27

Hilfsfunktionen und -klassen

Dieses Kapitel behandelt die folgenden Themen:

- Relationale Operatoren
- Unterstützung der Referenzsemantik für R-Werte
- Template pair für Wertepaare
- Tupel
- Funktionsobjekte
- Binden von Argumentwerten
- Standard-Templates für rationale Zahlen, Zeit und Dauer
- Standard-Template für Wrapper- oder Hüll-Klassen

27.1 Relationale Operatoren

Die folgenden relationalen Operatoren werden im Header `<utility>` im Namespace `rel_ops`, der innerhalb des Namespaces `std` liegt, definiert.

```
namespace rel_ops {                          // Rückgabe:
 template<class T>
  bool operator!=(const T& x, const T& y); // !(x == y)
 template<class T>
  bool operator>(const T& x, const T& y); // y < x
 template<class T>
  bool operator<=(const T& x, const T& y); // !(y < x)
 template<class T>
  bool operator>=(const T& x, const T& y); // !(x < y)
}
```

Wie die rechte Seite zeigt, reicht es für die Anwendung der Operatoren aus, wenn die beteiligten Typen selbst den ==-Operator im ersten Fall (!=) und den <-Operator in den anderen Fällen zur Verfügung stellen. Wenn der Namespace `rel_ops` eingebunden wird, werden die anderen Operatoren daraus abgeleitet.

27.2 Unterstützung der Referenzsemantik für R-Werte

In diesem Abschnitt geht es um die Unterstützung der Referenzsemantik für R-Werte durch die Standardfunktionen `move()` und `forward()`. Die Grundlagen dazu finden Sie in Abschnitt 22.2 ab Seite 591.

move()

`move()` (Header `<utility>`) ist eine Funktion, die eine Referenz auf einen R-Wert zurückgibt (`&&`). Damit erleichtert sie manche Optimierungen, wie das folgende Beispiel zeigt. Zur Klasse `StringTyp` siehe Abschnitt 22.3 (Seite 596). Für interne Zwecke hat diese Klasse eine private Funktion `swap()` zur Vertauschung zweier Strings. Nun soll die Möglichkeit gegeben werden, auch außerhalb der Klasse eine Vertauschung zweier `StringTyp`-Objekte vorzunehmen. Weil es bereits eine Standardfunktion `swap()` gibt und es hier weniger um das Vertauschen, sondern um die Anwendung und Erklärung zu `move()` geht, heißt diese Funktion `void tauschen(StringTyp& a, StringTyp& b)`. Eine Implementierung könnte wie folgt aussehen:

```
void tauschen0(StringTyp& a, StringTyp& b) { // Version 1
   StringTyp tmp(a);       // Kopierkonstruktor
   a = b;                  // operator=()
   b = tmp;                // operator=()
}
```

Wie im Kommentar angegeben, werden einmal der Kopierkonstruktor und zweimal der Zuweisungsoperator aufgerufen. Jede dieser Operationen verursacht `new` und `delete` – aufwendige Operationen im Vergleich zum Tauschen von Adressen. Tatsache ist, dass der Inhalt des Objekts `a` nach Kopie in der ersten Zeile gar nicht mehr gebraucht wird. Dasselbe gilt für `b` in der zweiten Zeile. Hier kommt nun `move()` ins Spiel!

```
void tauschen(StringTyp& a, StringTyp& b) { // Version 2
   if(&a != &b) {
      StringTyp tmp(move(a)); // moving Konstruktor
      a = move(b);            // moving operator=()
      b = move(tmp);          // moving operator=()
   }
}
```

Der Vergleich `if(&a != &b)` muss hier sein, weil `move()` die Ressourcen »stiehlt«. Wenn `&a == &b` wäre, würde sich ohne die Abfrage `a` mit der Anweisung `a = move(b);` selbst die Ressourcen entziehen.

Weil `move()` eine Referenz auf einen R-Wert zurückgibt, können die erheblich effizienteren Varianten `StringTyp(StringTyp&&)` und `operator=(StringTyp&&)` aufgerufen werden. Anstelle der `move()`-Funktion hätte man alternativ eine statische Typumwandlung einsetzen können (`static_cast<StringTyp&&>(a)`). Die `move()`-Funktion hat den Vorteil, dass sie auch von Compilern, die Referenzen auf R-Werte noch nicht unterstützen, akzeptiert wird. `move()` tut dann einfach nichts, außer direkt das Argument zurückzugeben. Anstelle des moving-Konstruktors würde der normale Kopierkonstruktor treten. Mit `move()` geschriebener Code muss nach einem Update des Compilers, der dann möglicherweise eine geänderte Implementation von `move()` enthält, nicht geändert werden. Mit einer weiteren Modifikation könnten auch temporäre Objekte getauscht werden:

```
StringTyp a("einA");
tauschen(a, StringTyp("temporaer")); // oder
tauschen(StringTyp("temporaer"), a);
```

Dazu muss der R-Wert per `&&` übergeben werden, das heißt, es muss noch die folgenden überladenen Varianten geben:

Listing 27.1: move()

```
// vollständiges Beispiel siehe cppbuch/k27/move/move.cpp
void tauschen(StringTyp& a, StringTyp&& b) {
   StringTyp tmp(move(a)); // moving (bewegender) Konstruktor
   a = move(b);            // moving operator=()
   b = move(tmp);          // moving operator=()
}

void tauschen(StringTyp&& a, StringTyp& b) {
   StringTyp tmp(move(a)); // moving Konstruktor
   a = move(b);            // moving operator=()
   b = move(tmp);          // moving operator=()
}
```

move() für Container-Bereiche

Um bequem ganze Bereiche mit `move()` bewegen zu können, gibt es die Funktion

```
template<class InputIterator, class OutputIterator>
OutputIterator move(InputIterator first,
                    InputIterator last,
                    OutputIterator result);
```

Der Header ist `<algorithm>`. Die Funktion bewegt die Elemente aus dem Bereich `[first, last)` in den Bereich `result, result + (last-first))`, indem für jedes `n < (last-first)` die Anweisung `*(result+n) = std::move(*(first+n));` ausgeführt wird. Die Funktion

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 move_backward(BidirectionalIterator1 first,
                                     BidirectionalIterator1 last,
                                     BidirectionalIterator2 result);
```

bewegt die Elemente aus dem Bereich `[first, last)` in den Bereich `result - (last-first), result)`, wobei am Ende (`last-1`) angefangen wird. `result` darf nicht im Bereich `[first, last)` liegen, damit nicht noch zu bewegende Daten zerstört werden.

forward()

`forward()` ist eine Funktion, die das sogenannte »perfect forwarding« unterstützt. »Perfect forwarding« ist dann gefragt, wenn man eine Template-Funktion mit Referenz-Parametern schreiben will, und wenn beabsichtigt ist, diese Parameter an eine andere Funktion weiterzuleiten. Das Problem dabei ist, dass zum Beispiel ein temporäres Objekt zwar zum Parametertyp `const&` passt, die Eigenschaft »temporär« aber nicht an die andere Funktion weitergeleitet werden kann. `forward()` hilft dabei und ermöglicht es, viele überladene Funktionen durch ein Template zu ersetzen. Weil `forward()` hauptsächlich in Bibliotheksfunktionen zum Tragen kommt, wird es hier nicht weiter diskutiert. Sie finden ein Beispiel im Verzeichnis *cppbuch/k27/forward*, und eine gute Darstellung mit weiteren Beispielen unter *http://msdn.microsoft.com/en-us/library/dd293668.aspx*.

27.3 Paare

Das Template `pair` erlaubt die Kombination heterogener Wertepaare:

Listing 27.2: Template pair (vereinfachter Auszug)

```
template<class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;
  pair(const T1& x, const T2& y);
  template<class U, class V> pair(const pair< U, V>& p);
  template<class U, class V> pair(pair< U, V>&& p);
  template<class U, class V> pair& operator=(pair< U, V>&& p);
  void swap(pair& p);
  // ...
};
```

Der (nicht aufgeführte) Standardkonstruktor initialisiert die Bestandteile mit ihrem jeweiligen Standardkonstruktor. Gleichheit und der Vergleich sind ebenfalls definiert:

```
// gibt x.first == y.first && x.second == y.second zurück
template<class T1, class T2>
bool operator==(const pair<T1, T2>& x,
                const pair<T1, T2>& y);

template<class T1, class T2>
bool operator<(const pair<T1, T2>& x,
               const pair<T1, T2>& y);
```

Die anderen relationalen Operatoren werden hier aus Platzgründen nicht aufgeführt; sie werden entsprechend Abschnitt 27.1 abgeleitet. Was bedeutet nun der Rückgabewert für `operator<()`? Antwort:

`true` : falls `x.first < y.first`

`false` : falls `y.first < x.first`

Falls beide Bedingungen nicht zutreffen, wird `x.second < y.second` zurückgegeben, mit anderen Worten: Es liegt ein lexikografischer Vergleich vor. Die Hilfsfunktion

```
template<class T1, class T2>
pair<T1, T2> make_pair(const T1& x, const T2& y);
```

erzeugt ein Paar aus den Parametern. Die beteiligten Typen werden aus den Parametern abgeleitet. Zum Beispiel gibt der Aufruf

```
return make_pair(1, 2.782);
```

ein `pair<int, double>`-Objekt zurück. Alternativ kann der gewünschte Typ angegeben werden:

```
return make_pair<int, double>(1, 2.782);
```

Das Beispiel zeigt `make_pair()` und die Anwendung des `<`-Operators.

Listing 27.3: Vergleich zweier pair-Objekte

```
// cppbuch/k27/pair/pair.cpp
#include <utility>
#include<iostream>
using namespace std;

int main() {
   pair<string, string> p1 = make_pair("Donald", "Knuth");
   pair<string, string> p2 = make_pair("Donald", "Duck");
   cout << "Vergleich zweier Paare" << endl;
   if(p1 > p2) { // Anwendung von Operator >
      cout << p2.first << " " << p2.second
           << " liegt alphabetisch vor "
           << p1.first << " " << p1.second << endl;
   }
   else {
      cout << p1.first << " " << p1.second
           << " liegt alphabetisch vor "
           << p2.first << " " << p2.second << endl;
   }
}
```

Um `pair` kompatibel zu `tuple` zu machen, stellt der Header `<utility>` die Funktionen `get<0>(const pair<T1, T2>& p)` und `get<1>(const pair<T1, T2>& p)` für den Zugriff auf die `pair`-Elemente `first` bzw. `second` zur Verfügung. Einzelheiten finden Sie im folgenden Abschnitt.

27.4 Tupel

Das Template `tuple` ist eine Verallgemeinerung des obigen `pair`-Templates auf weniger oder mehr als zwei heterogene Elemente, wobei mindestens 10 möglich sein sollen. Diese Kombination heißt *Tupel* und wird durch das Template `tuple` realisiert. Es ist nicht im Header `<utility>` angesiedelt, sondern in `<tuple>`.

```
tuple<> nichts;                 // enthält kein Element
tuple<float> eins;              // enthält ein float-Element
tuple<int, float> zwei;         // enthält ein int- und ein float-Element
tuple<int, float, string> drei; // enthält zusätzlich ein string-Element
// usw.
```

Bei mehreren Elementen wäre der Zugriff über `first`, `second` usw. wie bei `pair` zu umständlich. Deswegen gibt es spezielle Template-Funktionen. Der Aufruf `get<n>(einTupel)` gibt das n-te Element des Tupels `einTupel` zurück. Das folgende Beispiel zeigt Tupel mit zwei und drei Elementen, `make_tupel()` und die Extraktion der Elemente mit `get<n>()`.

Listing 27.4: Tupel

```
// cppbuch/k27/tuple/tuple.cpp
#include<iostream>
#include<string>
#include<tuple>
using namespace std;

int main() {
   tuple<string, string> t2("Donald", "Knuth");
   tuple<string, string, int> t3("Donald", "Duck", 17);
   // get<>() gibt Referenzen zurück
   cout << "Zugriff mit get<>: "
        << get<0>(t2) << " " << get<1>(t2) // Donald Knuth
        << endl;
   get<0>(t3) = "Dagobert";                // Änderung
   cout << get<0>(t3) << " " << get<1>(t3) // Dagobert Duck
        << endl;
   // Abfrage der Anzahl der Elemente mit tuple_size<T>::value
   typedef tuple<string, string> tupeltyp;
   cout << "tupeltyp hat "
        << tuple_size<tupeltyp>::value   // 2
        << " Elemente" << endl;
}
```

Eine umfangreichere Anwendung von `tuple` finden Sie im Abschnitt 24.10.1. Dort werden Tupel zum »Aufsammeln« von Operanden benutzt, um Operatoren möglichst effizient zu überladen.

27.5 Funktionsobjekte

Im Header `<functional>` sind Klassen definiert, die dem Erzeugen verschiedener Funktionsobjekte dienen. Funktionsobjekte haben alle einen `operator()()` und werden ausführlich ab Seite 344 beschrieben.

Im nächsten Abschnitt sind einige Klassen für Funktionsobjekte aufgeführt. Dieser Programmausschnitt zeigt eine einfache Anwendung:

```
// Auszug aus cppbuch/k27/funktionsobjekt/addierer.cpp
std::plus<double> addierer;
double d1 = 3.1415926;
double d2 = 2.718;
std::cout << d1 << " + " << d2 << " = "
          << addierer(d1, d2) << std::endl;
```

27.5.1 Arithmetische, vergleichende und logische Operationen

In der Rückgabespalte der folgenden Klassen für Funktionsobjekte meint `x` das erste und `y` das zweite Argument.

```
// arithmetische Operationen:
template<class T> struct plus;      // x + y
template<class T> struct minus;     // x - y
template<class T> struct multiplies; // x * y
template<class T> struct divides;   // x / y
template<class T> struct modulus;   // x % y
template<class T> struct negate;    // -x
```

```
// Vergleiche:
template<class T> struct equal_to;  // x == y
template<class T> struct not_equal_to; // x != y
template<class T> struct greater;   // x > y
template<class T> struct less;      // x < y
template<class T> struct greater_equal; // x >= y
template<class T> struct less_equal; // x <= y
```

```
// logische Operationen:
template<class T> struct logical_and; // x && y
template<class T> struct logical_or; // x || y
template<class T> struct logical_not; // !x
```

27.5.2 Funktionsobjekte zum Negieren logischer Prädikate

`not1()` und `not2()` sind Funktionen, die ein Funktionsobjekt zurückgeben, dessen Aufruf ein Prädikat negiert. Das Prädikat kann einen (`not1()`) oder zwei (`not2()`) Parameter haben. Die zurückgegebenen Funktionsobjekte sind vom Typ `unary_negate` bzw. `binary_negate`:

```
template<class Predicate>
class unary_negate {
```

```
  public:
    explicit unary_negate(const Predicate& pred)
    : P(pred) {
    }

    bool operator()(const typename Predicate::argument_type& x) const {
        return !P(x);
    }

    typedef typename Predicate::argument_type argument_type;
    typedef bool result_type;
  protected:
    Predicate P;
};
```

```
template<class Predicate>
class binary_negate {
  public:
    explicit binary_negate(const Predicate& pred)
    : P(pred) {
    }

    bool operator()(
         const typename Predicate::first_argument_type& x,
         const typename Predicate::second_argument_type& y) const {
           return !P(x, y);
    }

    typedef typename Predicate::first_argument_type first_argument_type;
    typedef typename Predicate::second_argument_type second_argument_type;
    typedef bool result_type;
  protected:
    Predicate P;
};
```

Wenn `P` ein Prädikat des Typs `Predicate` ist, gibt der Aufruf `not1(P)` das Objekt `unary_negate<Predicate>(P)` zurück. `not2(P)` liefert dementsprechend das Objekt `binary_negate<Predicate>(P)`. Eine mögliche Anwendung sei hier am Beispiel einer Sortierfunktion demonstriert, die ein Funktionsobjekt zum Vergleich der `int`-Elemente der Tabelle benötigt:

```
sortieren(Tabelle, Anzahl, less<int>());
// umgekehrte Reihenfolge:
sortieren(Tabelle, Anzahl, not2(less<int>()));
```

27.5.3 Binden von Argumentwerten

Die Template-Funktion `bind()` bindet Parameter oder Werte an eine Funktion. Das von `bind()` zurückgegebene Objekt ist ein Funktionsobjekt und kann wie eine Funktion aufgerufen werden (überladener `operator()()`). Das folgende Beispiel zeigt die anschließend genauer diskutierte Wirkungsweise.

Listing 27.5: Binden von Argumentwerten mit bind()

```
// cppbuch/k27/bind/bind1.cpp
#include <functional>
#include <iostream>
using std::bind;
using namespace std::placeholders;

template<class T>       // T muss ein Typ für Funktionsobjekte sein
void aufrufen(T funcObj) {
   int i1 = 1;
   int i2 = 3;
   int i3 = 5;
   int doppel = funcObj(i1, i2, i3); // Funktionsaufruf über Objekt
   std::cout << doppel << std::endl;
}

int verdoppeln(int i) {
   return 2*i;
}

int main() {
   aufrufen(bind(verdoppeln, 9)); // Funktion an Wert binden
   aufrufen(bind(verdoppeln, _1)); // Funktion an Parameter 1 binden
   aufrufen(bind(verdoppeln, _2)); // Funktion an Parameter 2 binden
   aufrufen(bind(verdoppeln, _3)); // Funktion an Parameter 3 binden
}
```

1. In der `main()`-Funktion wird zuerst der Zahlenwert 9 an die Funktion `verdoppeln()` gebunden. Der Aufruf des zurückgegebenen Funktionsobjekts geschieht innerhalb der Funktion `aufrufen()`, wobei die Parameter `i1`, `i2` und `i3` alle ignoriert werden, weil ja schon ein Wert an die Funktion gebunden ist. Es wird das Doppelte von 9, also 18, ausgegeben.
2. Im nächsten Schritt wird die Funktion `verdoppeln()` an den ersten Parameter gebunden. Das bedeutet: Ruft man das Funktionsobjekt mit mehreren Parametern auf, wird nur der erste ausgewählt. Der Wert des ersten Parameters (`i1`) ist 1, also wird 2 ausgegeben. Dabei ist `_1` eine vordefinierte Größe, die im Namespace `std::placeholders` definiert ist.
3. Entsprechendes gilt für die nächsten Anweisungen und `_2` bzw. `_3`.

Dem Funktionsobjekt können mehrere Parameter übergeben werden. Wie viele es maximal sein dürfen, ist implementationsabhängig, aber es sollen mindestens 10 sein. Die Funktion `verdoppeln()` benötigt nur einen Parameter, `bind()` ist aber für mehrere Parameter geeignet, was im Folgenden anhand einer Funktion, die zwei Parameter benötigt, gezeigt werden soll. Der bekannte Funktor `less` vergleicht zwei Werte miteinander und gibt `true` zurück, wenn der erste Wert kleiner ist. Wenn der zweite Wert festgelegt ist, zum Beispiel auf 103, genügt ein Funktionsobjekt, das mit `bind` erzeugt wird, um den ersten Wert, der kleiner ist (hier: 101), herauszufinden. Der verwendete Algorithmus `find_if()` wird weiter unten in Abschnitt 30 beschrieben.

Listing 27.6: Prädikat mit bind() verwenden

```
// cppbuch/k27/bind/bind2.cpp
#include <functional>
#include<vector>
#include<iostream>
#include<algorithm>

using std::bind;
using namespace std::placeholders;

int main() {
   std::vector<int> v;
   v.push_back(111);
   v.push_back(107);
   v.push_back(101);
   v.push_back(90);
   v.push_back(106);

   int wert = 103;
   std::cout << "Das erste Element < " << wert << " ist: " <<
      *find_if(v.begin(), v.end(),
              bind(std::less<int>(), _1, wert)) << std::endl;
}
```

27.5.4 Funktionen in Objekte umwandeln

Funktionsobjekte, auch Funktoren genannt, sind flexibler als Funktionszeiger. Von vielen Algorithmen der Standardbibliothek gibt es zwei Varianten: eine mit einem Standardverhalten, unde eine, die als Parameter ein Funktionobjekt nimmt, um das Verhalten des Algorithmus zu ändern. Die übergebene Funktion heißt Callback-Funktion, weil sie aus dem Algorithmus heraus aufgerufen wird.

Das Klassen-Template `function` erzeugt Funktionsobjekte aus Funktionen, Zeigern auf Funktionen und anderen Funktionsobjekten. Das folgende Programm zeigt, wie eine Tabelle aus Funktoren mit verschiedenen Funktionen aufgebaut wird, die als Callback-Funktion in der Funktion `berechnen()` dienen. `berechnen()` verknüpft die Elemente zweier Vektoren `a` und `b` mit der übergebenen Funktion und legt das Ergebnis in einem anderen Vektor ab. Die jeweilige Funktion wird in einer Schleife über den Tabellenindex ausgewählt.

Listing 27.7: Tabelle mit Callback-Funktionen

```
// cppbuch/k27/funktionsobjekt/function.cpp
#include <algorithm>
#include <cassert>
#include <functional> // function
#include <iostream>
#include <vector>
#include <showSequence.h> // cppbuch/include

void berechnung(const std::vector<double>& v1,
                const std::vector<double>& v2,
```

```
              std::vector<double>& ergebnis,
              const std::function<double (double, double)>& f) {
   assert((v1.size() == v2.size()) && (v1.size() == ergebnis.size()));
   for(size_t i = 0; i < v1.size(); ++i) {
      ergebnis[i] = f(v1[i], v2[i]);      // Funktionsaufruf
   }
}

double differenzquadrat(double arg1, double arg2) {
   return (arg1 - arg2)*(arg1 - arg2);
}

using namespace std;

int main() {
   // als Variable verwenden
   function<double (double, double)> malnehmen = multiplies<double>();
   cout << malnehmen(4, 42) << endl;

   vector<double> a = {1.00, 2.00, 3.00, 4.00, 5.00};
   vector<double> b = {1.01, 2.02, 3.03, 4.04, 5.05};
   vector<double> erg(b.size());

   // Tabelle mit Funktionen
   vector<function<double (double, double)> > funktionen;
   funktionen.push_back(plus<double>()); // std::-Funktor
   funktionen.push_back(differenzquadrat); // Funktion
   double (*fptr)(double, double) = differenzquadrat;
   funktionen.push_back(fptr);           // Funktionszeiger

   // Zugriff auf verschiedene Funktionen per Index i
   for(size_t i = 0; i < funktionen.size(); ++i) {
      berechnung(a, b, erg, funktionen[i]);
      showSequence(erg);
   }
}
```

Elementfunktionen in Objekte umwandeln

Gelegentlich möchte man für alle Objekte einer Klasse eine Elementfunktion ausführen. So wäre es denkbar, alle in einem Vektor abgelegten graphischen Objekte zeichnen zu lassen. Mit Bezug auf die Klasse `GraphObj` des Abschnitts 7.6.2 könnte die Anweisung wie folgt lauten:

```
for_each(objekte.begin(), objekte.end(), &GraphObj::zeichnen); // Fehler!
```

Hier besteht das Problem, dass der Standard-Algorithmus `for_each()` ein Funktionsobjekt oder eine Funktion erwartet – aber keine Methode einer Klasse. Für solche Zwecke gibt es die Funktion `mem_fn()`, die eine Elementfunktion in ein Funktionsobjekt umwandelt. Das Programm zeigt, dass damit auch die polymorphe Nutzung funktioniert:

Listing 27.8: Elementfunktion als Funktionsobjekt nutzen

```
// cppbuch/k27/funktionsobjekt/memfn.cpp
#include<algorithm>
#include<functional>
#include<vector>
#include"../../k7/abstrakt/quadrat.h"

using namespace std;

int main() {
   Quadrat q(Ort(100, 20), 50);
   Rechteck r(Ort(30, 40), 20, 20);
   vector<GraphObj*> objekte = {&q, &r};
   // alle Objekte zeichnen (polymorph)
   for_each(objekte.begin(), objekte.end(), mem_fn(&GraphObj::zeichnen));
}
```

Man könnte hier auch `function` nehmen, es wäre aber etwas umständlicher:

```
function<void (GraphObj*)> f = &GraphObj::zeichnen;
for_each(objekte.begin(), objekte.end(), f);
```

27.6 Templates für rationale Zahlen

Der Header `<ratio>` definiert die Klasse `ratio` für rationale Zahlen, die zur Compilationszeit festliegen. Die Klasse ist wie folgt definiert:

```
template <intmax_t N, intmax_t D = 1> //
class ratio {
  public:
    typedef ratio<num, den> type;
    static const intmax_t num;  // numerator (Zähler)
    static const intmax_t den;  // denominator (Nenner)
};
```

Zähler und Nenner sind Bestandteile des Typs. `intmax_t` ist ein im C-Standard [ISOC] festgelegter `int`-Datentyp für große Zahlen, der im Header `<cstdint>` definiert ist. Auf meinem System ist `sizeof(intmax_t)` gleich 8 (Bytes). Für Zehnerpotenzen gibt es vordefinierte Namen entsprechend dem Internationalen Einheitensystem (SI). So sind zum Beispiel micro und mega definiert:

```
typedef ratio<1, 1000000> micro;
typedef ratio<1000000, 1> mega;
```

Es gibt ergänzend Templates zur Unterstützung von Rechenoperationen, die den Typ des Ergebnisses zurückgeben. Zum Beispiel ist der Typ `ratio_multiply<micro, mega>::type` identisch mit `ratio<1, 1>`.

Kürzen des Bruchs, also die Division durch den größten gemeinsamen Teiler, wird zur Compilationszeit vorgenommen (sonst wäre der Typ `ratio<1000000, 1000000>`). Falls Sie rätseln sollten, wie das funktioniert, bitte ich Sie, einen Blick in den Abschnitt 6.4 (Seite 251) zu werfen. Abschließend sei ein Programm gezeigt, das Zähler und Nenner einiger `ratio`-Typen ausgibt und auch die Arithmetik zur Compilationszeit zeigt. Die Funktion `printRatio()` ist eine Hilfsfunktion zur Anzeige mit einem Bruchstrich. Die Funktion benötigt keine Argumente; nur ihr Typ ist entscheidend. Weitere Einzelheiten finden Sie in [ISOC++].

Listing 27.9: `ratio`-Arithmetik

```
// cppbuch/k27/ratio/main.cpp
#include<ratio>
#include<iostream>

template<typename T>
void printRatio() {
   std::cout << T::num << '/' << T::den << std::endl;
}

using namespace std;

int main() {
   cout << mega::num << endl; // Zähler
   cout << mega::den << endl; // Nenner
   cout << micro::num << endl; // Zähler
   cout << micro::den << endl; // Nenner

   // Arithmetik zur Compilationszeit
   cout << ratio<7, -21>::num << endl; // -1
   cout << ratio<7, -21>::den << endl; // 3

   cout << ratio_add<mega, micro>::type::num << endl; // 1000000000001
   cout << ratio_add<mega, micro>::type::den << endl; // 1000000

   cout << ratio_multiply<ratio<7,-21>, ratio<9,-33>>::type::num << endl;
   cout << ratio_multiply<ratio<7,-21>, ratio<9,-33>>::type::den << endl;

   cout << "micro = ";
   printRatio<micro>();

   cout << "7/-21 = ";
   printRatio<ratio<7,-21> >();

   cout << "(7/-21)*(9/-33) = ";
   printRatio<ratio_multiply<ratio<7,-21>, ratio<9,-33>>::type >();

   cout << "kilo (=1000)*milli(=1/1000) = ";
   printRatio<ratio_multiply<kilo, milli>::type >();
}
```

27.7 Zeit und Dauer

In Ergänzung zu den in Abschnitt 35.10 behandelten C-Funktionen zur Bearbeitung und Auswertung der Systemzeitinformation stellt der Header `<chrono>` Klassen im Namespace `std::chrono` zur Verarbeitung von Zeitinformationen und Berechnung von Zeitdauern zur Verfügung. Das Beispiel zeigt, wie mit Zeiten gerechnet werden kann und wie Zeiten gemessen werden können. `sleep_for(dauer)` lässt das Programm die angegebene Dauer untätig sein, während `sleep_until(zeitpunkt)` das Programm bis zu einem bestimmtem Zeitpunkt schlafen legt. Weitere Einzelheiten finden Sie in [ISOC++].

Listing 27.10: Zeitfunktionen

```
// cppbuch/k27/chrono/main.cpp
#include<chrono>
#include<iostream>
#include <thread>

using namespace std;
using namespace std::chrono;

int main() {
   seconds fastEinTag = hours(23) + minutes(59) + seconds(59);
   cout << "23h 59m 59s sind "
        << fastEinTag.count() << " Sekunden" << endl;

   // Zeitmessung
   auto anfang = system_clock::now();
   // Rechnung, um Zeit vergehen zu lassen ....
   cout << "Rechnung läuft ..." << endl;
   double summe = 0.0;
   for(long i = 0; i < 200000000L; ++i) {
      summe += i/10000.0;
   }
   // Rechnung beendet, Dauer ausgeben:
   nanoseconds ns = system_clock::now() - anfang;
   cout << "Dauer [ns] = " << ns.count() << endl;
   cout << "Dauer [us] = "
        << duration_cast<microseconds>(ns).count() << endl;
   cout << "Dauer [ms] = "
        << duration_cast<milliseconds>(ns).count() << endl;
   cout << "Dauer [s] = "
        << duration_cast<seconds>(ns).count() << endl;

   cout << "1500 ms warten ..." << endl;
   this_thread::sleep_for(milliseconds(1500));

   cout << "warten bis (jetzt + 5 Sekunden) ..." << endl;
   auto dann = system_clock::now() + seconds(5);
   this_thread::sleep_until(dann);
}
```

27.8 Hüllklasse für Referenzen

Es kann vorkommen, dass ein Objekt an eine Funktion, die eine Übergabe per Wert erwartet, übergeben werden soll, obwohl das Erzeugen einer Kopie des Objekts nicht sinnvoll oder nicht möglich ist. Der erste Fall tritt auf, wenn der Aufrufer der Funktion das übergebene Objekt direkt beeinflussen will – die Kopie in der Funktion ist aber unerreichbar. Ein Beispiel dafür finden Sie in Abschnitt 13.3 auf Seite 429, wo es darum geht, die Kopie eines dem Konstruktor der Klasse `thread` übergebenen Objekts zu verhindern.

Um ein nicht zu kopierendes Objekt dennoch an eine Funktion, die eine Übergabe per Wert erwartet, übergeben zu können, bietet C++ die Hüllklasse (englisch *wrapper*) `reference_wrapper` an; der Header ist `<functional>`. Im Folgenden nehme ich zur Vereinfachung an, dass das folgende Template mit einer Klasse instanziiert wird, für deren Objekte `operator()(ArgTypes&&...)` aufgerufen werden kann. Damit ergibt sich die im Vergleich zu [ISOC++] etwas vereinfachte Schnittstelle für `reference_wrapper`:

```
template <class T>
class reference_wrapper {
public :
   typedef T type;
   reference_wrapper(T&);
   reference_wrapper(const reference_wrapper<T>& x);
   reference_wrapper& operator=(const reference_wrapper<T>& x);
   operator T& () const;
   T& get() const;       // gibt gespeicherte Referenz zurück
   // Aufruf
   template <class... ArgTypes>
   typename result_of<T(ArgTypes...)>::type
   operator() (ArgTypes&&...) const;
};
```

Der Rückgabetyp ergibt sich aus dem Aufruf `get().operator()(ArgTypes&&...)`. `result_of` ist ein Hilfstemplate, das den Rückgabetyp ermittelt, und dessen Interna nicht bekannt sein müssen. Die Klasse `CallWrapper` im Listing auf Seite 430 ist ein einfaches Äquivalent zu `reference_wrapper`.

Tipp

Es ist einfacher, die Klasse `reference_wrapper` nicht direkt zu benutzen, sondern die Funktion `ref()`, die ein `reference_wrapper`-Objekt zurückgibt.

Beispiel mit `reference_wrapper`:

```
Worker worker;
std::reference_wrapper<Worker> aufrufer(worker);
thread t(aufrufer);   // Thread anlegen und starten
```

Beispiel mit `ref()`:

```
Worker worker;
thread t(std::ref(worker));  // Thread anlegen und starten
```