einer *Funktion*, die jeden `int`-Wert `w` auf eine Klasse `binaer<w>` bzw. auf den `int`-Wert `binaer<w>:value` abbildet. Um „die Privatperson“ und die „Bühnenrolle“ klar zu unterscheiden, werden Gebilde wie `binaer` im Folgenden als **M-Funktionen** (M wie meta) bezeichnet, die als Klassenschablonen implementiert oder realisiert wurden. Die M-Funktion `binaer` hat einen `int`-Parameter und liefert als Ergebnis einen Typ der Form `binaer<w>` bzw. den Wert des `int`-Ausdrucks `binaer<w>::value`.

3 Eine typische M-Funktion namens binaerT

Im vorigen Abschnitt wurde die M-Funktion `binaer` vorgestellt. Sie bildet `int`-Werte auf `int`-Werte ab. Das ist eher untypisch für eine M-Funktion. Typischer ist folgende Organisation:

1. Jeder Argument- und Zielwert einer M-Funktion wird durch eine spezielle *Werteklasse* dargestellt (jeder Wert durch eine eigene Klasse, verschiedene Werte durch verschiedene Klassen!).

2. Die M-Funktion bildet dann Werteklassen auf Werteklassen ab.

Eine typische *Werteklasse* (oder: *Hüllklasse für Werte*) enthält (entsprechend einer unter Metaprogrammierern verbreiteten Konvention) vor allem zwei Klassenelemente namens `value` (ein Attribut, welches den repräsentierten Wert enthält) und `value_type` (ein `typedef`-Name, der den Typ des Attributs `value` bezeichnet). Eine *Werteschablone* bildet *Werte* auf *Werteklassen* ab.

Beispiel-3-01: Eine Werteschablone namens `int_`, die jeden `int`-Wert `n` auf eine entsprechende `int`-Werteklasse `int_<n>` abbildet, vereinbart in einer Kopfdatei namens `int.hpp`

```
1   #ifndef int_hpp
2   #define int_hpp
3
4   template<int n>
5   struct int_ {
6      static const int value = n;
7      typedef      int value_type;
8   };
9
10  #endif // ifndef int_hpp
```

Die Schablone `int_` bildet den `int`-Wert `3` auf die Klasse `int_<3>` ab. Diese Klasse hat ein Klassenattribut namens `value` mit dem Wert `3` (`int_<3>::value` ist gleich `3`). Die Klasse `int_<3>` *repräsentiert* den `int`-Wert 3.

Jetzt können wir eine typische M-Funktion definieren, die Werteklassen auf Werteklassen (oder: Typen auf Typen, statt Werte auf Werte) abbildet.

Beispiel-3-02: Eine typische M-Funktion `binaerT`, die Werteklassen der Form `int_<n>` auf ebensolche abbildet

```
11  template<class T>  // T sollte von der Form int_<n> sein
12  struct binaerT {
13     typedef int_
14                <binaerT
15                   <int_
16                      <T::value/10
17                      >
18                   >::type::value
19                   * 2 + T::value%10
20                >
21             type;
22  };
23
24  template<>         // Eine Spezialisierung der Schablone binaerT
25  struct binaerT<int_<0> > {
26     typedef int_<0> type;
27  };
```

`binaerT` ist eine Klassenschablone (siehe `struct` in Zeile 12) mit einem Typparameter `T` (siehe `class T` in Zeile 11). Jeder Instanz dieser Schablone, d. h. jede Klasse der Form `binaerT<T>` enthaelt nur ein einziges Klassenelement, nämlich einen `typedef`-Namen `type`, deklariert in den Zeilen 13 bis 21. Dieser Name `type` bezeichnet eine Werteklasse der Form `int_<...>`, wie man vor allem in Zeile 13 erkennen kann.

Die Schablone `binaerT` sollte man nur auf ganz bestimmte Typen `T` anwenden, nämlich auf Werteklassen der Form `int_<n>`. Zumindest sollte der Typ `T` eine Klasse mit einem Klassenelement namens `value` sein, und `value` sollte zu einem arithmetischen Typ gehören, sonst sind die Ausdrücke `T::value` in Zeile 16 und 18 syntaktisch falsch.

Wendet man die M-Funktion `binaerT` auf eine Klasse `int_<n>` an, so ruft die M-Funktion sich (in Zeile 14 bis 18) rekursiv auf mit einem Parameter `int_<n>::value/10` gleich `n/10` auf. Die Rekursion geht so lange weiter, bis `n` kleiner als `10` und `n/10` gleich `0` ist. In diesem speziellen Fall liefert die Spezialisierung (Zeile 23 bis 26) eine Klasse `binaerT<int_<0> >`, deren Klassenelement `type` die Werteklasse `int_<0>` bezeichnet (siehe Zeile 26).

Beispiel-3-03: Gegenüberstellung der Schablonen `binaerT` und `binaer`

```
28  template<class T>                    template<int n>
29  struct binaerT {                     struct binaer {
30     typedef int_                         static int const value =
31                <binaerT                     binaer
32                   <int_                      <
33                      <T::value/10                n/10
34                      >
35                   >::type::value             >::value
36                   * 2 + T::value%10          * 2 + n%10
37                >                            ;
38             type;
39  }; // binaerT                       }; // binaer
```

Aufgabe-3-01: Schreiben Sie eine „untypische“ M-Funktion namens `plus`, die zwei `int`-Werte addiert. Diese M-Funktion sollte als Schablone mit zwei `int`-Parametern realisiert werden. Die Instanz `plus<5,3>` dieser Schablone sollte ein Klassenattribut namens `value` vom Typ `int` mit dem Wert `8` enthalten.

Aufgabe-3-02: Schreiben Sie eine „typische“ M-Funktion namens `plus`, die zwei `int`-Werte addiert. Diese M-Funktion sollte als Klassenschablone mit zwei generischen Typparametern realisiert werden. Als generisch aktuelle Parameter sollte man Typen den Form `int_<n>` erwarten. Die Instanz `plus<int_<5>, int_<3> >` dieser Schablone sollte einen `typedef`-Namen `type` enthalten, für den gilt: der Ausdruck `type::value` ist vom Typ `int` und hat den Wert `8`.

4 Typen mit bestimmten Informationen verbinden: Eigenschaftsschablonen

Angenommen, wir wollen bestimmte C++-Typen mit bestimmten Informationen verbinden, die zur Compilezeit und Laufzeit eines Programms zur Verfügung stehen sollen. Als einfaches Beispiel wollen wir bestimmte Typen mit dem Namen ihres Autors verbinden.

Falls es sich um *Klassentypen* handelt, die wir selbst programmiert haben, könnten wir die Klassendefinitionen entsprechend um ein geeignetes Element erweitern. Bei „eingekauften Klassentypen“ ist eine solche Erweiterung ziemlich problematisch, und bei Typen, die keine Klassen sind (z. B. bei fundamentalen Typen wie `int` und `long` oder bei Reihungstypen wie `int[]` und `string[]` oder bei Adresstypen wie `int*` und `string*` etc.) können wir die Definitionen nicht erweitern, weil es für diese Typen keine Definitionen gibt.

Statt zu versuchen, vorhandene (und nicht vorhandene) Typdefinitionen zu verändern, kann man das Problem auch mit einer *Eigenschaftsschablone* (engl. trait template) lösen. Eine Eigenschaftsschablone ist eine Klassenschablone mit genau *einem Typparameter*.

Beispiel-4-01: Typen mit dem Namen ihrer Autoren verbinden

```
1   // Die Eigenschaftsschablone autor fuer den allgemeinen Fall:
2   template<class T>
3   struct autor {
4      static string name() {return "Unbekannt";}
5   };
6
7   // Eine Spezialisierung der Schablone fuer den Typ int:
8   template<>
9   struct autor<int> {
10     static string name() {return "Kernighan";}
11  };
12
13  // Eine Spezialisierung der Schablone fuer den Typ string:
14  template<>
15  struct autor<string> {
16     static string name() {return "Stroustrup";}
17  };
18
19  // Eine Spezialisierung der Schablone fuer den Typ autor<int>:
20  template<>
21  struct autor< autor<int> > {
22     static string  name() {return "Ulrich";}
23  };
```

Der Ausdruck `autor<int>::name()` hat jetzt den Wert `"Kernighan"` und der Ausdurck `autor< vector<long> >::name()` den Wert `"Unbekannt"`.

Die Eigenschaftsschablone `autor` muss zwar für jeden Typ, den wir mit einem Autornamen verbinden wollen, spezialisiert werden, aber sie hat den wichtigen Vorteil, „nicht-invasiv“ zu sein,

d. h. sie erfordert keine Änderungen an den betreffenden Typen. Ausserdem kann man sie für *beliebige* Typen spezialisieren, auch für fundamentale Typen wie `int` und `long` oder für Reihungstypen wie `int[]` und `string[]` oder für Adresstypen wie `int*` und `string*` etc.

Der C++-Standard verlangt z. B., dass jede Implementierung eine Eigenschaftsschablone namens `char_traits` und zwei Spezialisierungen `char_traits<char>` und `char_traits<wchar>` definieren muss. Jede Spezialisierung der Schablone `char_traits` muss 5 Typnamen (`char_type`, `int_type`, `off_type`, `pos_type` und `state_type`) und 11 Funktionen (`assign`, `eq`, `lt`, `compare`, `length`, `find`, `move`, `copy`, `not_eof`, `to_char_type`, `to_int_type`, `eq_int_type` und `eof`) enthalten (siehe Abschnitt 21.1 des C++-Standards). Ganz entsprechend verlangt der C++-Standard für jeden Iterator-Typ eine Spezialisierung einer Eigenschaftsschablone namens `iterator_traits`, in der 5 Typnamen (`difference_type`, `value_type`, `pointer_type`, `reference_type` und `iterator_category`) definiert werden (siehe Abschnitt 24.3.1 des C++-Standards).

5 Zwei einfache Anwendungen der boost-Bibliothek

Aufbauend auf den in den vorigen Abschnitten skizzierten Grundideen haben Programmierer der Boost-Gruppe (siehe `www.boost.org`) eine umfangreiche, quelloffene Bibliothek von Metafunktionen erstellt. Eine davon wird in den folgenden Beispielen angewendet.

Anwendungsproblem: Gleitpunktzahlen sollte man in aller Regel auch dann als *gleich* betrachten, wenn sie sich um einen kleinen Betrag unterscheiden. Die Funktionsschablone `gleich` hat einen Typparameter, den man nur durch einen *Gleitpunkttyp* ersetzen sollte. Jede Instanz der Schablone `gleich` hat drei Parameter: Die beiden zu vergleichenden Gleitpunktzahlen und eine maximale Differenz. Nur wenn die beiden Gleitpunktzahlen sich um mehr als diese maximale Differenz unterscheiden, ist das Ergebnis `false`.

Beispiel-5-01: Die Funktionsschablone `gleich`

```
24  // Datei GleitVergleich.hpp
25  #ifndef _GLEITVERGLEICH_HPP_
26  #define _GLEITVERGLEICH_HPP_
27
28  #include <cmath>                    // fuer abs (Gleitpunkttypen)
29  #include <cstdlib>                  // fuer abs (Ganzzahltypen)
30
31  #include <boost/type_traits.hpp>    // fuer boost::is_float<T>
32  #include <boost/static_assert.hpp>  // fuer BOOST_STATIC_ASSERT()
33
34  template<class GleitTyp>
35  bool gleich(GleitTyp gz1, GleitTyp gz2, GleitTyp maxDiff=3) {
36     // Liefert false, wenn sich gz1 und gz2 um mehr als maxDiff
37     // unterscheiden, und sonst true.
38
39     // Falls GleitTyp kein Gleitpunkttyp ist, gibt der folgende
40     // Befehl (zur Compilezeit) eine Fehlermeldung aus:
41     BOOST_STATIC_ASSERT(boost::is_float<GleitTyp>::value);
42
43     return std::abs(gz1 - gz2) <= maxDiff;
44  } // template gleich
45
46  #endif //_GLEITVERGLEICH_HPP_
```

Der Rumpf der Funktionsschablone `gleich` besteht im Wesentlichen aus der einen Zeile 43.

Im folgenden Beispiel darf man die Funktionsschablone `runde` ebenfalls nur mit *Gleitpunkttypen* instanziieren:

Beispiel-5-02: Gleitpunktzahlen auf eine bestimmte Anzahl von Stellen (nach oder vor dem Dezimalpunkt) runden

```
47  // Datei RundenSicher.hpp
48  #ifndef _RUNDENSICHER_HPP_
49  #define _RUNDENSICHER_HPP_
50
51  #include <cmath>                    // fuer floor
52  #include <boost/type_traits.hpp>    // fuer boost::is_float<T>
53  #include <boost/static_assert.hpp>  // fuer BOOST_STATIC_ASSERT()
54
55  // Die folgende Schablone kann man nur mit Gleitpunkttypen parametrisieren,
56  // sonst bekommt man zur Compilezeit eine Fehlermeldung:
57  template<class GleitTyp>
58  GleitTyp runde(GleitTyp z, int anzStellen) {
59     // Rundet z "auf anzStellen viele Stellen". Ist anzStellen
60     // POSITIV, z. B. 3, so wird auf 3 Stellen NACH dem Punkt
61     // gerundet. Ist anzStellen NEGATIV, z. B. -3, so wird auf
62     // 3 Stellen VOR dem Punkt ("auf glatte Tausender") gerundet.
63
64     // Falls GleitTyp kein Gleitpunkttyp ist, gibt der folgende
65     // Befehl (zur Compilezeit) eine Fehlermeldung aus:
66     BOOST_STATIC_ASSERT(boost::is_float<GleitTyp>::value);
67
68     // Mit dem faktor wird z um anzStellen viele Dezimalstellen
69     // verschoben (nach links bzw. rechts):
70     GleitTyp faktor = pow(10.0, anzStellen);
71
72     z *= faktor;        // z "hinschieben"
73     z += 0.5;           // runden
74     z  = std::floor(z); // Nachpunktstellen beseitigen
75     z /= faktor;        // z "zurueckschieben"
76      return z;
77  } // runde
78
79  #endif //_RUNDENSICHER_HPP_
```

Es folgt eine Liste weiterer Metafunktionen, die man ganz ähnlich wie `is_float` anwenden kann:

Primäre Eigenschaften	Sekundäre Eigenschaften	Erläuterungen der sekundären Eigenschaften
is_array<T>	is_arithmetic<T>	is_float || is_integral
is_class<T>	is_compound<T>	!(is_arithmetic || is_void)
is_enum<T>	is_fundamental<T>	is_arithmetic || is_void
is_float<T>	is_member_func-tion_pointer<T>	Ist ein Zeiger-auf-Objektmethode-Typ
is_function<T>	is_object<T>	! is_function || is_reference || is_void
is_integral<T>	is_scalar<T>	is_arithmetic || is_enum || is_pointer || is_member_pointer
is_member_pointer<T>		
is_pointer<T>		
is_reference<T>		
is_union<T>		
is_void<T>		

Aufgabe-5-01: Programmieren Sie eine Funktionsschablone namens `ganzToString` entsprechend der folgenden Deklaration:

```
1   template<class GanzTyp>
2   std::string ganzToString(GanzTyp g, int basis=10, int grupLen=3);
```

Die Schablone soll nur mit *Ganzzahltypen* instanziierbar sein (Versuche, sie mit anderen Typen zu instanziieren, sollen zur Compilezeit eine Fehlermeldung auslösen).

Jede Instanz dieser Schablone (d. h. jede Funktion `ganzToString`) soll ihren ersten Parameter (die Ganzzahl `g`) in einen "gut lesbaren String" umwandeln, etwa so:

```
3    short s01 = 12345;
4    short s02 = -12345;
5    short s03 = 255;
6    short s04 = -0;
7    int   i01 = 123456789;
8    int   i02 = -1000000000;
9
10   ganzToString(s01)        ist gleich "+12_345"
11   ganzToString(s02)        ist gleich "-12_345"
12   ganzToString(s03,  2)    ist gleich "+11_111_111"
13   ganzToString(s03,  2, 4) ist gleich "+1111_1111"
14   ganzToString(s03, 16)    ist gleich "+FF"
15   ganzToString(s04)        ist gleich "+0";
16   ganzToString(i01)        ist gleich "+123_456_789"
17   ganzToString(i01, 10, 4) ist gleich "+1_2345_6789"
18   ganzToString(i02)        ist gleich "-1_000_000_000"
```

Der Parameter `basis` gibt an, in welchem *Zahlensystem* die Zahl `g` dargestellt werden soll: im 2-er-System (binär) oder im 5-er-System (quintal) oder im 8-er-System (octal) oder im 16-er-System (hexadecimal) oder (ganz exotisch) im 10-er-System (decimal) oder ... etc.

Als `basis` sind nur die Zahlen 2 bis 16 erlaubt. Falls `basis` einen anderen Wert hat, soll statt dessen `10` genommen werden.

"Gruppen von Ziffern" sollen durch je einen Unterstrich `'_'` voneinander getrennt werden. Wie viele Ziffern zu einer Gruppe gehören, wird durch den Parameter `grupLen` angegeben. Falls `grupLen` einen unplausiblen Wert hat (kleiner als `1` oder größer als `10`) soll stattdessen der Wert `3` genommen werden.

Der Ergebnisstring soll immer mit einem *Vorzeichen* beginnen (`'+'` bzw. `'-'`), mindestens *eine* Ziffer, aber *keine unnötigen führenden Nullen* enthalten.